BERTONE, RODOLFO; THOMAS, PABLO
Introducción a las Bases de Datos. Fundamentos y Diseño
Primera Edición, Buenos Aires
Prentice Hall - Pearson Education, 2011.
ISBN: 978-987-615-136-8
Formato: 21 x 27 cm
Páginas XXXII + 460 = 492 páginas

EDICIÓN:
Magdalena Browne
magdalena.browne@pearsoned.cl

DISEÑO DE INTERIORES:
Carlos E. Capuñay R.

DISEÑO DE PORTADA:
Bookestudio

DIAGRAMACIÓN:
Rosario Capuñay R.

ISBN: 978-987-615-136-8

Impreso en Argentina por Gráfica Pinter S.A./ T-Tres S.R.L.
Printed in Argentina

Prentice Hall
es una marca de
PEARSON

Introducción a las Bases de Datos

Fundamentos y Diseño

Rodolfo Bertone
Pablo Thomas

Prentice Hall

Perú • Argentina • Bolivia • Brasil • Chile • Colombia • Costa Rica
España • Guatemala • México • Puerto Rico • Uruguay • Venezuela

"Así que Yo les digo: pidan, y se les dará;
busquen, y encontrarán;
llamen, y se les abrirá la puerta.
Porque todo aquel que pide, recibe;
y el que busca, encuentra;
y al que llama, se le abrirá":

Jesús de Nazareth

Para Ale, Vicky y Nacho, las tres razones de mi vida.

Para mis viejos, que desde el Cielo me ayudan todo el tiempo.

Rodolfo Bertone

Para Mónica, Ezequiel y Serena, ejes de mi vida.

A mis padres, que contribuyeron a que llegue a este "lugar".

Pablo Thomas

Índice

Presentación

La creciente evolución de la informática en todos sus aspectos no debe menospreciar la importancia que tiene la formación básica de un alumno. Parte de este aprendizaje está relacionado con el manejo de la información, particularmente con el significado, la construcción y el uso de una base de datos.

La función académica obliga al docente a describir los conceptos de manera clara, con el fin de que el alumno tenga la posibilidad de comprender cada tema en forma precisa.

El propósito fundamental de este libro es presentar los contenidos de un curso introductorio de Bases de Datos. Al respecto, se puede encontrar un sinnúmero de material de reconocida validez y de diferentes alcances. Este trabajo está desarrollado bajo las pautas que los autores emplean en el dictado de la asignatura Introducción a las Bases de Datos, en la Facultad de Informática de la Universidad Nacional de La Plata (UNLP), Argentina. Se analizan una serie de temas introductorios a las bases de datos, tales como los aspectos físicos de archivos y su composición estructural; conceptos que permiten comprender el comportamiento, en tiempos de respuesta, de una base de datos.

La investigación sobre bases de datos no es un tema cerrado. En los últimos cuarenta años, la forma de modelar, relacionada con los tipos de gestores de bases de datos, ha ido evolucionando. Las necesidades del mercado han marcado gran parte de este cambio. Si se comparan las necesidades de un sistema de información de las décadas de 1970, 1980, 1990 y la primera de 2000, se puede comprobar que, desde simples datos parcialmente estructurados, se ha migrado hacia estructuras mucho más complejas que incluyen, actualmente, imágenes, sonido y video, además de información general. Las bases de datos deben, entonces, estar capacitadas para soportar este tipo de información, y resolver las necesidades del usuario de una manera simple y básicamente eficiente.

Sin embargo, este material no pierde de vista que está orientado a un primer curso de Bases de Datos, o a aquel lector capacitado que desee comprender un mecanismo sencillo, con el fin de generar un modelo de datos que soporte un sistema de información. Por esta razón, se dedica una sección completa –la Sección III– al modelado de datos asociado a problemas, donde se ejemplifican casos simples, dentro de lo que se conoce como sistemas de gestión clásicos.

Es casi inevitable que cada obra presente los temas de acuerdo con la visión de cada autor. Este libro no es una excepción en ese aspecto. No se presenta una metodología innovadora, pero sí una forma de enfrentar los problemas de modelado de datos, que resume visiones de varios autores. En cada caso se han tomado aquellas consideradas fundamentales, y se ha tratado de resumirlas de una manera clara y efectiva.

Contenidos. Estructura del libro

El libro está dividido en seis secciones. Cada una de ellas presenta temas, *a priori* independientes entre sí, lo que le permitirá al lector realizar su lectura sin hacerlo desde el comienzo del libro.

Las secciones definidas son las siguientes: "Conceptos básicos", "Aspectos físicos de bases de datos", "Modelado de datos", "Procesamiento de consultas", "Seguridad e integridad de los datos" y "Conceptos avanzados".

La Sección I contiene un solo capítulo dedicado a presentar las definiciones iniciales y los motores de bases de datos. Además, describe los propósitos y características fundamentales de una base de datos.

La Sección II está orientada a los aspectos físicos de una base de datos. Esto incluye la administración de archivos, necesaria para brindar soporte físico al modelo de datos. La sección comienza desde el concepto de archivos. En tal sentido, se presentan los dos primeros capítulos, donde se introduce el concepto de archivo desde sus orígenes. En este punto, es menester mencionar la gran similitud que existirá en los Capítulos 2 y 3 de este material con respecto al libro *Algoritmos, datos y programas...*, de esta misma editorial, dado que el autor de ambos materiales es la misma persona.

Dentro de la misma sección, el resto de los capítulos brinda conceptos más avanzados sobre archivos, directamente relacionados con la administración y la recuperación eficiente de la información contenida en ellos. El Capítulo 4 describe archivos con registro de longitud variable y el procesamiento de bajas de información. A partir del Capítulo 5, el objetivo principal se centra en la recuperación eficiente de información, para lo cual este capítulo introduce el concepto de índice con sus ventajas y desventajas. Este tema continúa en el Capítulo 6, que revisa los conceptos de árboles especialmente como estructuras de datos utilizadas para la organización de índices, con énfasis en los árboles binarios y sus deficiencias al efecto.

El Capítulo 7 está orientado a la presentación de las estructuras de árboles balanceados, como política de administración de índices. En él se describen los árboles *B*, *B** y, posteriormente, *B+* como solución integral al problema de índices. Además, se menciona una de las políticas posibles para una organización balanceada.

Por último, el Capítulo 8 presenta la técnica de dispersión o *hashing*, analizando sus características y comparando los resultados obtenidos, básicamente, con los árboles *B+*.

La Sección III, como ya se anticipó, está dedicada al modelado de datos. Aquí se brinda una visión del modelado de datos en tres etapas, denominadas conceptual, lógica y física. Cada una de ellas se vincula con diferentes etapas en un proyecto de ingeniería de *software*.

El Capítulo 9 está dedicado al análisis de los conceptos iniciales de modelado, y se introduce formalmente el modelo entidad-relación.

Los Capítulos 10, 11 y 12 presentan con detalle las etapas de modelado antes enunciadas.

Por último, el Capítulo 13 analiza por separado los conceptos de normalización. Si bien se parte desde un modelo no normalizado y se finaliza en un modelo completamente normalizado, se hace especial énfasis en el estudio de las cuatro primeras formas normales (1FN; 2FN; 3FN; BCFN o FNBC). La 4FN y las formas normales superiores son presentadas de manera resumida.

La Sección IV está dedicada al procesamiento de consultas. Una vez afianzados los conceptos de construcción de un modelo, el lector debe comprender cómo operar con la base de datos generada. El Capítulo 14 describe los lenguajes de consulta básicos: álgebra relacional, cálculo relacional de tuplas y cálculo relacional de dominios, mientras que el Capítulo 15 presenta los conceptos básicos del lenguaje de consulta SQL; en general, se preservan los aspectos definidos por el estándar internacional ANSI SQL-92. Además, como modo de trabajo alternativo se presenta otro lenguaje de consulta, QBE.

El Capítulo 16, en tanto, está dedicado al proceso de optimización de consultas.

La Sección V del libro trata sobre los aspectos de seguridad e integridad de los datos. Para ello, los Capítulos 17, 18 y 19 presentan los conceptos básicos de transacciones y métodos de recuperación, comenzando desde el procesamiento en entornos monousuarios, y su evolución a entornos concurrentes. Allí se describen conceptos básicos de concurrencia, sin profundizar en estos, dado que están fuera del alcance de este material.

Por último, la Sección VI presenta conceptos avanzados vinculados con bases de datos distribuidas, bases de datos orientadas a objetos, *data warehouse* y *data mining*. Estos temas son explicados en forma somera, con la inclusión de las principales características de cada concepto.

Acerca de los autores

Rodolfo Bertone es docente de la asignatura Introducción a las Bases de Datos, en la UNLP, desde 1995. Actualmente, es profesor titular; la titulación alcanzada es de magíster en Ingeniería de Software, y su tema de tesis, "Bases de datos distribuidas". Además, es Consejero Académico de la Facultad de Informática de la misma universidad desde su normalización, en 1999. Tiene a su cargo las cátedras de Bases de Datos 1 y 2, de la carrera de Informática, en la Universidad Nacional del Noroeste de la Provincia de Buenos Aires (UNNOBA), y la cátedra de Fundamentos de Bases de Datos, en la Universidad Nacional de la Patagonia Austral (UNPA), sede Caleta Olivia, donde es profesor visitante. También es docente visitante de la Facultad de Ingeniería de la Universidad Nacional de la Patagonia "San Juan Bosco" (UNPSJB), en Comodoro Rivadavia (provincia del Chubut), y profesor del posgrado de la UNLP. Asimismo, es profesor en varias materias de posgrado tanto en esta última casa de estudios como en varias universidades de Argentina.

Pablo Thomas es docente de la asignatura Introducción a las Bases de Datos, en la UNLP, desde el año 2004. Actualmente, es profesor adjunto. Además, es profesor del magíster en Ingeniería de Software de la Facultad de Informática de la misma universidad, en los cursos de Administración de Proyectos de Software e Ingeniería de Requerimientos; la titulación alcanzada es de magíster en Ingeniería de Software. Es, también, Consejero Académico de la Facultad de Informática de la UNLP desde el año 2003.

Reconocimientos

Es fundamental reconocer a varios actores que influyeron directamente en la construcción de *Introducción a las Bases de Datos. Fundamentos y diseño*.

A todos los integrantes del cuerpo docente de la asignatura que en estos últimos quince años realizaron aportes para mejorarla y, de esta forma, fueron partícipes en menor o mayor grado en aportar ideas que fueron de utilidad para la confección de este libro. Aquí, los autores hacemos una mención especial al A.C. Ariel Sobrado, jefe de Trabajos Prácticos de la asignatura Introducción a las Bases de Datos de la UNLP, y a la Srta. Danae López, auxiliar de cátedra, que participaron en la revisión de algunos capítulos de este libro, aportando una visión más cercana al alumno. También se extiende el agradecimiento a la Lic. Ana Smail, jefa de Trabajos Prácticos en las asignaturas Bases de Datos 1 y 2 de la UNNOBA, que ha aportado ejemplos y ejercicios al presente material.

Además, agradecemos a la Lic. Soledad Antonetti y al Lic. Ariel Miglio, por su participación en la construcción de la herramienta CASER.

Por último, deseamos agradecer a la Facultad de Informática de la UNLP, representada por sus autoridades, por haber permitido generar este material.

Guía para la utilización de este libro

Muchas son las posibilidades que el lector tiene a su alcance con este material. Este fue concebido para mantener la estructura que los autores llevan adelante, en su asignatura de la Facultad de Informática de la UNLP.

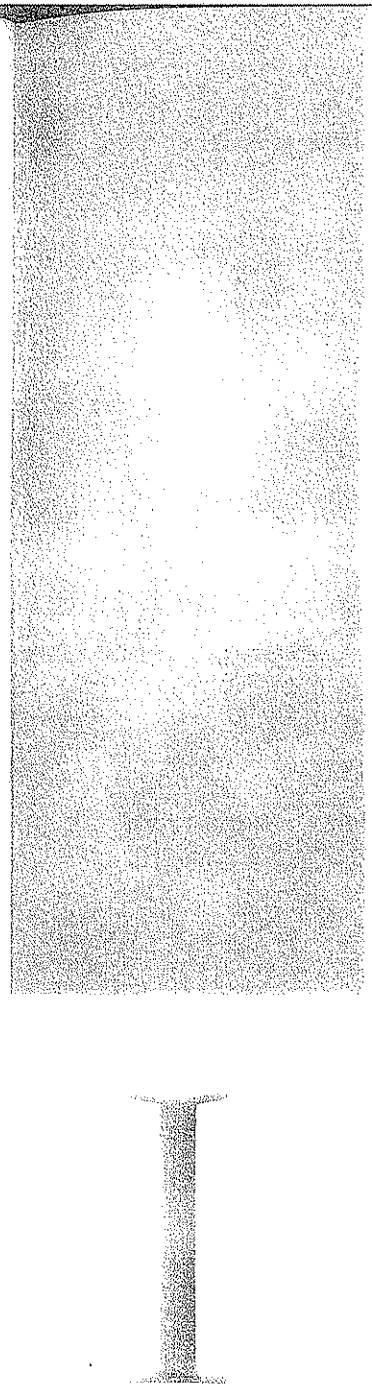
El libro está desarrollado para ser abordado por estudiantes o graduados en Informática (o ramas afines), cuyo interés sea comprender los fundamentos de Bases de Datos. En los primeros capítulos, donde se incluye algorítmica, es menester que el lector conozca conceptos básicos de programación. La algorítmica empleada es procedural, con la utilización de Pascal como lenguaje de programación –dadas su amplia difusión y su fácil lectura–, el cual fue considerado suficiente para detallar los programas incluidos en este libro.

El lector que aborde este material puede decidir sin mayores inconvenientes cuál es su punto de partida. Si bien las secciones están relacionadas entre sí, aquella persona que necesite conocer o discutir otra metodología para el modelado de datos deberá enfocar su lectura en la Sección III. Asimismo, si por ejemplo desea comprender cómo trabaja SQL, bastará con acceder al Capítulo 15.

Conceptos básicos

CAPÍTULO 1

Bases de datos y gestores de bases de datos



CAPÍTULO 1

Bases de datos y gestores de bases de datos

Introducción

Hasta hace pocos años, hablar de una "Base de Datos" se reducía al ámbito informático. Actualmente, este concepto se ha popularizado, producto de la inserción de la informática en la vida cotidiana de las personas, específicamente en actividades donde se necesite "guardar" algún tipo de información relevante y posteriormente recuperarla bajo algún criterio específico. No obstante, es pertinente aclarar al lector que el estudio de los conceptos básicos de Bases de Datos requiere el conocimiento de terminología específica que se supone tiene un alumno de una carrera terciaria de informática en sus primeros años.

La informática evoluciona a un ritmo vertiginoso. Esto provoca una constante innovación de tecnologías de desarrollo de *software* y almacenamiento de datos. A pesar de esto, es necesario el estudio de los temas elementales, en nuestro caso, referidos a Bases de Datos.

Este capítulo es de carácter introductorio y presenta una definición de Base de Datos. Luego, se exponen los sistemas Gestores de Bases de Datos, los distintos niveles de visión y las distintas categorías de usuarios que se pueden establecer.

Conceptos generales de bases de datos

Una de las definiciones más divulgadas de una **"Base de Datos" (BD)** se refiere a una "colección o conjunto de datos interrelacionados". La expresión **colección** o **conjunto**, en este caso, conduce a la idea

hipotética de "muchos" datos. Por datos, la Real Academia Española entiende "cualquier información dispuesta de manera adecuada para su tratamiento por una computadora"[1]; e interrelación indica que los datos no están aislados sino que se vinculan entre sí, integrándose para formar un "todo", siendo el "todo" la Base de Datos.

La definición analizada, si bien tiene consenso, no expresa una característica fundamental de estos datos interrelacionados, que es el propósito por el cual los datos se interrelacionan, propósito asociado a la resolución de algún problema del mundo real.

Por lo tanto, se considerará "Base de Datos", a una colección o conjunto de datos interrelacionados con un propósito específico vinculado a la resolución de un problema del mundo real.

Esta definición transmite una idea de simplicidad, dado que no indica ninguna pauta específica respecto de cómo los datos se vinculan entre sí, ni cómo representan la información del problema a resolver.

Por ende, es necesario establecer dos cuestiones esenciales que posteriormente se analizarán a lo largo de este libro:

- La selección de la información a representar de un dominio de problema específico. Dicho en otras palabras, la realización del proceso de abstracción.
- El diseño de un esquema de representación para esos datos que cumpla con ciertas características deseables que posteriormente se enunciarán.

Conceptos generales de Gestores de Bases de Datos

Un **Sistema de Gestión de Bases de Datos (SGBD)** consiste en un conjunto de programas necesarios para acceder y administrar una BD.

Actualmente, cualquier sistema de *software* necesita interactuar con información almacenada en una BD y para ello requiere del soporte de un SGBD. Por lo tanto, no es posible separar una BD del SGBD.

Un SGBD posee dos tipos diferentes de lenguajes: uno para especificar el esquema de una BD, y el otro para la manipulación de los datos.

La definición del esquema de una BD implica el diseño de la estructura que tendrá efectivamente la BD; describir los datos, las relaciones

1 Note el lector que dato e información son utilizados como sinónimos, y esa consideración será asumida para todo el libro.

entre ellos, la semántica asociada y las restricciones de consistencia. Para esto se utiliza un lenguaje especial, llamado **Lenguaje de Definición de Datos (LDD)**. El resultado de compilar lo escrito con el LDD es un archivo llamado Diccionario de Datos. Un Diccionario de Datos es un archivo con Metadatos, es decir, datos acerca de los datos.

Para manipular los datos de una BD, se utiliza un **Lenguaje de Manipulación de Datos (LMD)**, mediante el cual se puede recuperar información, agregar nueva información y modificar o borrar datos existentes. Existen dos tipos de LMD:

- **LMD procedimentales:** el usuario debe especificar **qué** datos requiere y **cómo** obtener esos datos.
- **LMD no procedimentales:** el usuario debe especificar **qué** datos requiere, sin detallar **cómo** obtener esos datos.

Los LMD no procedimentales son más fáciles de utilizar, ya que su aprendizaje es más sencillo.

Los objetivos más relevantes de un SGBD son:

- **Controlar la concurrencia:** varios usuarios pueden acceder a la misma información en un mismo periodo de tiempo. Si el acceso es para consulta, no hay inconvenientes, pero si más de un usuario quiere actualizar el mismo dato a la vez, se puede llegar a un estado de inconsistencia que, con la supervisión de del SGBD, se puede evitar.
- **Tener control centralizado:** tanto de los datos como de los programas que acceden a los datos.
- **Facilitar el acceso a los datos:** dado que provee un lenguaje de consulta para recuperación rápida de información.
- **Proveer seguridad para imponer restricciones de acceso:** se debe definir explícitamente quiénes son los usuarios autorizados a acceder a la BD.
- **Mantener la integridad de los datos:** esto implica que los datos incluidos en la BD respeten las condiciones establecidas al definir la estructura de la BD (por ejemplo, que determinado dato no tenga valor negativo) y que, ante una falla del sistema, se posea la capacidad de restauración a la situación previa.

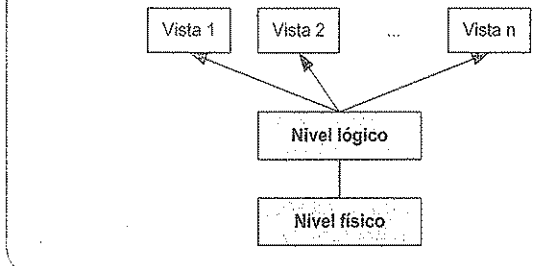
Los SGBD, por lo tanto, proveen los mecanismos necesarios para poder definir y manipular una BD, y si bien no serán objeto de estudio de este texto, no se puede soslayar su importancia.

Niveles de visión de los datos

Para que un Sistema de *Software* sea útil, éste debe interactuar con una BD y recuperar su información en el menor tiempo posible. Bajo esta circunstancia, la representación de la información de una BD muchas veces utiliza estructuras complejas. Esta complejidad se oculta a los usuarios a través de distintos niveles de abstracción, para poder simplificar la interacción cuando estos usuarios no son especializados. Es así como es posible establecer tres niveles (ver Figura 1.1)

- **Nivel de Vista:** corresponde al nivel más alto de abstracción. En este nivel se describe parcialmente la BD, solo la parte que se desea ver. Es posible generar diferentes vistas de la BD, cada una correspondiente a la parte a consultar.
- **Nivel Lógico:** en este nivel se describe la BD completa, indicando qué datos se almacenarán y las relaciones existentes entre esos datos. El resultado es una estructura simple que puede conducir a estructuras más complejas en el nivel físico.
- **Nivel Físico:** este es el nivel más bajo de abstracción, en el cual se describe cómo se almacenan realmente los datos. Se detallan las estructuras de datos más complejas de bajo nivel.

FIGURA 1.1



Modelo de datos

Según la Real Academia Española, un modelo es un "esquema teórico, generalmente en forma matemática, de un sistema o de una realidad compleja, como la evolución económica de un país, que se elabora para facilitar su comprensión y el estudio de su comportamiento".

Una Base de Datos tiene una estructura compuesta por varios elementos, de los cuales el modelo de datos es fundamental. El modelo de datos es un conjunto de herramientas conceptuales que permite describir los datos y su semántica, sus relaciones y las restricciones de integridad y consistencia.

Existen tres grupos de modelos de datos:

- Modelos lógicos basados en objetos.
- Modelos lógicos basados en registros.
- Modelos físicos.

Los modelos basados en objetos son utilizados para describir datos en los niveles de vista y lógico; mientras que los basados en registros son utilizados para describir datos en los niveles lógico y físico. Ambos son de interés para este libro.

Modelos lógicos basados en objetos

Estos modelos son utilizados para representar la información de un problema del mundo real con un esquema de alto nivel de abstracción. Dentro de estos modelos se destacan el modelo de entidades y relaciones, y el modelo orientado a objetos.

El modelo de datos entidad relación (ER) está integrado por entidades y sus relaciones.

Se define como entidad a un objeto de la realidad que se describe por un conjunto de características denominadas atributos. Actualmente, este modelo es el más utilizado debido a su simplicidad y, al mismo tiempo, a su potencia expresiva. Se analizará en más detalle en la Sección 3.

El modelo orientado a objetos representa la realidad con objetos de similares características a los objetos del mundo real. Si bien su alto nivel de abstracción es adecuado para generar representaciones comprensibles, este modelo no será tema de estudio en este libro.

Modelos lógicos basados en registros

Estos modelos utilizan la estructura de datos de registro para almacenar la información en una Base de Datos. Existen tres modelos basados en registros: modelo jerárquico, modelo de red y modelo relacional.

La esencia del modelo jerárquico consiste en utilizar registros vinculados entre sí formando una estructura de árbol.

El modelo en red también utiliza registros con vínculos que generan una estructura de grafo.

El modelo relacional, que se detalla en el Capítulo 9, utiliza un conjunto de tablas para representar la información. Una tabla se compone por filas, denominadas tuplas, y por columnas. Cada fila representa un registro, y cada columna, un campo o atributo. Estas tablas se vinculan entre sí y establecen relaciones de integridad entre los datos que las componen. Este modelo es el más utilizado desde hace tres décadas.

Categorías de usuarios de bases de datos

Como se mencionó al inicio de este capítulo, el uso de BD no se restringe solo al ámbito informático. En consecuencia, los potenciales usuarios pueden clasificarse en las siguientes categorías:

- **Administrador de BD:** este papel lo cumple la persona que tiene el control centralizado de la BD a través del SGBD. Sus funciones más importantes, entre otras, son definir el esquema de la BD (Diccionario de Datos), otorgar derechos de acceso a la BD a otros usuarios y establecer las restricciones de integridad entre los datos para mantener la consistencia.
- **Programadores de aplicaciones:** son los profesionales del ámbito informático que desarrollan sistemas de *software*.
- **Usuarios sofisticados:** realizan consultas a la BD a través de un lenguaje de consulta; generalmente, son profesionales del ámbito informático.
- **Usuarios especializados:** son aquellos usuarios que desarrollan aplicaciones no tradicionales (Sistemas Expertos, Sistemas de Información Geográfica, conocidos como GIS, entre otros).
- **Usuarios normales:** son los denominados usuarios clásicos que utilizan los sistemas de *software* desarrollados por los usuarios antes descriptos; por lo tanto, su acceso es indirecto.

Cuestionario del capítulo

1. ¿Qué es una Base de Datos?
2. ¿Qué es un Sistema Gestor de Bases de Datos (SGBD)? ¿Cuáles son sus propósitos? ¿Qué lenguajes incluye?
3. ¿Qué niveles de visión se pueden establecer sobre los datos?
4. ¿Qué es un modelo de datos? Enumere tres modelos de datos.
5. ¿Qué tipos de usuarios puede tener una Base de Datos?
6. ¿Qué diferencia encuentra entre un Administrador de Base de Datos y un Programador de Aplicaciones que interactúan con Bases de Datos?

II

SE

Aspectos físicos de BD

CAPÍTULO 2

Archivos, estructuras y operaciones básicas

Objetivo

El objetivo de este capítulo es presentar conceptos básicos de archivos, comenzando desde su definición, analizando sus propiedades esenciales, así como las operaciones elementales que permiten desarrollar la algorítmica más simple sobre ellos.

Si bien la orientación de los algoritmos propuestos es procedimental y cercana al lenguaje Pascal, se presentan los programas y procedimientos en un seudocódigo que permite al lector abordar los temas sin un conocimiento acabado de ese lenguaje de programación.

Se discuten sobre el final del capítulo, además, las alternativas para borrar información en el archivo, las cuales serán desarrolladas en detalle en el Capítulo 4.

Archivos

Los datos que necesitan ser preservados más allá de la ejecución de un algoritmo deben residir en archivos. Para que esto suceda, un archivo no puede residir en la memoria RAM, sino que debe estar en dispositivos de almacenamiento permanente de información. El ejemplo más natural como medio permanente para contener información lo constituye el disco rígido.

Definiciones

Se presentan a continuación algunas definiciones de archivo de datos.

Un **archivo** es una colección de registros semejantes, guardados en dispositivos de almacenamiento secundario de la computadora (Wiederhold, 1983).

Un **archivo** es una estructura de datos que recopila, en un dispositivo de almacenamiento secundario de una computadora, una colección de elementos del mismo tipo (De Giusti, 2001).

Un **archivo** es una colección de registros que abarcan entidades con un aspecto común y originadas para algún propósito particular.

Estas definiciones muestran que un archivo es una estructura de datos homogénea. Los archivos se caracterizan por el crecimiento y las modificaciones que se efectúan sobre éstos. El crecimiento indica la incorporación de nuevos elementos, y las modificaciones involucran alterar datos contenidos en el archivo, o quitarlos.

Aspectos físicos

Las estructuras de datos, en general, se definen y utilizan sobre la memoria RAM de la computadora y tienen existencia mientras el programa se encuentra activo. Por lo tanto, cuando es necesario tener persistencia en la información que maneja un algoritmo, esta información debe ser almacenada en archivos.

Los archivos residen en dispositivos de memoria secundaria o almacenamiento secundario, los cuales son medios que están ubicados fuera de la memoria RAM, y que son capaces de retener información luego que el programa finaliza o luego de apagar la computadora. Algunos ejemplos de dispositivos de almacenamiento secundario, entre otros, son los discos rígidos, los discos flexibles (*diskettes*), los discos ópticos, las cintas.

Existen dos diferencias básicas entre las estructuras de datos que residen en la memoria RAM y aquellas que lo hacen en almacenamiento secundario:

- **Capacidad:** la memoria RAM tiene capacidad de almacenamiento más limitada que, por ejemplo, un disco rígido.
- **Tiempo de acceso:** la memoria RAM mide su tiempo de acceso en orden de nanosegundos (10^{-9} segundos), en tanto que el disco rígido lo hace en el orden de milisegundos (10^{-3} segundos).

No es propósito de este libro profundizar sobre los aspectos de *hardware* de una computadora. Queda para el lector obtener, en libros dedicados a esos temas, mayores detalles acerca de las diferencias entre las

memorias principales (RAM) y secundarias (discos magnéticos u ópticos, o memorias Flash). Solamente se enfatiza la diferencia sustancial existente en los tiempos de acceso a estos dispositivos. Los análisis en capítulos posteriores están orientados a crear estructuras de archivos que puedan brindar tiempos de respuesta razonables, teniendo en cuenta las diferencias de velocidades entre las memorias RAM y secundarias.

Administración de archivos

Se deben considerar dos aspectos fundamentales cuando se trabaja con archivos:

- Visión física
- Visión lógica

Los archivos residen en memoria secundaria. En un disco rígido pueden coexistir un gran número de archivos. La visión física de un archivo tiene que ver con el almacenamiento de este en memoria secundaria. Es responsabilidad del sistema operativo de una computadora resolver cuestiones relativas al lugar de almacenamiento de archivo en disco, cómo se ubicará la información en el mismo y cómo será recuperada. Nuevamente, para este libro no son aspectos a profundizar y no serán detallados.

Los archivos son manejados por algoritmos. Desde el punto de vista práctico, un algoritmo no referencia al lugar físico donde el archivo reside, sino que tiene una visión lógica de este; es decir que lo utiliza como si estuviera almacenado en la memoria RAM. Desde un algoritmo se debe establecer una "conexión" con el sistema operativo, y este será el responsable de determinar el lugar de residencia del archivo en disco y su completa administración.

A partir de las visiones anteriores, se deben distinguir dos conceptos diferentes de archivo pero interrelacionados:

- **Archivo físico:** es el archivo residente en la memoria secundaria y es administrado (ubicación, tipos de operaciones disponibles) por el sistema operativo.
- **Archivo lógico:** es el archivo utilizado desde el algoritmo. Cuando el algoritmo necesita operar con un archivo, genera una conexión con el sistema operativo, el cual será el responsable de la administración. Esta acción se denomina independencia física.

Antes que el programa pueda operar sobre el archivo, el sistema operativo debe recibir instrucciones para hacer un enlace entre el nombre lógico que utilizará el algoritmo y el archivo físico. Cada lenguaje de programación define una instrucción para tal fin.

Tipos de archivos

Los archivos de datos pueden analizarse desde diversas perspectivas. Es posible entonces definir el tipo de archivo por la información que contiene, por la estructura de información que contiene o por la organización física del mismo en el dispositivo de memoria secundaria.

Desde la perspectiva de la información que contiene, un archivo puede contener un tipo de dato simple: entero, real o carácter. Además, puede contener una estructura heterogénea como lo son los registros. De acuerdo con las necesidades de cada algoritmo, debe definirse el tipo de dato que el archivo necesite.

A partir de lo definido anteriormente, se infiere que un archivo contiene elementos del mismo tipo. Esto lo convierte en una estructura homogénea de datos. Los registros, en general, poseen longitud fija, determinada por la suma de las longitudes de los campos que los componen. Cuando un archivo se genera a partir de registros con este formato, se denomina archivo de longitud fija y predecible de los elementos de datos contenidos. El resto del capítulo tratará acerca de archivos con estas características. Además, es posible contar con registros con longitudes variables, que se adapten a las necesidades propias de cada dato. En estos casos, los archivos generados tendrán una lógica de trabajo diferente, la cual será discutida en el Capítulo 4.

Acceso a información contenida en los archivos

Básicamente, se pueden definir tres formas de acceder a los datos de un archivo:

- **Secuencial:** el acceso a cada elemento de datos se realiza luego de haber accedido a su inmediato anterior. El recorrido es, entonces, desde el primero hasta el último de los elementos, siguiendo el orden físico de estos.
- **Secuencial indizado:** el acceso a los elementos de un archivo se realiza teniendo presente algún tipo de organización previa, sin tener en cuenta el orden físico.

Suponga el lector que se dispone de un archivo con los nombres de los empleados de una empresa. Los datos de dicho archivo están almacenados en el orden en el cual fueron ingresados; sin embargo, a partir de un acceso secuencial indizado, es posible recuperar los datos en forma ordenada, es decir, como si hubiesen sido ingresados alfabéticamente.

- **Directo:** es posible recuperar un elemento de dato de un archivo con un solo acceso, conociendo sus características, más allá de que exista un orden físico o lógico predeterminado.

Estos modos de acceso resultan de fundamental importancia para esta Sección Sí bien en este capítulo se tratarán archivos con acceso secuencial para comprender su operatoria básica, en el resto de la Sección se presentarán las desventajas de este acceso, su bajo nivel de *performance* y la necesidad de contar con mejores organizaciones para la administración de archivos que representen en forma más adecuada a las bases de datos.

Operaciones básicas sobre archivos

Para poder operar con archivos, son necesarias una serie de operaciones elementales disponibles en todos los lenguajes de programación que utilicen archivos de datos. Estas operaciones incluyen:

- La definición del archivo lógico que utilizará el algoritmo y la relación del nombre lógico del archivo con su almacenamiento en el disco rígido.
- La definición de la forma de trabajo del archivo, que puede ser la creación inicial de un archivo o la utilización de uno ya existente.
- La administración de datos (lectura y escritura de información).

Estas operaciones elementales se traducen en una serie de instrucciones en la que cada lenguaje define la operatoria detallada sobre archivos.

El lector debe tener en cuenta que en los ejemplos del libro se utilizará una notación seudocódigo Pascal.

Definición de archivos

Como cualquier otro tipo de datos, los archivos necesitan ser definidos. Se reserva la palabra clave `file` para indicar la definición del archivo. La siguiente sentencia define una variable de tipo `archivo`:

```
Var archivo_logico: file of tipo_de_dato;
```

donde `archivo_logico` es el nombre de la variable, `file` es la palabra clave reservada para indicar la definición de archivos y `tipo_de_dato` indica el tipo de información que contendrá el archivo.

Otra opción para definir archivos se presenta a continuación:

```
Type archivo= file of tipo_de_dato;

Var archivo_logico: archivo;
```

en este caso, se define un tipo `archivo` y, luego, una variable `archivo_logico` del tipo definido anteriormente.

El Ejemplo 2.1 muestra la definición de algunos archivos para almacenar tanto tipos de datos simples como compuestos.

EJEMPLO 2.1

```
Type
     Persona = record
          DNI                  :string[8]
          ApellidoyNombre      :string [30];
          Direccion            :string [40];
          Sexo                 :char;
          Salario              :real;
     end;
     ArchivodeEnteros     = file of integer;
     ArchivodeReales      = file of real;
     ArchivodeCaracteres  = file of char;
     ArchivodeLinea       = file of string;
     ArchivodePersonas    = file of Persona;

Var
     Enteros       : ArchivodeEnteros;
     Reales        : ArchivodeReales;
     Caracteres    : ArchivodeCaracteres;
     Texto         : ArchivodeLinea;
     Personas      : ArchivodePersonas
```

Correspondencia archivo lógico-archivo físico

Como se indicó anteriormente, el sistema operativo de la computadora es el responsable de la administración del archivo en disco. El algoritmo utiliza un tipo de datos `file` también definido anteriormente, que representa el nombre lógico del mismo. Se debe, entonces, indicar que el archivo lógico utilizado por el algoritmo se corresponde con el archivo físico administrado por el sistema operativo. La sentencia encargada de hacer esta correspondencia es:

```
Assign (nombre_logico, nombre_fisico)
```

donde `nombre_logico` es la variable definida en el algoritmo, y `nombre_fisico` es una cadena de caracteres que representa el camino donde quedará (o ya se encuentra) el archivo y el nombre del mismo.

El Ejemplo 2.2 presenta, a partir de la definición del Ejemplo 2.1, las asignaciones correspondientes. Observaciones a partir del ejemplo:

- En algunos casos, coinciden los nombres lógicos con la variable de tipo `string` definida para el nombre físico, pero esto no representa ningún inconveniente en la definición.
- Algunas definiciones de nombre físico contienen la extensión que se quiere dar al archivo.
- El archivo `archivodelinea` definido como `file of string` contendrá elementos de 255 caracteres de longitud en cada caso.

EJEMPLO 2.2

```
Type
     Persona = record
          DNI                  :string[8]
          ApellidoyNombre      :string [30];
          Direccion            :string [40];
          Sexo                 :char;
          Salario              :real;
     end;
     ArchivodeEnteros     = file of integer;
     ArchivodeReales      = file of real;
     ArchivodeCaracteres  = file of char;
     ArchivodeLinea       = file of string;
     ArchivodePersonas    = file of Persona;

Var
     Enteros       : ArchivodeEnteros;
     Reales        : ArchivodeReales;
     Caracteres    : ArchivodeCaracteres;
     Texto         : ArchivodeLinea;
     Personas      : ArchivodePersonas

Begin
     {comienzo del programa}
     Assign( Enteros,'c:\archivos\enteros.dat');
     Assign( Reales,'c:\archivos\numerosreales');
     Assign( Caracteres,'c:\archivos\letras.dat');
     Assign( Texto, 'c:\archivos\linea');
     Assign( Personas, 'c:\archivos\pers.dat');
     ....

End.
```

Apertura y creación de archivos

Hasta el momento, se ha detallado cómo definir un archivo y se ha establecido la relación con el nombre físico. Para operar con un archivo desde un algoritmo, se debe realizar la apertura.

Para abrir un archivo, existen dos posibilidades:

- Que el archivo aún no exista porque aún no fue creado.
- Que el archivo exista y se quiera operar con él.

Para ello, se dispone de dos operaciones diferentes:

- La operación `rewrite` indica que el archivo va a ser creado y, por lo tanto, la única operación válida sobre el mismo es escribir información.
- La operación `reset` indica que el archivo ya existe y, por lo tanto, las operaciones válidas sobre el mismo son lectura/escritura de información.

Estas operaciones definen entonces la modalidad de trabajo que se podrá realizar sobre el algoritmo. El lector debe notar la importancia y utilización de cada una de ellas. Si se intentara abrir como lectura/escritura un archivo inexistente, la operación `reset` generaría un error. Si se realizara la apertura de un archivo con la operación `rewrite`, y el archivo existiera previamente en disco, se borraría toda la información en él contenida, comenzando con un archivo vacío.

El formato de las dos operaciones es el siguiente:

```
rewrite(nombre_logico)
```

```
reset(nombre_logico)
```

donde `nombre_logico` es la variable definida previamente.

Cierre de archivos

Una vez finalizada la operatoria sobre un archivo, el mismo debe ser cerrado. Cerrar un archivo significa una serie de eventos:

- Cada archivo tiene un comienzo y necesariamente debe contener un fin. La marca de fin de archivo se conoce por su sigla en inglés `EOF` (*end of file*) y servirá posteriormente para controlar las lecturas sobre el archivo y no exceder la longitud del mismo.
- Transferir definitivamente la información volcada sobre el archivo a disco, es decir, transferir el *buffer* a disco.

La operación de cierre de archivo es:

```
close(nombre_logico)
```

donde `nombre_logico` corresponde a la variable de tipo `file` del archivo que se desea cerrar.

Lectura y escritura de archivos

Para leer o escribir información en un archivo, las instrucciones son:

```
read(nombre_logico, var_dato);
```

```
write(nombre_logico, var_dato);
```

en ambos casos, `nombre_logico` corresponde al nombre lógico del archivo desde donde se desea leer o escribir. En tanto, `var_dato` corresponde a la variable del algoritmo que contendrá la información a transferir al archivo o la variable que recibirá la información desde el archivo, según el caso.

La variable `var_dato` y el tipo de dato de los elementos del archivo `nombre_logico` deben coincidir. Además, en caso de que `var_dato` corresponda a una variable de tipo registro, no debe realizarse una lectura o escritura campo a campo.

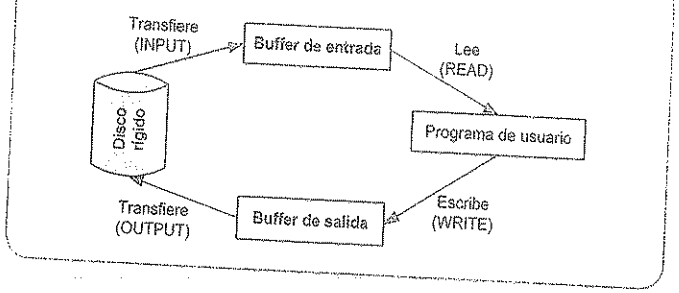
Buffers de memoria

Las lecturas y escrituras desde o hacia archivo se realizan sobre *buffers*. Se denomina *buffer* a una memoria intermedia (ubicada en RAM) entre un archivo y un programa, donde los datos residen provisoriamente hasta ser almacenados definitivamente en la memoria secundaria, o donde los datos residen una vez recuperados de dicha memoria secundaria.

La justificación de lo anterior se debe a cuestiones de *performance*. Mientras que una operación sobre la memoria RAM tiene una demora del orden de nanosegundos, una operación sobre disco tiene una demora del orden de milisegundos. Por lo tanto, varias operaciones de lectura y/o escritura directamente sobre la memoria secundaria reducirían notablemente la *performance* de un algoritmo. Por este motivo, el sistema operativo, al controlar dichas operaciones, intenta minimizar el tiempo requerido para estas.

La operación `read` lee desde un *buffer* y, en caso de no contar con información, el sistema operativo realiza automáticamente una operación `input`, trayendo más información al *buffer*. La diferencia radica en que cada operación `input` transfiere desde el disco una serie de registros. De esta forma, cada determinada cantidad de instrucciones `read`, se realiza una operación `input`. Cada `read` se mide en nanosegundos, en tanto que los `input` se miden en milisegundos, pero se realizan con menor frecuencia.

Figura 2.1



De forma similar procede la operación `write`; en este caso, se escribe en el *buffer*, y si no se cuenta con espacio suficiente, se descarga el *buffer* a disco por medio de una operación `output`, dejándolo nuevamente vacío. (Figura 2.1)

Esta forma de trabajo optimiza la *performance* de lectura y escritura sobre archivos.

Creación de archivos

Se presentan a continuación algunos ejemplos de algoritmos que crean archivos de datos, cuya información se lee desde teclado.

El Ejemplo 2.3 crea un archivo de números reales, bajo las siguientes precondiciones:

- Los números se obtienen desde teclado hasta recibir el número cero, el cual no se incorpora al archivo.
- El nombre del archivo es determinado desde el mismo algoritmo.

EJEMPLO 2.3

```
Program CrearArchivo;
Type
     ArchivodeReales      = file of real;

Var
     Reales          : ArchivodeReales;
     NroReal         : real;

Begin
     {enlace entre el nombre lógico y el nombre físico}
     Assign( Reales,'c:\archivos\numerosreales');

     {apertura del archivo para creación}
     rewrite( Reales );
```

continúa >>>


```
     {lectura de un número real}
     read( NroReal );

     while (NroReal <> 0)do
       begin
          {escritura del número en el archivo}
          write( Reales, NroReal );

          {lectura de otro número real}
          read( nro );
       end;

     {cierre del archivo}
     close( Reales );

end.
```

El Ejemplo 2.4 crea un archivo con datos de personas, bajo las siguientes precondiciones:

- Los datos se leen de teclado y finalizan cuando se lee un DNI nulo o vacío.
- El nombre del archivo se obtiene desde teclado.

En este algoritmo, el lector debe notar la diferencia entre leer información desde el teclado (lo cual debe resolverse campo a campo) y la escritura en el archivo, donde se indica solamente el registro.

EJEMPLO 2.4

```
Programa CrearArchivo;
Type
     Persona = record
          DNI                  :string[8]
          ApellidoyNombre      :string [30];
          Direccion            :string [40];
          Sexo                 :char;
          Salario              :real;
     end;
     ArchivodePersonas    = file of Persona;

Var
     Personas         : ArchivodePersonas;
     Nombrefisico     : string[12];
     Per              : Persona;

Begin
    write( 'Ingrese el nombre del archivo:' );
    readln(NombreFisico)
```

continúa >>>

```
        {enlace entre el nombre lógico y el nombre físico}
        Assign( Personas, NombreFisico);

        {apertura del archivo para creación}
        rewrite( Personas );

        {lectura del DNI de una persona}
        readln( per.DNI );

        while (per.DNI <> '') do
          begin
              {lectura del resto de los datos de la persona}
              readln(per.ApellidoyNombre);
              readln(per.Direccion);
              readln(per.Sexo);
              readln(per.Salario);

              {escritura del registro de la persona en el archivo}
              write( Personas, per );

              {lectura del DNI de una nueva persona}
              readln( per.DNI );
          end;

      {cierre del archivo}
      close( Personas );

end.
```

En ambos casos, la apertura del archivo se realiza mediante la instrucción `rewrite` dado que el archivo aún no existe, se está creando.

Operaciones adicionales con archivos

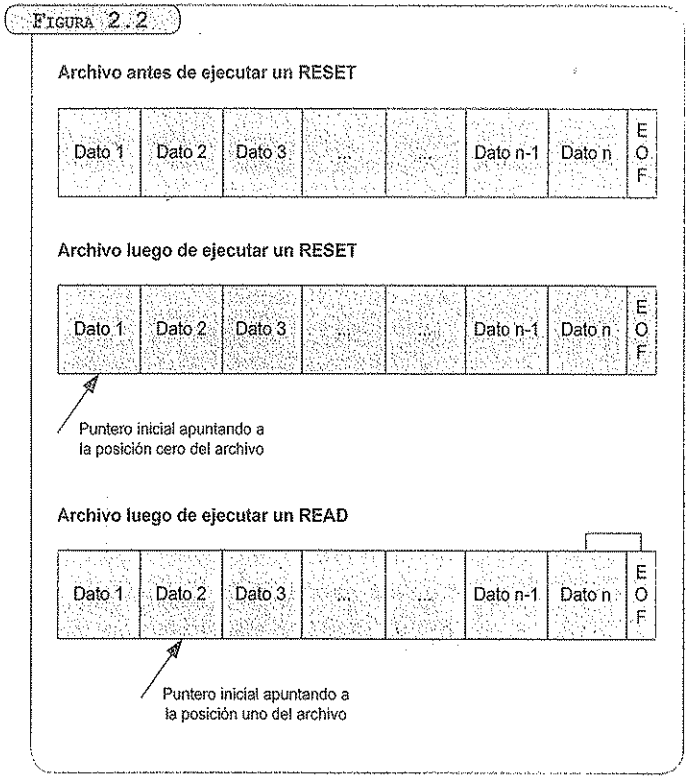
Hasta el momento, fueron presentadas las operaciones básicas sobre archivos de datos. Son necesarias, además, una serie de operaciones adicionales que permitirán operar con el archivo una vez creado. Estas operaciones son:

- Control de fin de datos.
- Control de tamaño del archivo.
- Control de posición de trabajo dentro del archivo.
- Ubicación física en alguna posición del archivo.

Puntero de trabajo de un archivo

En el momento de ejecutarse la instrucción `reset`, el sistema operativo coloca un puntero direccionando al primer registro disponible dentro del archivo. Este puntero, relacionado al nombre lógico del archivo, indicará en todo momento la posición actual de trabajo. Esta posición actual direcciona al elemento que retornará una instrucción `read` o el lugar donde se escribirá un dato con la instrucción `write`. Tanto la instrucción `read` como la `write` avanzan en forma automática el puntero una posición luego de ejecutarse. (Figura 2.2)

FIGURA 2.2



Control de fin de datos

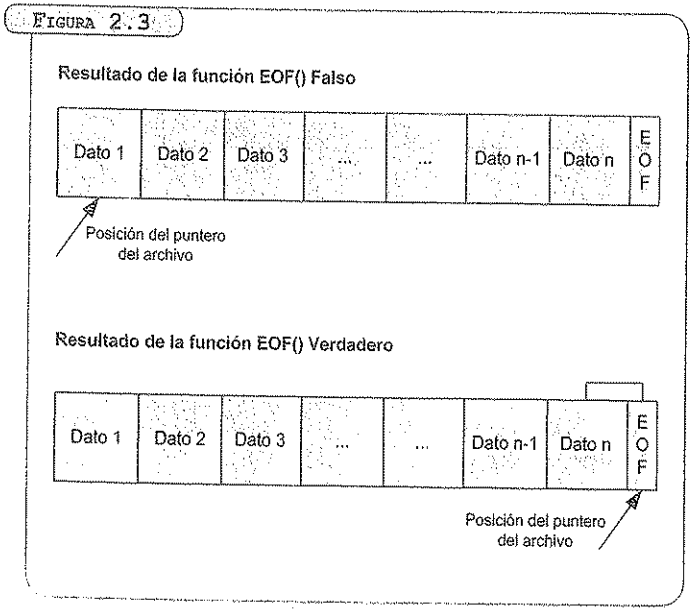
Para poder recorrer un archivo de datos existente, se debe realizar la lectura de cada uno de sus elementos. Como se dijo anteriormente, el tipo de acceso que tendrán los archivos en los Capítulos 2 y 3 corresponde al acceso secuencial. De esta forma, los datos son obtenidos en el orden en que fueron ingresados, desde el comienzo y hasta el final. Es necesario, entonces, contar con una operación que permita controlar la recuperación de datos sin exceder el fin de archivo.

Un algoritmo debe controlar el fin de archivo antes de realizar una operación de lectura. La función

```
eof(nombre_logico);
```

es la encargada de resolver este control. Aquí, `nombre_logico` corresponde al archivo que se está evaluando. La función retornará verdadero si el puntero del archivo referencia a EOF, y falso en caso contrario. (Figura 2.3.)

Figura 2.3



Note el lector que si un algoritmo realiza una operación de lectura y el puntero del archivo se encuentra apuntando a EOF, se genera un error en la ejecución del programa.

Control de tamaño del archivo

En determinados algoritmos, es necesario conocer la cantidad de elementos que conforman un archivo. Se dispone para tal fin de una función que retorna un valor entero indicando cuántos elementos se encuentran almacenados en un archivo de datos.

```
filesize(nombre_logico);
```

Control de posición de trabajo dentro del archivo

Para conocer la posición actual del puntero del archivo, se utiliza la función

```
Filepos(nombre_logico)
```

la cual retorna un número entero que indica la posición actual del puntero. Dicho valor estará siempre comprendido entre cero y la cantidad de elementos que tiene el archivo (`filesize`).

Ubicación física en alguna posición del archivo

Con cada operación de lectura y/o escritura, avanza automáticamente el puntero lógico del archivo. Este movimiento del puntero debe ser, en algunas circunstancias, manejado por el algoritmo. Para modificar la posición del puntero, se dispone de la instrucción

```
seek(nombre_logico, posición)
```

donde `nombre_logico` indica el puntero del archivo a modificar y `posición` debe ser un valor entero (o variable de tipo entera) que indica el lugar donde será posicionado el puntero.

Note el lector que `posición` debe ser un valor comprendido entre cero y `filesize(nombre_logico)`. En su defecto, la instrucción generará un error de ejecución.

Lectura de archivos

El algoritmo siguiente al de creación de un archivo debe permitir leer los elementos contenidos en el mismo. Así, el Ejemplo 2.5 presenta el código correspondiente a leer el archivo generado en el Ejemplo 2.4.

EJEMPLO 2.5

```
Procedure Recorrido(var Personas: ArchivodePersonas );

var Per: Persona; {para leer elementos del archivo}

begin

     {dado que el archivo está creado, debe abrirse como de
     lectura/escritura}
     reset(Personas);

     while not eof(Personas)do
       begin

          {se obtiene un elemento desde archivo}
          read( Personas, Per );

          {se presenta por pantalla cada dato del elemento leido}
          writeln( Per.DNI );
          writeln( Per.ApellidoyNombre );
          writeln( Per.Direccion );
          writeln( Per.Sexo );
          writeln( Per.Salario );
          writeln;
       end;

     {cierre del archivo}
     close( Personas );

end;
```

A partir del Ejemplo 2.5:

- El lector puede observar que el ejemplo se presenta como un módulo, en lugar de un programa. En este módulo es importante notar la forma en que se pasa el parámetro correspondiente al nombre lógico del archivo. Si bien el algoritmo no modifica el archivo, y por lo tanto el parámetro podría ser pasado por valor, el mismo se envía por referencia. Se debe notar, entonces, que el parámetro correspondiente al nombre lógico de un archivo no se maneja de forma similar que el resto de los parámetros. El nombre lógico del archivo siempre representa un nexo con el nombre físico y, por ende, es necesario pasarlo siempre a un módulo por referencia.
- Además, no se realiza la asignación nombre lógico-nombre físico dentro del módulo. Esta debe, en general, resolverse en el cuerpo del programa principal.
- El archivo debe abrirse como lectura. La instrucción prevista para tal fin es `reset`.
- La lectura del archivo se realiza registro a registro, pero la impresión en pantalla debe resolverse campo a campo.

Agregar más información a un archivo

Otro tipo de algorítmica básica sobre archivos consiste en incorporar nueva información a los mismos. Siguiendo el Ejemplo 2.4, sobre el archivo de personas se incorporan nuevos datos; el Ejemplo 2.6 muestra esta acción.

EJEMPLO 2.6

```
Procedure Agregar (var Personas: ArchivodePersonas );

     Procedure Leer( var Per:Persona);
     Begin
          readln( Per.DNI );
          readln( Per.ApellidoyNombre );
          readln( Per.Direccion );
          readln( Per.Sexo );
          readln( Per.Salario );
     End;

var  Per: Persona; {para leer elementos del archivo}
begin
     {dado que el archivo está creado, debe abrirse como de
     lectura/escritura}
     reset(Personas);
     {posicionamiento al final del archivo}
     seek( Personas, filesize(Personas));
     {lectura de datos de una persona}
     Leer(Per)
     while (Per.DNI<>'')do
       begin
          {se agrega una persona al archivo}
          write(Personas, Per)
          {lectura de datos de otra persona}
          Leer(Per)
     end;
   {cierre del archivo}
   close( Personas );
end;
```

Las conclusiones, en este caso, son:

- Nuevamente, el problema se resuelve con un módulo, donde se respeta la forma de pasaje de parámetros anteriormente descripta.
- El archivo debe abrirse nuevamente con la instrucción `reset`, esto dado a que debe incorporar nuevos datos manteniendo los anteriores. Si se hubiese realizado una apertura con la instrucción `rewrite`, entonces los datos ya almacenados en el archivo desaparecerían.

- La instrucción `filesize(nombre_logico)` determina cuántos registros tiene el archivo. El lector debe recordar que, en general, los archivos numeran a sus registros a partir del registro cero. Por lo tanto, la instrucción `seek(nombre_logico, filesize(nombre_logico))` dejará el puntero lógico del archivo referenciando a EOF, y las escrituras posteriores quitarán la marca.
- La orden `close(nombre_logico)` incorpora nuevamente la marca sobre el nuevo final del archivo.

Modificar la información de un archivo

El último ejemplo, correspondiente a la denominada algorítmica básica sobre archivos, corresponde a su vez a aquel problema donde se plantea la necesidad de modificar los datos contenidos en un archivo. Manteniendo la estructura del problema inicial, el Ejemplo 2.7 presenta un módulo que permite modificar el salario de las personas, realizando a este un incremento de 15%.

EJEMPLO 2.7

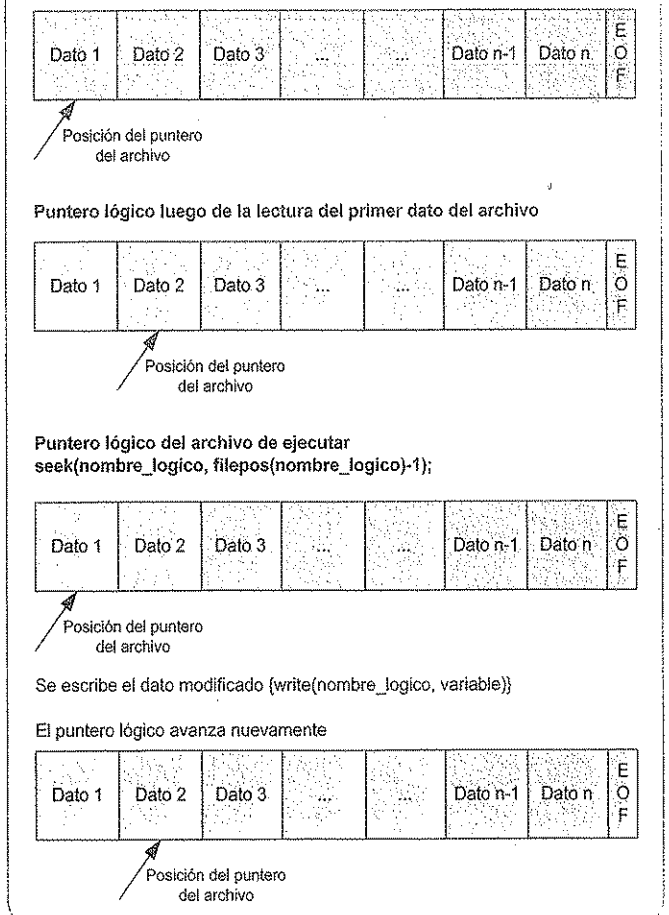
```
Procedure Actualizar(var Personas: ArchivodePersonas );
var Per: Persona; {para leer elementos del archivo}
begin
    {dado que el archivo está creado, debe abrirse como de
    lectura/escritura}
    reset(Personas);
    while not eof (Personas)do
     begin
        Read( Personas, Per);
        {modificación del salario}
        Per.salario := Per.salario * 1.15;
        {ubicar al puntero del archivo en el registro leído}
        Seek( Personas,filepos(Personas) -1 );
        {se graba la persona con salario modificado}
        write(Personas, Per)
     end;
   {cierre del archivo}
   close( Personas );
end;
```

Las conclusiones, en este caso, son:

- Luego de efectuar una operación de lectura sobre el archivo, el puntero lógico se desplaza una posición hacia adelante. Por lo tanto, es necesario retroceder una posición antes de proceder a la escritura del registro modificado.

- Luego, la operación de escritura lleva al puntero nuevamente sobre el siguiente registro. La Figura 2.4 presenta gráficamente el caso anterior.

FIGURA 2.4



Puntero lógico del archivo de ejecutar
`seek(nombre_logico, filepos(nombre_logico) -1);`

Se escribe el dato modificado
`{write(nombre_logico, variable) }`

Proceso de bajas

Las operaciones esenciales sobre archivos son:

- **Alta:** ingresar nuevos datos al archivo.
- **Modificación:** alterar el contenido de algún dato del archivo.
- **Consulta:** presentar el contenido total o parcial del archivo.
- **Baja:** quitar información del archivo.

Hasta el momento, se describieron, en líneas generales, las tres primeras operaciones. El proceso de baja debe ser discutido con mucho mayor nivel de detalle. Esto se debe a que, en general, cuando se opera con bases de datos, las circunstancias donde se desea quitar información de un archivo están muy limitadas. El proceso de baja será discutido en detalle en el Capítulo 4, así como las circunstancias que lo hacen poco frecuente.

Cuestionario del capítulo

1. ¿Qué es un archivo? ¿En qué se diferencia de otras estructuras de datos conocidas por el lector?
2. ¿Cuáles son las principales características a tener en cuenta cuando se opera con un archivo? ¿Qué es el *buffering*?
3. ¿Qué tipos de accesos reconoce el lector sobre un archivo?
4. ¿En qué se diferencia el nombre lógico de un archivo del nombre físico?
5. ¿Por qué existen dos formas de abrir un archivo, `reset` y `rewrite`?
6. ¿Por qué es importante que un algoritmo contenga la instrucción `close`?
7. ¿Cómo trabaja el puntero lógico del archivo?

Ejercitación

1. Siguiendo la lógica del Ejercicio 2.4, el lector deberá realizar un algoritmo que permita crear un archivo que contenga números enteros.
2. Realice un algoritmo que permita guardar en un archivo todos los productos que actualmente se encuentran en venta en un determinado negocio. La información que es importante considerar es: nombre del producto, cantidad actual, precio unitario de venta, tipo de producto (que puede ser comestible, de limpieza o vestimenta).
3. Teniendo como base al ejercicio anterior, plantee un algoritmo que permita presentar en pantalla los productos del archivo que correspondan al rubro de limpieza o cuya cantidad actual supere las 100 unidades.
4. Realice un algoritmo que permita modificar los precios unitarios de los productos del archivo generado en el Ejercicio 2, incrementando en 10% los correspondientes al rubro comestibles, en 5% a los de limpieza y en 20% a los de vestimenta.

CAPÍTULO 3

Algorítmica clásica sobre archivos

Objetivo

El objetivo del Capítulo 3 está relacionado con el desarrollo de algoritmos considerados clásicos en la operatoria de archivos secuenciales. Estos algoritmos se resumen en tres tipos: de actualización, *merge* y corte de control.

El primer caso, con todas sus variantes, permite introducir al alumno en problemas donde se actualiza el contenido de un archivo resumen o "maestro", a partir de un conjunto de archivos con datos vinculados a ese archivo maestro.

En el segundo caso, se dispone de información distribuida en varios archivos que se reúne para generar un nuevo archivo, producto de la unión de los anteriores.

Por último, el corte de control, muy presente en la operatoria de BD, determina situaciones donde, a partir de información contenida en archivos, es necesario generar reportes que resuman el contenido, con un formato especial.

Proceso de actualización de archivos

Si bien el Capítulo 2 presentó algoritmos denominados de actualización, esto se realizó bajo la consigna de cambiar algún dato de un archivo. Esto es, y siguiendo con el Ejemplo 2.7, modificar el salario de las personas contenidas en el archivo. En este apartado, el contenido de un archivo será modificado por el contenido de otro archivo.

Se deben introducir, en este punto, los conceptos de archivo maestro y archivo detalle, que se continuarán utilizando en el resto del capítulo. Se denomina archivo maestro al archivo que resume información sobre un dominio de problema específico. Ejemplo: el archivo de productos de una empresa que contiene el *stock* actual de cada producto.

Por otra parte, se denomina archivo detalle al archivo que contiene novedades o movimientos realizados sobre la información almacenada en el maestro. Ejemplo: el archivo con todas las ventas de los productos de la empresa realizadas en un día particular.

En el resto del apartado, se plantean diversas situaciones que ejemplifican el proceso de actualización. Es muy importante que el lector analice las precondiciones que en cada caso se plantean. Los algoritmos propuestos tienen en cuenta tales precondiciones que, en caso de no ser cumplidas, determinan la falla de su ejecución.

Actualización de un archivo maestro con un archivo detalle (I)

Presenta la variante más simple del proceso de actualización. Las precondiciones del problema son las siguientes:

- Existe un archivo maestro.
- Existe un único archivo detalle que modifica al maestro.
- Cada registro del detalle modifica a un registro del maestro. Esto significa que solamente aparecerán datos en el detalle que se correspondan con datos del maestro. Se descarta la posibilidad de generar altas en ese archivo.
- No todos los registros del maestro son necesariamente modificados.
- Cada elemento del maestro que se modifica es alterado por uno y solo un elemento del archivo detalle.
- Ambos archivos están ordenados por igual criterio. Esta precondición, considerada esencial, se debe a que hasta el momento se trabaja con archivos de datos de acuerdo con su orden físico. Más adelante, se discutirán situaciones donde los archivos respetan un orden lógico de datos.

El Ejemplo 3.1 presenta un caso práctico donde se tienen en cuenta las precondiciones anteriores.

EJEMPLO 3.1

```
program ejemplo_3_1;
  type
    producto = record
      cod: str4;
      descripcion: string[30];
      pu: real;
      stock: integer;
    end;
    venta_prod = record
      cod: str4;
      cant_vendida: integer;
    end;
    detalle = file of venta_prod;
    maestro = file of producto;
  var
    regm: producto;
    regd: venta_prod;
    mae1: maestro;
    det1: detalle;
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle');
  reset (mae1);
  reset (det1);
  while (not eof(det1)) do begin
    read(mae1,regm);
    read(det1,regd);
    {se busca en el maestro el producto del detalle}
    while (regm.cod <> regd.cod) do
      read (mae1,regm);
   {se modifica el stock del producto con la cantidad
    vendida de ese producto}
    regm.stock := regm.stock - regd.cant_vendida;

    {se reubica el puntero en el maestro}
    seek (mae1, filepos(mae1)-1);

    {se actualiza el maestro}
    write(mae1,regm);
  end;
  close(det1);
  close(mae1);
end.
```

Algunas consideraciones del ejemplo anterior son las siguientes:

- El proceso de actualización finaliza cuando se termina de recorrer el archivo detalle. Una vez procesados todos los registros del archivo detalle, el algoritmo finaliza, sin la necesidad de recorrer el resto del archivo maestro.

- Es necesario buscar aquel registro del archivo maestro que se actualiza a partir del archivo detalle. Esto se debe a que no todos los elementos del archivo maestro necesariamente serán modificados.
- Nótese que las lecturas sobre el archivo maestro se realizan sin controlar el fin de archivo. Si bien esta operatoria se indicó como incorrecta en el Capítulo 2, el lector debe notar que, a partir de la tercera precondición planteada, el archivo detalle no genera altas. Por lo tanto, para cada registro del archivo detalle debe necesariamente existir el correspondiente en el archivo maestro.

Actualización de un archivo maestro con un archivo detalle (II)

Este es otro caso, levemente diferente del anterior; solo se modifica una precondición del problema y hace, de esta forma, variar el algoritmo resolutorio. Las precondiciones, en este caso, son las siguientes:

- Existe un archivo maestro.
- Existe un único archivo detalle que modifica al maestro.
- Cada registro del detalle modifica a un registro del maestro. Esto significa que solamente aparecerán datos en el detalle que se correspondan con datos del maestro. Se descarta la posibilidad de generar altas en el maestro.
- Cada elemento del archivo maestro puede no ser modificado, o ser modificado por uno o más elementos del detalle.
- Ambos archivos están ordenados por igual criterio. Esta precondición, considerada muy fuerte, se debe a que hasta el momento se manipulan archivos de datos de acuerdo con su orden físico. Más adelante, se discutirán situaciones donde los archivos respetan un orden lógico de datos.

El Ejemplo 3.2 presenta un seudocódigo donde se intenta resolver el problema planteado.

EJEMPLO 3.2

```
program ejemplo_3_2;
  type
      producto = record
        cod: str4;
        descripcion: string[30];
        pu: real;
        stock: integer;
```

continúa >>>

```
    end;
    venta_prod = record
      cod: str4;
      cant_vendida: integer;
    end;
    detalle = file of venta_prod;
    maestro = file of producto;
var
  regm: producto;
  regd: venta_prod;
  mael: maestro;
  detl: detalle;
  cod_actual: str4;
  tot_vendido: integer;
begin
  assign (mael, 'maestro');
  assign (detl, 'detalle');
  reset (mael);
  reset (detl);
  while (not eof(detl)) do
    begin
      read(mael,regm);
      read(detl,regd);
      {se busca en el maestro el producto del detalle}
      while (regm.cod <> regd.cod) do
       read (mael,regm);

      {se totaliza la cantidad vendida del detalle}
       cod_actual := regd.cod;
       tot_vendido := 0;
       while (regd.cod = cod_actual) do
         begin
           tot_vendido := tot_vendido + regd.cant_vendida;
           read(detl,regd);
         end;

       {se modifica el stock del producto con la cantidad vendida
       de ese producto}
       regm.stock := regm.stock - tot_vendido;

      {se reubica el puntero en el maestro}
       seek (mael, filepos(mael)-1);

      {se actualiza el maestro}
       write(mael,regm);

    end;
    close(detl);
    close(mael);
end.
```


Si se realiza un análisis detallado de la solución, se encontrará que, entre este ejemplo y el anterior, solo se agrega una iteración que permite agrupar todos los registros del detalle que modificarán a un elemento del maestro, y a partir de ese agrupamiento, actualizar efectivamente el registro correspondiente del maestro.

Sin embargo, esta solución presenta algunos inconvenientes. Se puede observar que la segunda operación read sobre el archivo detalle se hace sin controlar el fin de datos de este último. Esto podría solucionarse agregando una selección (`if`) que permita controlar dicha operación. Pero cuando finaliza la iteración interna, al retornar a la iteración principal, se lee otro registro del archivo detalle, perdiendo de esa forma un dato ya leído.

La solución presentada en el Ejemplo 3.2 no es correcta: genera inconvenientes, y las posibles soluciones son modificaciones forzadas para lograr una solución.

Por la forma de trabajo de los archivos en lenguajes como Pascal, este tipo de problemas requiere implantar soluciones diferentes. El inconveniente se plantea con la operación `read` sobre el archivo detalle. Es necesario leer fuera de la iteración principal del algoritmo y, más adelante, dentro de la iteración. Por lo tanto, el control del `EOF` debe resolverse de otra manera. El Ejemplo 3.3 presenta una solución alternativa.

EJEMPLO 3.3

```
program ejemplo_3_3;
  const valoralto='9999';
  type str4 = string[4];
       producto = record
         cod: str4;
         descripcion: string[30];
         pu: real;
         cant: integer;
       end;
       venta_prod = record
         cod: str4;
         cant_vendida: integer;
       end;
       detalle = file of venta_prod;
       maestro = file of producto;
  var
    regm: producto;
    regd: venta_prod;
    mael: maestro;
    detl: detalle;
    total: integer;
    aux: str4;
```

continúa >>>

```
procedure leer (var archivo:detalle; var dato:venta_prod);
    begin
      if (not eof(archivo))
       then
         read (archivo,dato)
       else
         dato.cod:= valoralto;
    end;

{programa principal}
begin
    assign (mael, 'maestro');
    assign (det1, 'detalle');
    reset (mael);
    reset (det1);
    read(mael,regm);
    leer(det1,regd);

   {se procesan todos los registros del archivo detalle}
    while (regd.cod <> valoralto) do
      begin
       aux := regd.cod;
       total := 0;

       {se totaliza la cantidad vendida de productos iguales
       en el archivo de detalle}
       while (aux = regd.cod ) do
         begin
           total := total + regd.cant_vendida;
           leer(det1,regd);
         end;

       {se busca en el maestro el producto del detalle}
       while (regm.cod <> aux) do
         read (mael,regm);

       {se modifica el stock del producto con la cantidad
       total vendida de ese producto}
       regm.cant := regm.cant - total;

       {se reubica el puntero en el maestro}
       seek (mael, filepos(mael)-1);

       {se actualiza el maestro}
       write(mael,regm);

       {se avanza en el maestro}
       if (not eof (mael))
         then
           read(mael,regm);

   end;

 close (det1);
 close (mael);

end.
```


Puede observarse ahora que se presenta una solución que utiliza un procedimiento de lectura, y un control de fin de proceso que va más allá del control del EOF.

El procedimiento de lectura, denominado `leer`, es el responsable de realizar el `read` correspondiente sobre el archivo, en caso de que este tuviera más datos. En caso de alcanzar el fin de archivo, se asigna a la variable `dato.cod`, por la cual el archivo está ordenado, un valor imposible de alcanzar en condiciones normales de trabajo. Este valor indicará que el puntero del archivo ha llegado a la marca de fin.

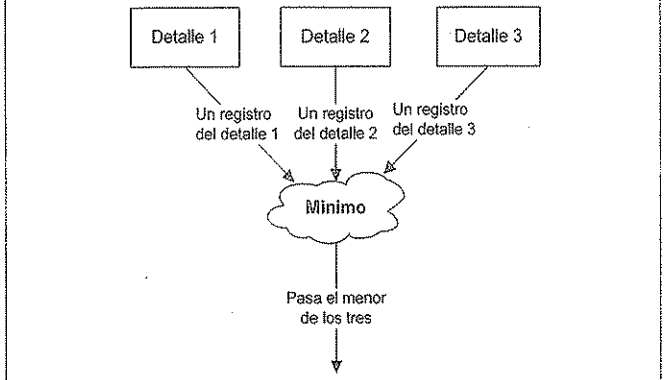
En el módulo principal se controla el fin del procesamiento evaluando el valor que tiene la variable por la cual el archivo está ordenado (regd. cod), controlando que no tome el valor que indique indirectamente EOF.

Actualización de un archivo maestro con *N* archivos detalle

Bajo las mismas consignas del ejemplo anterior, se plantea un proceso de actualización donde, ahora, la cantidad de archivos detalle se lleva a *N* (siendo $N > 1$) y el resto de las precondiciones son las mismas.

El Ejemplo 3.4 presenta la resolución de un algoritmo de actualización a partir de tres archivos detalle. Para ello, se agrega un nuevo procedimiento, denominado `minimo`, que actúa como filtro. El objetivo de este proceso a partir de la información recibida es retornar el elemento más pequeño de acuerdo con el criterio de ordenamiento del problema. En el Ejemplo 3.4, el procedimiento mínimo retorna el código mínimo de producto. La Figura 3.1 presenta la situación gráficamente.

Figura 3.1



El objetivo del procedimiento `minimo` es determinar el menor de los tres elementos recibidos de cada archivo (para poder retornarlo como el más pequeño) y leer otro registro del archivo desde donde provenía ese elemento.

De esta forma, el resto del algoritmo del Ejemplo 3.3 se mantiene igual. El fin de datos se controla con el mismo criterio. En el momento en que los tres archivos detalle se hayan recorrido hasta su último elemento, el menor elemento retornado corresponderá a la marca de fin utilizada. De este modo finaliza el procesamiento.

EJEMPLO 3.4

```
program ejemplo_3_4;

  const valoralto='9999';

  type str4 = string[4];

       producto = record
         cod: str4;
         descripcion: string[30];
         pu: real;
         cant: integer;
       end;

       venta_prod = record
         cod: str4;
         cant_vendida: integer;
       end;

       detalle = file of venta_prod;
       maestro = file of producto;
var

 regm: producto;
 min, regd1, regd2,regd3: venta_prod;
 mae1: maestro;
 det1,det2,det3: detalle;
 aux: str4;
 total_vendido:integer;

procedure leer (var archivo:detalle; var dato:venta_prod);
begin
  if (not eof(archivo))then
    read (archivo,dato)
  else
    dato.cod:= valoralto;
end;

procedure minimo (var r1,r2,r3,min:venta_prod);
begin
  if (r1.cod<=r2.cod) and (r1.cod<=r3.cod)then
    begin
      min := r1;
      leer(det1,r1);
    end
  else
    if (r2.cod<=r3.cod)then
      begin
```

continúa >>>


```
        min := r2;
        leer(det2,r2);
      end
    else
      begin
        min := r3;
        leer(det3,r3)
      end;
end;

{programa principal}
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle1');
  assign (det2, 'detalle2');
  assign (det3, 'detalle3');
  reset (mae1);
  reset (det1);
  reset (det2);
  reset (det3);
  read(mae1,regm);
  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);
  minimo(regd1,regd2,regd3,min);

{se procesan todos los registros de los archivos detalle}
  while (min.cod <> valoralto) do
   begin
     {se totaliza la cantidad vendida de productos iguales en
     el archivo de detalle}
     aux := min.cod;
     total_vendido := 0
     while (aux = min.cod ) do
       begin
         total_vendido := total_vendido + min.cantvendida;
         minimo(regd1,regd2,regd3,min);
       end;

     {se busca en el maestro el producto del detalle}
     while (regm.cod <> min.cod) do
       read(mae1,regm);

     {se modifica el stock del producto con la cantidad total
     vendida de ese producto}
     regm.cant := regm.cant - total;

     {se reubica el puntero en el maestro}
     seek (mae1, filepos(mae1)-1);

     {se actualiza el maestro}
     write(mae1,regm);

     {se avanza en el maestro}
     if (not eof (mae1))
       then read(mae1,regm);
   end;

  close (det1);
  close (det2);
  close (det3);
  close (mae1);

end.
```

Proceso de generación de un nuevo archivo a partir de otros existentes. *Merge*

El segundo grupo de problemas presentado en este capítulo está vinculado con la generación de un archivo que resuma información de uno o varios archivos existentes. Este proceso recibe el nombre de *merge* o unión, y la principal diferencia con los casos previamente analizados radica en que el archivo maestro no existe, y por lo tanto debe ser generado.

Se muestran algunas variantes de este tipo de problemas.

Primer ejemplo

El primer ejemplo plantea un problema muy simple. Las precondiciones son las siguientes:

- Se tiene información en tres archivos detalle.
- Esta información se encuentra ordenada por el mismo criterio en cada caso.
- La información es disjunta; esto significa que un elemento puede aparecer una sola oportunidad en todo el problema. Si el elemento 1 está en el archivo detalle1, solo puede aparecer una vez en este y no podrá estar en el resto de los archivos.

EJEMPLO 3.5

```
program ejemplo_3_5;

  const valoralto='9999';

  type str4 = string[4];

       producto = record
         codigo: str4;
         descripcion: string[30];
         pu: real;
         cant: integer;
       end;

       detalle = file of producto;
var

 min, regd1, regd2,regd3: producto;
 det1, det2, det3, mae1: detalle;

procedure leer (var archivo:detalle; var dato:producto);
begin
```

continúa >>>


```
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.codigo:= valoralto;
end;

procedure minimo (var r1,r2,r3,min:producto);
begin
  if (r1.codigo<=r2.codigo) and (r1.codigo<=r3.codigo) then
    begin
      min := r1;
      leer(det1,r1);
    end
  else
    if (r2.cod<=r3.cod) then
      begin
        min := r2;
        leer(det2,r2);
      end
    else begin
        min := r3;
        leer(det3,r3)
    end;
 end;

{programa principal}
begin
  assign (mae1, 'maestro');
  assign (det1, 'detalle1');
  assign (det2, 'detalle2');
  assign (det3, 'detalle3');

  rewrite (mae1);
  reset (det1);
  reset (det2);
  reset (det3);

  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);

  minimo(regd1,regd2,regd3,min);

  {se procesan todos los registros de los archivos detalle}
   while (min.codigo <> valoralto) do
    begin
      write(mae1, min);
      minimo(regd1,regd2,regd3,min);
    end;

  close (det1);
  close (det2);
  close (det3);
  close (mae1);

end.
```

El Ejemplo 3.5 presenta la solución al problema planteado. Este algoritmo tiene aspectos similares al ejemplo anterior, dado que se utilizan tres archivos de detalle y los procedimientos `minimo` y `leer` se resuelven en forma análoga.

Segundo ejemplo

Como segundo ejemplo se presenta un problema similar, pero ahora los elementos se pueden repetir dentro de los archivos detalle, modificando de esta forma la tercera precondición del ejemplo anterior. El resto de las precondiciones permanecen inalteradas.

EJEMPLO 3.6

```
program ejemplo_3_6;

  const valoralto='9999';

  type str4 = string[4];

       producto = record
          codigo: str4;
          nombre: string[30];
          cant: integer;
       end;
       detalle =  file of producto;
var

 min, regd1, regd2, regd3, regmae: ventas;
 det1, det2, det3, mae1: detalle;
 codprod: str4;
 cantotal: integer;

procedure leer (var archivo:detalle; var dato:producto);
begin
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    codigo:= valoralto;
end;

procedure minimo (var r1,r2,r3,min:producto);
begin
  if (r1.codigo<=r2.codigo) and (r1.cod<=r3.codigo) then
    begin
      min := r1;
      leer(det1,r1);
    end
   else
```

continúa >>>


```
      if (r2.codigo<=r3.codigo)then
        begin
          min := r2;
          leer(det2,r2);
        end
      else
        begin
          min := r3;
          leer(det3,r3)
        end;
 end;

{programa principal}
begin
  assign (mael, 'maestro');
  assign (det1, 'detalle1');
  assign (det2, 'detalle2');
  assign (det3, 'detalle3');

  rewrite (mael);
  reset (det1);
  reset (det2);
  reset (det3);

  leer(det1, regd1);
  leer(det2, regd2);
  leer(det3, regd3);

  minimo(regd1,regd2,regd3,min);

  {se procesan todos los registros de los archivos detalle}
   while (min.codigo <> valoralto) do
    begin
      codprod := min.codigo;
      cantotal := 0;
      {se procesan todos los registros del mismo producto}
      while (codprod=min.codigo)
        begin
          cantotal:= cantotal + min.cant;
          minimo(regd1,regd2,regd3,min);
        end;

      write(mael, min);

    end;

  close (det1);
  close (det2);
  close (det3);
  close (mael);

end.
```

El algoritmo del ejemplo 3.6 expone la solución al problema. Se deberá notar que:

- Los elementos que correspondan al mismo código de producto deben agruparse para resumir la información. Una vez agrupados y resumidos, son almacenados en el archivo maestro.
- La solución utiliza la misma estrategia utilizada para ejemplos anteriores.

Corte de control

Se denomina corte de control al proceso mediante el cual la información de un archivo es presentada en forma organizada de acuerdo con la estructura que tiene el archivo.

Suponga que se almacena en un archivo la información de ventas de una cadena de electrodomésticos. Dichas ventas han sido efectuadas por los vendedores de cada sucursal de cada ciudad de cada provincia del país. Luego, es necesario informar al gerente de ventas de la empresa el total de ventas producidas de acuerdo con el siguiente formato:

Provincia: .....
Ciudad: .....

Sucursal: .....
Vendedor 1 $$

Vendedor n $$
Total sucursal: $$

Sucursal: .....
Vendedor 1 $$

Vendedor n $$
Total sucursal: $$

Total ciudad: $$

Ciudad: .....

Total ciudad: $$
Total prov.: $$
Prov.: .....

Total ciudad: $$

Total prov.: $$

Total empresa: $$

Deben tenerse en cuenta las siguientes precondiciones:

- El archivo se encuentra ordenado por provincia, ciudad, sucursal y vendedor.
- Se debe informar el total de vendido en cada sucursal, ciudad y provincia, así como el total final.
- En diferentes provincias pueden existir ciudades con el mismo nombre, o en diferentes ciudades pueden existir sucursales con igual denominación.

El Ejemplo 3.7 presenta una solución para el problema planteado. El lector podrá notar que el algoritmo utiliza el proceso de lectura definido previamente en este capítulo.

EJEMPLO 3.7

```
program ejemplo_3_7;

  const valoralto="ZZZ";

  type nombre = string[30];

       RegVenta = record
            Vendedor: integer;
            MontoVenta: real;
            Sucursal: nombre;
            Ciudad: nombre;
            Provincia: nombre;
       end;
       Ventas =  file of RegVenta;
var

 reg: RegVenta;
 archivo: Ventas;
 total, totprov, totciudad, totsuc: integer;
 prov, ciudad, sucursal: nombre;

procedure leer (var archivo: Ventas; var dato:RegVenta);
begin
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.provincia:= valoralto;
end;

(programa principal)
begin
  assign (archivo, 'archivoventas');
  reset (archivo);
```

continúa >>>

```
leer(archivo, reg);
total:= 0;

while (reg.Provincia <> valoralto) do
  begin
    writeln("Provincia:", reg.Provincia);
    prov := reg.Provincia;
    totprov := 0;

    while (prov=reg.Provincia)do
      begin
        writeln("Ciudad:", reg.Ciudad);
        ciudad := reg.Ciudad;
        totciudad := 0

        while (prov=reg.Provincia) and (Ciudad=reg.Ciudad)do
          Begin
            writeln("Sucursal:", reg.Sucursal);
            sucursal := regSucursal;
            totsuc := 0;

            while ((prov=reg.Provincia) and
                   (Ciudad=reg.Ciudad) and
                   (Sucursal=regSucursal)) do
              Begin
                write("Vendedor:", reg.Vendedor);
                writeln(reg.MontoVenta);
                totsuc := totsuc + reg.MontoVenta;
                leer(archivo, reg);
              end;

            writeln("Total Sucursal", totsuc);
            totciudad := totciudad + totsuc;
          end;

        writeln("Total Ciudad", totciudad);
        totprov := totprov + totciudad;
      end;

    writeln("Total Provincia", totprov);
    total := total + totprov,
  end;

writeln("Total Empresa", total);

close (archivo);

end.
```


Cuestionario del capítulo

1. ¿Cuáles son los procesos denominados algorítmica clásica sobre archivos?
2. ¿Por qué fue necesario utilizar el procedimiento para leer del archivo en algunos ejercicios del capítulo?
3. ¿Por qué el proceso de *merge* utilizó una apertura del archivo maestro con `rewrite`?
4. ¿Por qué se consideró que los archivos están físicamente ordenados?

Ejercitación

1. Una empresa posee un **archivo ordenado por código de promotor** con información de las ventas realizadas por cada uno de ellos (código de promotor, nombre y monto de venta). Sabiendo que en ese archivo pueden existir uno o más registros por cada promotor, realice un procedimiento que reciba el archivo anteriormente mencionado y lo compacte (esto es, generar un nuevo archivo donde cada promotor aparezca una única vez con sus ventas totales).
2. El encargado de ventas de un negocio desea administrar el *stock* de productos que vende. Para ello, genera un archivo maestro donde figuran todos los productos comercializados. Se maneja la siguiente información: código de barras, nombre comercial, proveedor, *stock* actual, *stock* mínimo. Diariamente se genera un archivo detalle donde se asientan todas las ventas de productos realizadas. Se pide generar un algoritmo que permita actualizar el archivo maestro con el detalle sabiendo que: i) ambos archivos están ordenados por código de barras del producto; ii) cada registro del maestro puede ser actualizado por 0, 1 o más registros del detalle; iii) el archivo detalle solamente contiene registros que están en el archivo maestro.
   a) Realice un proceso que reciba al archivo maestro y genere un listado donde figuren los productos cuyo *stock* está por debajo del mínimo permitido.
   b) Ídem anterior, generando, ahora, un archivo donde figure el nombre de los productos cuyo *stock* está por debajo del mínimo permitido.

3. El servicio meteorológico provincial posee un archivo con información de lluvias registradas mensualmente en las ciudades de la provincia durante el año 2007. Dicho organismo recibe un archivo con la información de los servicios regionales, los cuales indican la cantidad llovida para un mes determinado en su ciudad.

   a. Lea cuidadosamente el ejercicio y defina la estructura de ambos archivos.

   b. Genere ambos archivos.

   c. Actualice el archivo de servicios meteorológicos de la provincia con la información recibida de los servicios regionales.

   Nota: al generar los archivos, suponga que la información ingresa ordenada por ciudad. Además, el archivo de los servicios regionales puede contener varios datos de una misma ciudad.

4. a. Se necesita contabilizar los votos de las diferentes mesas electorales por provincia y localidad. Para ello, se posee un archivo con la siguiente información: código de provincia, código de localidad, número de mesa y cantidad de votos. Realice un listado como se muestra a continuación:

Código de provincia:
Código de localidad Total de votos
....
....
Total de votos por provincia:

Código de provincia:
Código de localidad Total de votos
....
Total de votos por provincia:
....

Nota: la información viene ordenada por código de provincia y código de localidad.

   b. En caso de que tuviera que resolver el inciso a pero sin la precondición de tener el archivo ordenado, ¿cómo lo resolvería? Justifique.

5. Una gran proveeduría deportiva tiene organizado su sistema de forma tal que cada sección genera su propio archivo con código de vendedor, nombre del vendedor, producto vendido, cantidad vendida y precio unitario del producto. Mensualmente, envía sus archivos a la sección administración. Se sabe que la proveeduría está dividida en cinco secciones (zapatillería, vestimenta, pesca, caza y *camping*) y que los vendedores están capacitados para realizar ventas en cualquier sector. Además, se sabe que los archivos están organizados por código de vendedor y que estos pueden realizar varias ventas. Se pide obtener un archivo que registre la información de cada vendedor y el monto total obtenido por sus ventas del mes correspondiente.

CAPÍTULO 4

Eliminación de datos. Archivos con registros de longitud variable

Objetivo

Este capítulo tiene dos objetivos fundamentales. El primero de ellos expone y discute el problema de eliminación de información contenida en archivos. Este proceso se puede llevar a cabo mediante algoritmos que operen de manera física o lógica sobre un archivo con registros de longitud fija. A lo largo de la primera parte del capítulo, se presentan las dos soluciones y se discute en detalle la eficiencia de estas. Asimismo, se presentan propuestas alternativas, que, en gran parte, mejoran la *performance* final del proceso de baja.

El segundo objetivo del capítulo está vinculado con archivos que contienen registros de longitud variable. En relación con este tema, se analizarán alternativas para realizar la algorítmica básica –altas, bajas, modificaciones y consultas–, comparando la eficiencia obtenida con la presentada anteriormente para archivos con registros de longitud fija.

Proceso de bajas

Se denomina **proceso de baja** a aquel proceso que permite quitar información de un archivo.

El proceso de baja puede ser analizado desde dos perspectivas diferentes: aquella ligada con la algorítmica y *performance* necesarias para borrar la información, y aquella que tiene que ver con la necesidad real de quitar información de un archivo en el contexto informático actual.

Históricamente, el proceso de baja era necesario para no mantener información poco útil o relevante en un archivo y, por consiguiente, recuperar espacio en dispositivos de almacenamiento secundario. En la actualidad, las organizaciones que disponen de BD consideran la información como su bien más preciado. De esta forma, el concepto de borrar información queda condicionado. En general, el conocimiento adquirido (información) no se quita, sino que se preserva en archivos o repositorios históricos. Teniendo en cuenta esta situación, el proceso de baja se relativiza en importancia. Sin embargo, en esta sección se presenta el proceso algorítmico inherente, con un análisis general de *performance*.

El proceso de baja puede llevarse a cabo de dos modos diferentes:

- **Baja física:** consiste en borrar efectivamente la información del archivo, recuperando el espacio físico.
- **Baja lógica:** consiste en borrar la información del archivo, pero sin recuperar el espacio físico respectivo.

Baja física

Se realiza baja física sobre un archivo cuando un elemento es quitado del archivo y es reemplazado por otro elemento del mismo archivo, decrementando en uno su cantidad de elementos.

La ventaja de este método consiste en administrar, en cada momento, un archivo de datos que ocupe el lugar mínimo necesario. La desventaja, como se discutirá a continuación, tiene que ver con la *performance* final de los algoritmos que implementan esta solución.

Para realizar el proceso de baja física existen, básicamente, dos técnicas algorítmicas:

- Generar un nuevo archivo con los elementos válidos, es decir, sin copiar los que se desea eliminar.
- Utilizar el mismo archivo de datos, generando los reacomodamientos que sean necesarios.

Baja física generando nuevo archivo de datos

El algoritmo del Ejemplo 4.1 presenta el caso de borrado físico generando un nuevo archivo de datos. La precondición del problema presentado en el ejemplo es:

- Se dispone de un archivo de empleados donde figura el empleado "Carlos García".

EJEMPLO 4.1

```
program ejemplo_4_1;
  const valoralto="ZZZ";
  type
      empleado = record;
            Nombre         : string[50];
            Direccion      : string[50];
            Documento      : string[12];
            Edad           : integer;
            Observaciones: string[200];
      end;
     archivoemple = file of Empleado;
var
 reg: empleado;
 archivo, archivonuevo: archivoemple;
procedure leer (var archivo: archivoemple; var dato:empleado);
    begin
      if (not eof(archivo))
        then
            read (archivo,dato)
        else
            dato.Nombre := valoralto;
    end;

{programa principal}
begin
  assign (archivo, 'archemple');
  assign (archivonuevo, 'archnuevo');
  reset(archivo);
  rewrite(archivonuevo);
  leer(archivo,reg)

  {se copian los registros previos a Carlos Garcia}
  while (reg.Nombre<>"Carlos Garcia")
    begin
       write(archivonuevo,reg)
       leer(archivo,reg)
    end

  {se descarta a Carlos Garcia}
  leer(archivo,reg)

  {se copian los registros restantes}
  while (reg.Nombre<>valoralto) do
    begin
       write(archivonuevo,reg)
       leer(archivo,reg)
    end

  close(archivonuevo);
  close(archivo);
end.
```

El algoritmo presentado consiste en recorrer el archivo original de datos, copiando al nuevo archivo toda la información, menos la correspondiente al elemento que se desea quitar.

Una vez finalizado el proceso, se debe eliminar el archivo inicial del dispositivo de memoria secundaria, renombrando luego al generado con el nombre original.

Un análisis de *performance* básico determina que este método necesita leer tantos datos como tenga el archivo original y escribir todos los datos, salvo el que se elimina. Esto significa, suponiendo que el archivo tiene *n* registros, *n* lecturas y *n*-1 escrituras; en ambos casos, las lecturas y escrituras se realizan en forma secuencial, en el orden en el cual aparecen en el archivo (es decir, para leer el registro *n*, antes se debe leer desde el registro 1 hasta el registro *n*-1) sobre ambos archivos. Este nivel de *performance* será de utilidad para comparar la eficiencia de esta solución con las que se presentarán más adelante en este capítulo.

Se debe notar que, una vez finalizado el proceso de generación del nuevo archivo, coexisten en el disco rígido dos archivos: el original y el nuevo sin el registro borrado. Esto significa que se debe disponer en memoria secundaria de la capacidad de almacenamiento suficiente para ambos archivos.

Baja física utilizando el mismo archivo de datos

El caso de borrado físico utilizando el mismo archivo de datos se presenta en el Ejemplo 4.2, donde se quita del archivo al empleado sin generar otro archivo. La precondición del problema presentado en el Ejemplo 4.2 es la misma que la presentada para el Ejemplo 4.1.

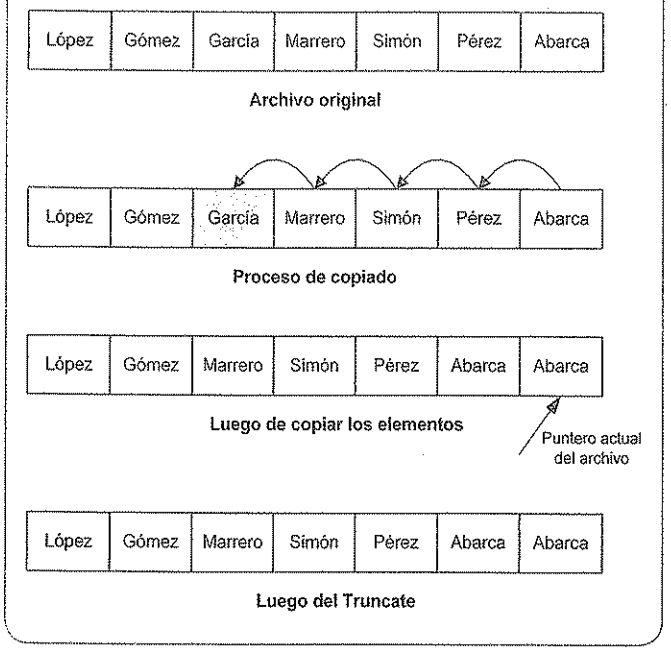
EJEMPLO 4.2

```
program ejemplo_4_2;
  const valoralto="ZZZ";
  type
      empleado = record;
            Nombre          : string[50];
            Direccion       : string[50];
            Documento       : string[12];
            Edad            : integer;
            Observaciones:    string[200];
     end;
    archivoemple = file of Empleado;
var
 reg: empleado;
 archivo: archivoemple;

procedure leer (var archivo: archivoemple; var dato:empleado);
    begin
      if (not eof(archivo))
        then
            read (archivo,dato)
        else
            dato.Nombre := valoralto;
    end;

{programa principal}
begin
  assign (archivo, 'archemple');
  reset(archivo);
  leer(archivo,reg);

{se avanza hasta Carlos Garcia}
while (reg.Nombre<>"Carlos Garcia")
 begin
  leer(archivo,reg);
 end;

{se avanza hasta el siguiente a Carlos Garcia}
leer(archivo,reg);

{se copian los registros restantes}
While (reg.Nombre<>valoralto) do
  begin
     Seek( archivo, filepos(archivo)-2 );
     write(archivo,reg);
     Seek( archivo, filepos(archivo)+1 );
     leer(archivo,reg);
  end;

 truncate(archivo);
end.
```

Para poder quitar el elemento a borrar, el algoritmo requiere localizar, primero, al empleado Carlos García. Luego, copiar sobre este registro el elemento siguiente y así sucesivamente, repitiendo esta operatoria hasta el final del archivo.

La Figura 4.1 presenta en forma gráfica el proceso. Se debe notar que el archivo no fue cerrado utilizando la instrucción close(); en su reemplazo se utilizó truncate(). La diferencia entre ambas radica en que close() coloca la marca de fin luego del último elemento, y de este modo el *n*-ésimo registro quedaría repetido. La instrucción truncate(), en cambio, coloca la marca de fin en el lugar indicado por el puntero del archivo en ese momento, quitándose todo lo que hubiera desde esa posición en adelante.

Figura 4.1



El análisis de *performance*, para este ejemplo, determina que la cantidad de lecturas a realizar es *n*, en tanto que la cantidad de escrituras dependerá del lugar donde se encuentre el elemento a borrar; en el peor de los casos, deberán realizarse nuevamente *n*-1 escrituras, si el elemento a borrar apareciera en la primera posición del archivo. Es de notar que, a diferencia del método anterior, no es necesario contar con mayor capacidad en el disco rígido. Esto se debe a que se utiliza la ubicación original del archivo en memoria secundaria.

Baja lógica

Se realiza una baja lógica sobre un archivo cuando el elemento que se desea quitar es marcado como borrado, pero sigue ocupando el espacio dentro del archivo. La ventaja del borrado lógico tiene que ver con la *performance*, basta con localizar el registro a eliminar y colocar sobre él una marca que indique que se encuentra no disponible. Entonces, la *performance* necesaria para llevar a cabo esta operación es de tantas lecturas como sean requeridas hasta encontrar el elemento a borrar, más una sola escritura que deja la marca de borrado lógico sobre el registro.

La desventaja de este método está relacionada con el espacio en disco. Al no recuperarse el espacio borrado, el tamaño del archivo tiende a crecer continuamente. Como se verá en este capítulo, esto puede ser compensado con técnicas alternativas para el alta de nuevos elementos.

El Ejemplo 4.3 presenta el algoritmo que elimina de manera lógica a un registro del archivo. En este caso, la precondición del problema es similar a los ejemplos anteriores de este capítulo.

EJEMPLO 4.3

```
program ejemplo_4_3;

  const valoralto="ZZZ";

  type
      empleado = record;
            Nombre        : string[50];
            Direccion     : string[50];
            Documento     : string[12];
            Edad          : integer;
            Observaciones: string[200];
      end;
      archivoemple = file of Empleado;
var
 reg: empleado;
 archivo: archivoemple;

procedure leer (var archivo: archivoemple; var dato:empleado);
begin
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.Nombre := valoralto;
end;

{programa principal}
begin
  assign (archivo, 'archemple');
  reset(archivo);
  leer(archivo,reg);
```

continúa >>>

```
  {se avanza hasta Carlos Garcia}
  while (reg.Nombre<>"Carlos Garcia")do
    begin
      leer(archivo,reg);
    end;

  {se genera una marca de borrado }
  reg.nombre := "***"

  {se borra lógicamente a Carlos Garcia}
  Seek( archivo, filepos(archivo)-1 );
  write(archivo,reg);

  close(archivo);
end.
```

El Ejemplo 4.4 muestra la forma en que debería recorrerse el archivo luego de ejecutado el Ejemplo 4.3. Así, ahora se consideran válidos solo aquellos elementos que no se encuentren indicados como borrados.

EJEMPLO 4.4

```
program ejemplo_4_4;
  const valoralto="ZZZ";
  type
       empleado = record;
              Nombre       : string[50];
              Direccion    : string[50];
              Documento    : string[12];
              Edad         : integer;
              Observaciones: string[200];
      end;
      archivoemple = file of Empleado;
var
 reg: empleado;
 archivo: archivoemple;
procedure leer (var archivo: archivoemple; var dato:empleado);
begin
  if (not eof(archivo)) then
    read (archivo,dato)
  else
    dato.Nombre := valoralto;
end;
{programa principal}
begin
  assign (archivo, 'archemple');
  reset(archivo);
  leer(archivo,reg);
  {se avanza hasta Carlos Garcia}
  while (reg.Nombre<>valoralto)do
    begin
      if (reg.Nombre<>'***')then
        write('El nombre de es:'+reg.Nombre)
      leer(archivo,reg);
    end;
  close(archivo);
end.
```


Es posible, en este momento, hacer una comparación de los dos métodos de baja. Las ventajas, en cada caso, tienen que ver con la *performance* del algoritmo y el espacio utilizado en el disco rígido. Mientras que la baja lógica no recupera espacio en memoria secundaria, se comporta de forma mucho más eficiente en el tiempo de respuesta. Además, es posible combinar el proceso de baja lógica con el proceso de ingreso de nueva información al archivo de datos.

Recuperación de espacio

El proceso de baja lógica marca la información de un archivo como borrada. Ahora bien, esa información sigue ocupando espacio en el disco rígido. La pregunta a responder sería: ¿qué hacer con dicha información? Esta pregunta tiene dos respuestas posibles:

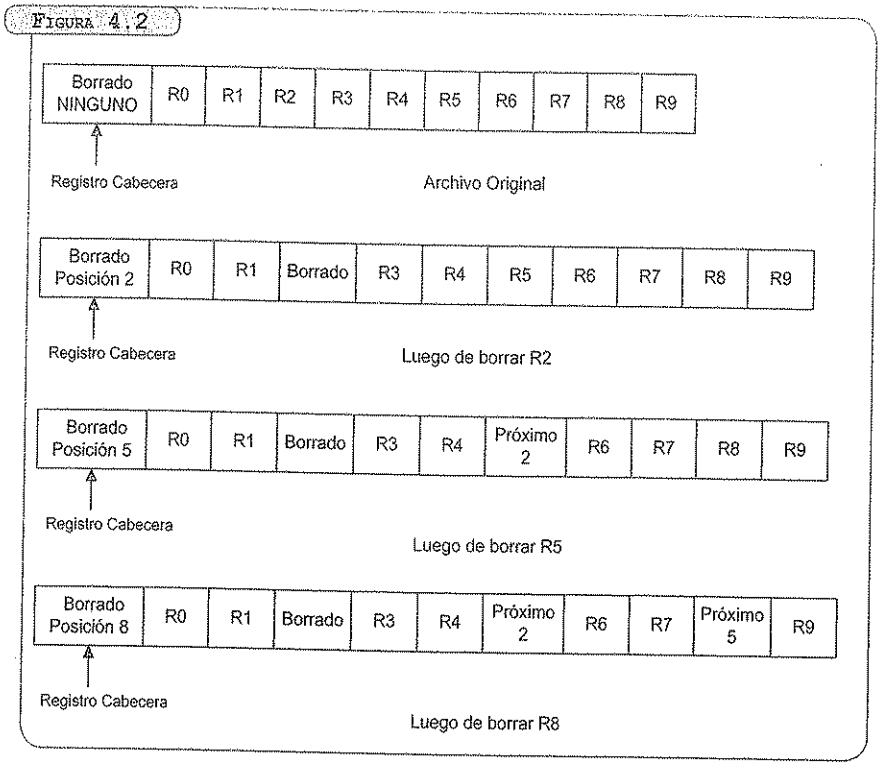
- **Recuperación de espacio:** periódicamente utilizar el proceso de baja física para realizar un proceso de compactación del archivo. El mismo consiste en quitar todos aquellos registros marcados como borrados, utilizando para ello cualquiera de los algoritmos discutidos anteriormente para borrado físico.
- **Reasignación de espacio:** otra alternativa posible consiste en recuperar el espacio, utilizando los lugares indicados como borrados para el ingreso (altas) de nuevos elementos al archivo.

Reasignación de espacio

Esta técnica consiste en reutilizar el espacio indicado como borrado para que nuevos registros se inserten en dichos lugares. Así, el proceso de alta discutido en capítulos anteriores se vería modificado; en lugar de avanzar sobre la última posición del archivo (donde se encuentra la marca de `EOF`), se debe localizar alguna posición marcada como borrada para insertar el nuevo elemento en dicho lugar.

Este proceso puede realizarse buscando los lugares libres desde el comienzo del archivo, pero se debe considerar que de esa manera sería muy lento. La alternativa consiste en recuperar el espacio de forma eficiente. Para ello, y como lo muestra la Figura 4.2, a medida que los datos se borran del archivo, se genera una lista encadenada invertida con las posiciones borradas. En la figura, primero se borra el registro 2, luego, el 5, y por último, el 8. Se puede observar cómo se lleva a cabo el proceso de borrado, indicando el espacio que queda libre. Al comienzo, en el registro cabecera del archivo solamente se indica que no hay registros borrados. A medida que se quitan elementos, este proceso se reitera en la lista de elementos borrados que se va construyendo

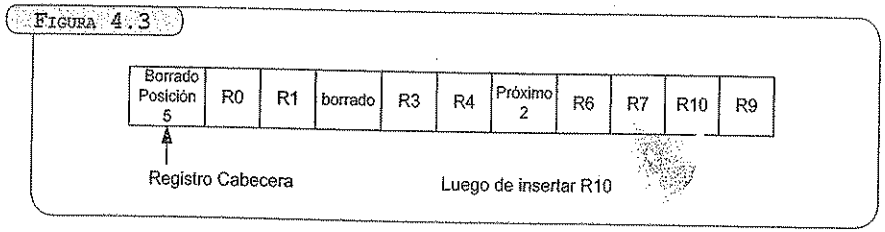
Figura 4.2



Se puede observar que en la posición donde estaba el registro 2 aparece la leyenda Borrado, en tanto que, en las demás posiciones, la leyenda indica Próximo y el número del siguiente registro borrado. Esto significa cuál es la siguiente posición libre, en tanto que para la posición 2, al no haber elemento siguiente, la leyenda Borrado indica fin de la lista.

Ahora, para recuperar el espacio, el proceso de alta debe, primeramente, verificar la lista por espacios libres. En caso de encontrar algún espacio libre, el registro se deberá insertar en esa posición, actualizando el registro cabecera, tal como lo muestra la Figura 4.3.

Figura 4.3



Luego de insertar el registro 10, en la posición donde antes se encontraba el registro 8, el registro cabecera del archivo es actualizado, apuntando ahora a la posición del registro 5.

Si el archivo no dispone de lugares para reutilizar, el registro cabecera no tiene ninguna dirección válida. En ese caso, para ingresar un nuevo elemento se debe acceder a la última posición del archivo, y agregarlo por ende al final.

Campos y registros con longitud variable

La información en un archivo siempre es homogénea. Esto es, todos los elementos almacenados en él son del mismo tipo. De esta forma, cada uno de los datos es del mismo tamaño, generando lo que se denomina archivos con registros de longitud fija.

La longitud de cada registro está determinada por la información que se guarda.

- Si se define un archivo que contiene números enteros, cada elemento ocupa 2 bytes.
- Si se define un archivo que contiene caracteres, cada elemento ocupa 1 byte.
- Si se define un archivo que contiene String[20], cada elemento ocupa 20 bytes.
- Si se define un archivo que contiene registros con Nombre y apellido (string[40]), DNI (string[8]), Edad (integer), cada elemento ocupa 50 bytes (la suma de la longitud de cada uno de los campos).

Administrar archivos con registros de longitud fija tiene algunas importantes ventajas: el proceso de entrada y salida de información, desde y hacia los *buffers*, es responsabilidad del sistema operativo; los procesos de alta, baja y modificación de datos se corresponden con todo lo visto hasta el momento.

No obstante, hay determinados problemas donde no es posible, no es deseable trabajar con registros de longitud fija. Supóngase el siguiente ejemplo. Se define un registro empleado con la siguiente estructura:

```
Type Empleado = Record;
       Nombre       : string[50];
       Direccion    : string[50];
       Documento    : string[12];
       Edad         : integer;
       Observaciones: string[200];
     End;
```

El registro anterior ocupa 314 bytes. Cada vez que se ingresan los datos de un empleado, en promedio se ocupan 30 lugares para el nombre, otro tanto para la dirección y, en algunos casos, 50 bytes para observaciones. En promedio, cada registro utiliza 124 de los 314 bytes disponibles. Esto significa desperdicio de espacio y mayor tiempo de procesamiento. Se transfieren 314 bytes de los cuales solo 124 representan información útil; el resto son espacios de relleno.

Para evitar estas situaciones, es de interés contar con alguna organización de archivos que solo utilice el espacio necesario para almacenar la información. Este tipo de soluciones se representan con archivos donde los registros utilicen longitud variable. En estos casos, como el nombre lo indica, la cantidad de espacio utilizada por cada elemento del archivo no está determinada *a priori*.

Para poder arribar a soluciones de este tipo, es necesario que los archivos de datos tengan una estructura diferente de lo visto hasta el momento. La declaración del archivo vista hasta el momento determina el tipo de dato que contendrá y, por consiguiente, el tamaño de cada elemento. Es necesario entonces definir el archivo de otra forma. Para ello existen varias soluciones, pero por razones prácticas en este libro se tratará al archivo de datos como una secuencia de caracteres, donde EOF seguirá representando la marca de fin de archivo.

Entonces, cada elemento de dato debe descomponerse en cada uno de sus elementos constitutivos y así, elemento a elemento, guardarse en el archivo. En el caso de necesitar transferir un string, debe hacerse carácter a carácter; en caso de tratarse de un dato numérico, cifra a cifra.

El siguiente problema consiste en definir dónde empieza y dónde termina cada registro o cada campo del archivo. Es necesario colocar marcas que delimiten cada elemento. Estas marcas se denominan marcas de fin de campo o marcas de fin de registro, en cada caso, y pueden ser cualquier carácter que luego no deberá poder ser parte de un elemento de dato del archivo.

El Ejemplo 4.5 describe un algoritmo que crea un archivo de empleados, con nombre y apellido, dirección y documento, donde cada registro es tratado como de longitud variable.

EJEMPLO 4.5

```
program ejemplo_4_5;
Var
 empleados: file; {archivo sin tipo}
 nombre,apellido,direccion,documento: string;

Begin
  Assign(empleados,'empleados.txt');
  Rewrite(empleados,1);
  writeln('Ingrese el Apellido');
  readln(apellido);
  writeln('Ingrese el Nombre');
  readln(nombre);
  writeln('Ingrese direccion');
  readln(direccion);
  writeln('Ingrese el documento');
  readln(documento);

  while apellido<>'zzz' do
    Begin
      BlockWrite(empleados,apellido,length(apellido)+1);
      BlockWrite(empleados,'#',1);
      BlockWrite(empleados,nombre,length(nombre)+1);
      BlockWrite(empleados,'#',1);
      BlockWrite(empleados,direccion,length(direccion)+1);
      BlockWrite(empleados,'#',1);
      BlockWrite(empleados,documento,length(documento)+1);
      BlockWrite(empleados,'@',1);
      writeln('Ingrese el Apellido');
      readln(apellido);
      writeln('Ingrese el Nombre');
      readln(nombre);
      writeln('Ingrese direccion');
      readln(direccion);
      writeln('Ingrese el documento');
      readln(documento);
    end;
  close(empleados);
end.
```

Se deben notar las características diferentes que tiene este tipo de problema:

1. El archivo se encuentra definido como `file` y permite realizar la transferencia de la información carácter a carácter.
2. Cuando se termina de insertar un campo, se utiliza como marca de fin de campo el carácter #.
3. Cuando se termina de insertar un registro, se utiliza como marca de fin de campo el carácter @.

Al utilizar registros de longitud variable, una de las principales diferencias con el uso de registros de longitud fija radica en las operaciones de lectura y escritura. Mientras que con registros de longitud fija basta con definir el tipo de datos del archivo, para que tanto la operación `read` como `write` realicen la transferencia estructurada de información, con registros de longitud variable, la operatoria debe resolverse leyendo o escribiendo de a uno los caracteres que componen un registro. Queda como ejercicio intelectual desarrollar el Ejemplo 4.5, suponiendo que el registro es definido con longitud fija.

La primera conclusión que se puede obtener a partir del uso de registros de longitud variable es que la utilización de espacio en disco es optimizada, respecto del uso, con registros de longitud fija. Sin embargo, esto conlleva un algoritmo donde el programador debe resolver en forma mucho más minuciosa las operaciones de agregar y quitar elementos.

El Ejemplo 4.6 muestra el algoritmo que permite recorrer el archivo anteriormente generado y presenta los datos en pantalla. Nuevamente, queda como tarea plantear la solución para archivos con registros de longitud fija.

EJEMPLO 4.6

```
program ejemplo_4_6;
Var
  empleados: file; {archivo sin tipo}
  campo, buffer :string;
Begin
  Assign(empleados,'empleados.txt');
  reset(empleados,1);
  while not eof(empleados) do
    Begin
      BlockRead(empleados,buffer,1);
      while (buffer<>'@') and not eof(empleados)do
        begin
          while ((buffer<>'@') and
                 (buffer<>'#') and
```

continúa >>>


```
                 not eof(empleados))do
            begin
              campo := campo + buffer ;
              BlockRead(empleados,buffer,1);
            end;
          writeln(campo);
        end;

      if not eof(empleados) then
        BlockRead(empleados,nombre,1);
    end;
  close(empleados);
end.
```

Alternativas para registros de longitud variable

Cuando se plantea la utilización de espacio con longitud variable sobre técnicas de espacio fijo, existen algunas variantes que se pueden analizar.

En el apartado anterior, se trabajó con registros de longitud variable y campos de longitud variable. En ese caso, la solución planteada utilizó delimitadores de campo para describir el final de cada uno de ellos. Así, si un registro dispone de 4 campos, luego de obtener 4 marcas de fin se asume la finalización de un registro y el comienzo de uno nuevo. Esto implica que no es necesario indicar de forma explícita el fin del registro.

Existen otras variantes; estas tienen que ver con utilizar indicadores de longitud de campo y/o registro. De esta manera, antes de almacenar un registro, se indica su longitud; luego, los siguientes bytes corresponden a elementos de datos de dicho registro.

Eliminación con registros de longitud variable

El proceso de baja sobre un archivo con registros de longitud variable es, *a priori*, similar a lo discutido anteriormente. Un elemento puede ser eliminado de manera lógica o física. En este último caso, el modo de recuperar espacio es similar a lo planteado en los Ejemplos 4.3 o 4.4.

El proceso de baja lógica no tiene diferencias sustanciales con respecto a lo discutido anteriormente. Sin embargo, cuando se desea recuperar el espacio borrado lógicamente con nuevos elementos, deben tenerse en cuenta nuevas consideraciones. Estas tienen que ver con el espacio disponible. Mientras que con registros de longitud fija los elementos a eliminar e insertar son del mismo tamaño, utilizando

registros de longitud variable esta precondición no está presente. Para insertar un elemento no basta con disponer de lugar; es necesario, además, que el lugar sea del tamaño suficiente.

La filosofía para administración del lugar disponible se plantea, *a priori*, similar al caso discutido con registros de longitud fija. Para ello se genera una lista invertida donde a partir de un registro cabecera se dispone de las direcciones libres dentro del archivo. Es necesario ahora indicar, además, la cantidad de bytes disponibles en cada caso para su reutilización.

El proceso de inserción debe localizar el lugar dentro del archivo más adecuado al nuevo elemento. Existen tres formas genéricas para la selección de este espacio:

- **Primer ajuste:** consiste en seleccionar el primer espacio disponible donde quepa el registro a insertar.
- **Mejor ajuste:** consiste en seleccionar el espacio más adecuado para el registro. Se considera el espacio más adecuado como aquel de menor tamaño donde quepa el registro.
- **Peor ajuste:** consiste en seleccionar el espacio de mayor tamaño, asignando para el registro solo los bytes necesarios.

Fragmentación

Para detallar el proceso de selección de espacio, es necesario avanzar en conceptos de fragmentación. Existen dos tipos de fragmentación: interna y externa.

Se denomina **fragmentación interna** a aquella que se produce cuando a un elemento de dato se le asigna mayor espacio del necesario.

Los registros de longitud fija tienden a generar fragmentación interna, se asigna tanto espacio como lo necesario de acuerdo con la definición del tipo de dato. Pero este espacio no siempre se condice con lo que realmente utiliza el registro. Por ejemplo, si un campo del registro es `Nombre` definido como un string[50], y el nombre en cuestión es "Juan Perez", solo se ocupan realmente 10 de los 50 lugares asignados.

Se denomina **fragmentación externa** al espacio disponible entre dos registros, pero que es demasiado pequeño para poder ser reutilizado.

Las dos fragmentaciones tienen que ver con la utilización del espacio. En el primer caso, fragmentación interna, se reserva lugar que no se utiliza; en tanto, la fragmentación externa se genera por dejar espacios tan pequeños que no pueden ser utilizados.

Fragmentación y recuperación de espacio

Como se mencionó anteriormente, el procedimiento de recuperación de espacio generado por bajas, utilizando registro de longitud variable, presenta tres alternativas.

Cada una de estas alternativas selecciona el espacio considerado más conveniente. Las técnicas de primer y mejor ajuste suelen implementar una variante que genera fragmentación interna. Así, una vez seleccionado el lugar libre, el espacio asignado corresponde a la totalidad de lo disponible. De esa forma, al nuevo registro se le puede asignar más del espacio necesario.

Por el contrario, la técnica de peor ajuste solo asigna el espacio necesario. Se recuerda que peor ajuste consiste en buscar sobre la lista de registros borrados el espacio disponible de mayor tamaño. Una vez localizado, se asignan solo los bytes necesarios y el resto del espacio libre sigue figurando en la lista de disponibles. De esta forma, es posible que la técnica de peor ajuste genere fragmentación externa dentro del archivo. Es deseable, en esos casos, disponer de un algoritmo que se ejecute periódicamente para la recuperación de estos espacios no asignados. Este tipo de algoritmo se conoce como *garbage collector*.

Conclusiones

Las tres técnicas definidas presentan ventajas y desventajas. Desde un punto de vista de *performance* para su implantación, el algoritmo que utilice primer ajuste tiende a ser el más rápido. Solo debe recorrerse la lista de registros borrados hasta encontrar el primer lugar donde quepa el nuevo elemento de datos. Dicho lugar se quita de la lista, asignando todo el espacio disponible al nuevo elemento.

Las alternativas de mejor y peor ajuste, en comparación con primer ajuste, son más ineficientes. En ambos casos, es necesario recorrer toda la lista para encontrar el lugar más adecuado según el caso. Una vez localizado dicho lugar, mejor ajuste se resuelve con mayor rapidez, se asigna dicho lugar completo y se quita de la lista. En tanto, peor ajuste debe asignar solo el espacio requerido y luego dejar en la lista el espacio restante con su respectiva capacidad.

Desde un punto de vista de la fragmentación generada, solo la técnica de peor ajuste mantiene la filosofía de trabajo de longitud variable, donde cada registro ocupa solamente el espacio que necesita.

Modificación de datos con registros de longitud variable

Hasta el momento, se discutieron los procesos de ingreso (alta) y borrado de datos sobre un archivo que contenga registros de longitud variable. Además, en el apartado anterior se presentaron alternativas de reutilización de espacio.

También es importante discutir el proceso de modificación de información de un archivo. Cuando se trabaja con archivos que soportan registros de longitud fija, una modificación consiste en sobreescribir un registro con el nuevo dato. Esto fue discutido en detalle en el Capítulo 3.

Sin embargo, cuando los archivos soportan registros de longitud variable, surge un nuevo problema. Modificar un registro existente puede significar que el nuevo registro requiera el mismo espacio en disco, que ocupe menos espacio o que requiera uno de mayor tamaño. Como es natural, el problema no se genera cuando ambos registros requieren el mismo espacio. Se puede suponer que si el nuevo elemento ocupa menos espacio que el anterior, no se genera una situación problemática dado que el espacio disponible es suficiente, aunque en ese caso se generaría fragmentación interna.

El problema surge cuando el nuevo registro ocupa mayor espacio que el anterior. En este caso, no es posible utilizar el mismo espacio físico, y el registro necesita ser reubicado.

En general, para evitar todo este análisis y para facilitar el algoritmo de modificación sobre archivos con registros de longitud variable, se estila dividir el proceso de modificación en dos etapas: en la primera se da de baja al elemento de dato viejo, mientras que en la segunda etapa el nuevo registro es insertado de acuerdo con la política de recuperación de espacio determinada.

Cuestionario del capítulo

1. ¿Por qué el proceso de baja de información de un archivo fue cambiando a lo largo del tiempo?
2. ¿Cuáles son las diferencias sustanciales entre el proceso de baja lógica y baja física?
3. ¿Cuáles son las ventajas que plantea la reutilización de espacio ligada con la baja de información en un archivo?
4. ¿Qué ventajas presenta la utilización de registros de longitud variable?
5. ¿Por qué el proceso de recuperación de espacio, utilizado en archivos con registros de longitud variable, difiere del mismo proceso con archivos que contengan registros de longitud fija?
6. ¿Cuáles son las principales características que llevarían a utilizar primer ajuste en vez de peor ajuste, como método de recuperación de espacios?
7. ¿Qué motiva a tener un proceso más complejo de modificación de datos, cuando se trabaja con registros de longitud variable?

Ejercitación

1. Utilizando el archivo generado en el Ejercicio 1 del Capítulo 3, genere un algoritmo que permita borrar los promotores que tengan ventas por debajo de $15000.
   a) Borrando los datos de manera física.
   b) Borrando los datos de manera lógica.
2. Realice los cambios necesarios a los algoritmos del Ejercicio 1, para poder determinar el tiempo de ejecución necesario en cada uno de ellos. Para tal fin, el archivo de datos inicial debe ser el mismo y contar con un mínimo de 50.000 registros.
3. Defina un programa que genere un archivo que contenga los datos personales de los docentes de la facultad, utilizando para ello registros de longitud fija.
4. Implante un algoritmo que, a partir del archivo de datos generado en el ejercicio anterior, elimine de forma lógica los docentes de más de 65 años.
5. Genere todas las modificaciones necesarias sobre el Ejercicio 4 de manera de poder implementar el proceso de altas reutilizando espacio. Para ello, deberá utilizar las políticas de:
   a) Primer ajuste b) Mejor ajuste c) Peor ajuste

CAPÍTULO 5

Búsqueda de información. Manejo de índices

Objetivo

En los capítulos previos, se definieron las operaciones básicas sobre archivos. Sin embargo, dichas operaciones se analizaron desde el punto de vista algorítmico y no con una visión de BD (motivación fundamental de este libro). Con esta motivación, el propósito de este capítulo es trabajar sobre la búsqueda de información en archivos, con especial atención en la *performance*, dividiendo el problema en dos grandes grupos: el primero de ellos realiza la búsqueda sobre archivos directamente, mientras que en el segundo caso se plantea el uso de estructuras de datos auxiliares que sirven de soporte a la mejora en el acceso.

Proceso de búsqueda

Cuando se realiza la búsqueda de un dato, se debe considerar la cantidad de accesos a disco en pos de encontrar esa información, y en cada acceso, la verificación de si el dato obtenido es el buscado (comparación). Es así que surgen dos parámetros a analizar: cantidad de accesos y cantidad de comparaciones. El primero de ellos es una operación sobre memoria secundaria, lo que implica un costo relativamente alto; mientras que el segundo es sobre memoria principal, por lo que el costo es relativamente bajo. A modo de ejemplo, si un acceso a memoria principal representa 5 segundos, un acceso a memoria secundaria representaría 29 días. Por lo tanto, se tomará en cuenta solo el costo de acceso a memoria secundaria para los análisis de *performance* tanto de este capítulo como de los subsiguientes.

El proceso de búsqueda implica un análisis de situaciones en función del tipo de archivo sobre el que se quiere buscar información.

El caso más simple consiste en disponer de un archivo serie, es decir, sin ningún orden preestablecido más que el físico, donde, para acceder a un registro determinado, se deben visitar todos los registros previos en el orden en que estos fueron almacenados. Aquí, la búsqueda de un dato específico se detiene en el registro que contiene ese dato (previo acceso a todos los registros anteriores), o en el final del archivo si el resultado no es exitoso (el dato no se encuentra). Por lo tanto, el mejor caso es ubicar el dato deseado en el primer registro (1 lectura), y el peor caso, en el último registro (*N* lecturas, siendo *N* la cantidad de registros). El caso promedio consiste entonces en realizar *N*/2 lecturas. Es decir, la *performance* depende de la cantidad de registros del archivo, siendo entonces de orden *N*.

Mejor caso → 1 acceso
Peor caso → *N* accesos

Promedio → *N*/2 accesos

Si el archivo está físicamente ordenado y el argumento de búsqueda coincide con el criterio de ordenación, la variación en relación con el caso anterior se produce para el caso de que el dato buscado no se encuentre en dicho archivo. En ese caso, el proceso de búsqueda se detiene en el registro cuyo dato es "mayor" buscado. De ese modo, en ese caso no existe la necesidad de recorrer todo el archivo. No obstante, la *performance* sigue siendo de orden *N*, y a la estrategia de búsqueda utilizada en ambos casos se la denomina secuencial.

Si se dispone de un archivo con registros de longitud fija y además físicamente ordenado, es posible mejorar esta *performance* de acceso si la búsqueda se realiza con el mismo argumento que el utilizado para ordenar este archivo. En este caso, la estrategia se ve alterada. La primera comparación del dato que se pretende localizar es contra el registro medio del archivo, es decir, el que tiene *NRR=N*/2. Si el registro no contiene ese dato, se descarta la mitad mayor o menor, según corresponda, reduciendo el espacio de búsqueda a los *N*/2 registros restantes (mitad restante). Nuevamente, se realiza la comparación pero con el registro medio de la mitad restante, repitiendo así este proceso hasta reducir el espacio de búsqueda a un registro.

En el ejemplo siguiente, se busca el elemento 15; como el archivo contiene 20 registros, el primer Número Relativo de Registro (NRR) accedido es el registro del décimo lugar.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Como en esa posición está el elemento 25, los registros posteriores al 10 se descartan.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Se calcula el nuevo punto medio, que será la posición 4 del archivo.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Como en esta posición tampoco está el elemento buscado, se descartan los menores al 5.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

El proceso calcula nuevamente el punto medio y localiza el elemento buscado.

1	3	4	5	8	11	15	18	21	25	29	30	40	43	45	56	67	87	88	90

Este criterio de búsqueda, donde la mitad de los registros restantes se descartan en cada comparación, se denomina búsqueda binaria. El Ejemplo 5.1 muestra el seudocódigo de dicha búsqueda.

EJEMPLO 5.1

```
Function Busqueda_Binaria (archivo, clave, long_archivo)

Variables
menor, mayor, clave_encontrada, registro: integer

registro   := -1
menor      := 1
mayor      := long_archivo

mientras ( menor <= mayor )
  medio :={(mayor+menor )/2}
  buscar reg con NRR = medio
  clave_encontrada = clave canónica correspondiente al
  reg. leído
  si clave < clave_encontrada
    entonces
      mayor := medio+1
    sino
      si clave > clave_encontrada
         entonces
              menor := medio + 1
      sino
              registro:= NRR

Busqueda_Binaria:= registro

fin
```

En general, una búsqueda binaria en un archivo con N registros se realiza en a lo sumo $log_2(N)+1$ comparaciones y en promedio $(log_2(N)+1)/2$ comparaciones. Por lo tanto, se concluye que la búsqueda binaria es de orden $log_2(N)$, logrando mejorar sustancialmente el caso anterior. No obstante, se debe considerar el costo adicional de mantener el archivo ordenado para posibilitar este criterio de búsqueda, y el número de accesos disminuye pero aún dista bastante de recuperar la información en un acceso a disco.

La búsqueda de un registro particular en un archivo que contiene registros de longitud fija puede optimizarse, al extremo de lograr ubicar dicho registro con solo un acceso. Esto sucede porque dicho archivo tiene los registros numerados de 0 a N, es decir, cada registro tiene un número relativo dentro del archivo. Dicho número se denomina NRR y permite calcular la distancia en bytes desde el principio del archivo hasta cualquier registro.

Así, para lograr ubicar un registro particular, basta con disponer del NRR del registro y acceder directamente al registro deseado.

Esta situación, si bien posibilita acceso directo, no deja de ser especial, ya que es muy poco probable conocer el NRR del registro que contiene el dato a buscar.

Retomando la búsqueda (secuencial o binaria) en archivos físicamente ordenados, como es necesario garantizar el orden del archivo para poder llevarla a cabo, se definen a continuación distintas alternativas para ordenar físicamente un archivo.

Ordenamiento de archivos

Dada la conveniencia de que un archivo se encuentre ordenado, para lograr mejor desempeño o *performance* en la búsqueda de datos, es necesario discutir alternativas de ordenación física para los archivos de datos.

Si se aplicara cualquier algoritmo de ordenación de vectores, por ejemplo, para realizar la ordenación de un archivo, habría que tener en cuenta que se debe leer y escribir varias veces cada elemento, con el consiguiente desplazamiento desde la posición inicial hasta la posición final en que cada uno quede ubicado. Estas operaciones realizadas en memoria secundaria serían excesivamente lentas.

Supóngase que se dispone de un archivo con 1.000 elementos y que el método de ordenación utilizado es el de N pasadas. Este método recorre el archivo buscando el elemento menor y lo posiciona en el

primer lugar, luego repite la misma operación buscando el segundo menor y así sucesivamente hasta ordenar todo el archivo. En este caso, serán necesarias 1.000 lecturas para determinar el menor elemento, y 2 escrituras que intercambien el primer elemento menor con el elemento que está en la posición 1 del archivo (1.002 accesos). El proceso se repite, ahora con 999 lecturas, para el segundo elemento con sus 2 escrituras (1.001 accesos). Esto significa que la cantidad de operaciones requeridas será:

$$1.002 + 1.001 + 1.000 + (\ldots) + 4 \text{ operaciones}$$

Supóngase ahora que el archivo contiene 1.000.000 de registros. El número final de accesos es excesivamente elevado.

Si el archivo a ordenar puede almacenarse en forma completa en memoria RAM, una alternativa muy atrayente es trasladar el archivo completo desde memoria secundaria hasta memoria principal, y luego ordenarlo allí. Si bien esto implica leer los datos de memoria secundaria, su acceso es secuencial sin requerir mayores desplazamientos, por lo que su costo no es excesivo. Luego, la ordenación efectuada en memoria principal será realizada con alta *performance*. La operatoria finaliza escribiendo nuevamente el archivo ordenado en memoria secundaria, otra vez en forma consecutiva, con pocos desplazamientos de la cabeza lectograbadora del disco.

Esta posibilidad constituye la mejor alternativa en cuanto a *performance*, para ordenar físicamente un archivo, pero solamente puede ser utilizada para archivos pequeños.

Si el archivo no cabe en memoria RAM, una segunda alternativa constituye transferir a memoria principal de cada registro del archivo solo la clave por la que se desea ordenar, junto con los NRRs, a los registros correspondientes en memoria secundaria. De ese modo, al transferir solo esos datos, es posible almacenar mayor cantidad de registros en RAM y, por lo tanto, ordenar un archivo de mayor cantidad de datos. Además, es una muy buena opción, ¿para qué transferir cada registro en forma completa si finalmente se considerará la clave como argumento para hacer la ordenación?

El algoritmo ordena en memoria principal solo las claves. Posteriormente, se debe leer nuevamente cada registro del archivo (de acuerdo con el orden establecido por las claves) y escribirlo sobre el archivo ordenado. De este modo, el proceso requiere leer el archivo en forma completa dos veces y reescribirse ordenado una vez. Esto implica muchos desplazamientos en memoria secundaria. Por lo tanto, es alto el costo a pagar para poder utilizar la memoria principal como medio de ordenación, además de la restricción de que todas las claves más los NRRs deben caber en dicha memoria, por lo que si el archivo es

realmente grande, esta alternativa se ve imposibilitada. No obstante, la idea de reorganizar solo las claves es interesante y será tratada nuevamente en este capítulo.

La tercera alternativa surge cuando el archivo es realmente grande, de modo tal que las claves no se pueden ordenar en memoria principal porque no caben.

Es así que esta opción implica plantear otra estrategia para ordenar el archivo, obviamente descartando la posibilidad de hacerlo directamente sobre memoria secundaria, por los costos ya discutidos. Esta estrategia consiste en los siguientes pasos:

1. Dividir el archivo en particiones de igual tamaño, de modo tal que cada partición quepa en memoria principal.
2. Transferir las particiones (de a una) a memoria principal. Esto implica realizar lecturas secuenciales sobre memoria secundaria, pero sin ocasionar mayores desplazamientos.
3. Ordenar cada partición en memoria principal y reescribirlas ordenadas en memoria secundaria. También en este caso la escritura es secuencial (estos tres pasos anteriores se denominan *sort* interno).
4. Realizar el *merge* o fusión de las particiones, generando un nuevo archivo ordenado. Esto implica la lectura secuencial de cada partición nuevamente y la reescritura secuencial del archivo completo. Esta operatoria ya fue anteriormente descripta en el Capítulo 3.

A modo de ejemplo, supóngase que se dispone de un archivo de 800.000 registros y 1 Mb de memoria principal (si bien es poca la capacidad de memoria definida, se deben tomar los datos a modo de ejemplo solamente). Cada registro ocupa 100 bytes, por lo que la longitud total del archivo es 80.000.000 bytes, es decir, aproximadamente 80 Mb (el archivo no se puede transferir a memoria principal).

La clave de cada registro ocupa 10 bytes, por lo que para albergar todas las claves se necesitan 8 Mb (tampoco se pueden transferir a memoria principal). Es así que la única opción posible es generar particiones del tamaño de memoria principal disponible, 1 Mb / 100 bytes por registro = 10.000 registros forman una partición, y al disponer de 800.000 registros se originarán 80 particiones.

Para estos datos, se analiza el desplazamiento necesario para realizar el *merge* o fusión, considerando solamente las operaciones de lectura (faltaría realizar los cálculos para la escritura).

Dado que cada una de las 80 particiones es del tamaño de la memoria principal disponible (puesto que así fue definido en su conformación), al realizar el *merge* simultáneo de las 80 particiones, solo es posible asignar 1/80 parte de la memoria a cada partición. En consecuencia,

es preciso desplazarse 80 veces por cada una de ellas para leerla en forma completa, y como hay 80 particiones para completar la operación de *merge*, se tienen que realizar:

80 particiones * 80 desplazamientos por porción =
6.400 desplazamientos

Ya se ha mencionado el alto costo de realizar desplazamientos en memoria secundaria, por lo que 6.400 desplazamientos resulta una cantidad que necesita ser reducida mediante algún método alternativo.

Generalizando, para realizar el *merge* de K formas de K porciones, donde cada partición es del tamaño disponible de memoria principal, se requieren k desplazamientos para leer todos los registros en cada partición individual. Además, dado que k porciones, la operación de *merge* requiere k^2 desplazamientos. Por lo tanto, la ordenación evaluada en términos de desplazamientos tiene una *performance* del orden k^2, y como k es directamente proporcional a la cantidad total (N) de registros del archivo, se puede concluir que la ordenación es una operación de $O(N^2)$ evaluada en términos de desplazamiento.

A continuación, se analizan situaciones alternativas al *sort* interno, que posibiliten incrementar el tamaño de cada partición, sin requerir memoria principal adicional. La *performance* del *merge* está ligada de manera inversa a la cantidad de particiones generadas. A mayor número de particiones, menor la *performance* final. Por consiguiente, si se logra reducir el número de particiones, la *performance* debe mejorar.

Selección por reemplazo

Si pudiera incrementarse de algún modo el tamaño de las particiones, decrecería la cantidad de trabajo requerida durante el *merge* para el proceso de ordenar un archivo. Particiones más grandes implican menor cantidad total de particiones, y al disponer de menos particiones, se necesitan menos desplazamientos en memoria secundaria para realizar las lecturas necesarias. La pregunta, en este punto, sería: ¿cómo crear particiones que sean por ejemplo dos veces más grandes que la cantidad de registros que se pueden almacenar en memoria principal? La respuesta implica suplir esta necesidad con una nueva estrategia que se utiliza con un algoritmo conocido como **selección por reemplazo**.

Este método se basa en el siguiente concepto: seleccionar siempre de memoria principal la clave menor, enviarla a memoria secundaria y reemplazarla por una nueva clave que está esperando ingresar a memoria principal.

Sintetizando, los pasos a cumplir serían:

1. Leer desde memoria secundaria tantos registros como quepan en memoria principal.
2. Iniciar una nueva partición.
3. Seleccionar, de los registros disponibles en memoria principal, el registro cuya clave es menor.
4. Transferir el registro elegido a una partición en memoria secundaria.
5. Reemplazar el registro elegido por otro leído desde memoria secundaria. Si la clave de este registro es menor que la clave del registro recientemente transferido a memoria secundaria, este nuevo registro se lo guarda como no disponible.
6. Repetir desde el Paso 3 hasta que todos los registros en memoria principal estén no disponibles.
7. Iniciar una nueva partición activando todos los registros no disponibles en memoria principal y repetir desde el Paso 3 hasta agotar los elementos del archivo a ordenar.

El Ejemplo 5.2 ilustra cómo funciona este método. Durante la primera partición, cuando las claves son muy pequeñas para incluirse allí, se las marca entre paréntesis, es decir, se las "duerme", indicando que formarán parte de la siguiente partición. Cabe destacar que el tamaño de memoria principal admite 3 claves, mientras que el tamaño de la primera partición quedó de 6 claves y el de la segunda partición, de 7 claves.

EJEMPLO 5.2

Claves en memoria secundaria:

34, 19, 25, 59, 15, 18, 8, 22, 68, 13, 6, 48, 17

↑

Principio de la cadena de entrada

Resto de la entrada	Mem. principal	Partición generada
34, 19, 25, 59, 15, 18, 8, 22, 68, 13	6 48 17	-
34, 19, 25, 59, 15, 18, 8, 22, 68	13 48 17	6
34, 19, 25, 59, 15, 18, 8, 22	68 48 17	13, 6
34, 19, 25, 59, 15, 18, 8	68 48 22	17, 13, 6
34, 19, 25, 59, 15, 18	68 48 (8)	22, 17, 13, 6
34, 19, 25, 59, 15	68 (18) (8)	48 ,22, 17, 13, 6
34, 19, 25, 59	(15) (18) (8)	68, 48, 22, 17, 13, 6

continúa >>>

Primera partición completa; se inicia la construcción de la siguiente:

Resto de la entrada	Mem. principal	Partición generada
34, 19, 25, 59	(15) (18) (8)	-
34, 19, 25	15 18 59	8
34, 19	25 18 59	15, 8
34	25 19 59	18, 15, 8
	25 34 59	19, 18, 15, 8
	- 34 59	25, 19, 18, 15, 8
	- - 59	34, 25, 19, 18, 15, 8
	- - -	59, 34, 25, 19, 18, 15, 8

Generalizando, se puede establecer que, en promedio, este método aumenta el tamaño de las particiones al doble de la cantidad de registros que se podrían almacenar en memoria principal (2P). Esto ha sido demostrado ya en la década de 1970. Además, si los datos de entrada a memoria principal están (aunque sea) parcialmente ordenados, el tamaño promedio de las particiones generadas es superior al doble (2P).

Selección natural

El método de selección natural en relación con selección por reemplazo es una variante que reserva y utiliza memoria secundaria en la cual se insertan los registros no disponibles. De este modo, la generación de una partición finaliza cuando este nuevo espacio reservado está en *overflow* (completo).

Comparando los tres métodos analizados:

- **Ventajas**:
  - ***Sort* interno:** produce particiones de igual tamaño. Algorítmica simple. No es un detalle menor que las particiones tengan igual tamaño; cualquier variante del método de *merge* es más eficiente si todos los archivos que unifica son de igual tamaño.
  - **Selección natural y selección por reemplazo:** producen particiones con tamaño promedio igual o mayor al doble de la cantidad de registro, que caben en memoria principal.
- **Desventajas**:
  - ***Sort* interno:** es el más costoso en términos de *performance*.
  - **Selección natural y selección por reemplazo:** tienden a generar muchos registros no disponibles. Las particiones no quedan, necesariamente, de igual tamaño.

Merge en más de un paso

En pos de buscar mejoras aun mayores que las alternativas ya vistas, se plantea la posibilidad de realizar el *merge* o fusión en más de un paso. Esto requiere leer 2 veces cada registro; no obstante, se analiza a modo de ejemplo la situación planteada anteriormente (archivo de 800.000 registros y 1 Mb de memoria principal, donde cada registro ocupa 100 bytes y la clave, 10 bytes). En este caso, en vez de realizar el *merge* entre 80 particiones, se realizará el *merge* de 10 conjuntos de 8 particiones cada una. Se generan, así, 10 archivos intermedios, los cuales deberán sufrir otro *merge* para obtener el archivo original ordenado.

El primer paso consiste en tomar 8 particiones, asignando un *buffer* a cada una. Estos 8 *buffers* de entrada pueden almacenar 1/8 de cada partición, requiriendo 8 desplazamientos cada uno para recuperar una partición, y como se dispone de 8 particiones, serán necesarios 64 desplazamientos en memoria secundaria para generar el archivo intermedio (8 desplazamientos * 8 particiones).

Este proceso debe repetirse 9 veces más para generar los 10 archivos intermedios; entonces, se deberán realizar 10 * 64 = 640 desplazamientos.

Además, cada una de las 10 particiones es 8 veces más grande; esto implica 80 desplazamientos más por partición, y como son 10 particiones, 10 * 80 = 800 desplazamientos más. En total, se requieren 640 + 800 = 1.440 desplazamientos.

Si comparamos los 6.400 desplazamientos iniciales que eran requeridos en el *merge* en un paso, la mejora es evidente.

Si bien existen otros métodos de *merge* (balanceado de *N* caminos, óptimo, *multiface*, entre otros), no serán analizados en este libro.

Indización

Se ha planteado que el propósito de ordenar un archivo radica en tratar de minimizar los accesos a memoria secundaria durante la búsqueda de información. Sin embargo, la *performance* es del orden $log_2(N)$, y puede ser mejorada.

Supóngase el problema de buscar un tema en un libro; independientemente de si este estuviera ordenado por temas, la acción natural sería buscar el material deseado en el índice temático del libro, y luego, acceder directamente a la página que se incluya en dicho índice. Es de notar que en este caso se busca la información en una fuente de datos adicional (el índice), que es de tamaño considerablemente menor, para luego acceder directamente a dicha información.

Analizando este ejemplo, puede intuirse que, con el uso de una fuente de información organizada adicional, se puede lograr mejorar la *performance* de acceso a la información deseada. Esta idea conduce a pensar en realizar algo similar con el acceso a la información almacenada en archivos,,es decir, utilizar una estructura adicional que permita mejorar la *performance* de acceso al archivo.

Un índice es una estructura de datos adicional que permite agilizar el acceso a la información almacenada en un archivo. En dicha estructura se almacenan las claves de los registros del archivo, junto a la referencia de acceso a cada registro asociado a la clave. Es necesario que las claves permanezcan ordenadas.

Esta estructura de datos es otro archivo con registros de longitud fija, independientemente de la estructura del archivo original. La característica fundamental de un índice es que posibilita imponer orden en un archivo sin que realmente este se reacomode.

De este modo, con el razonamiento previo, el índice es un nuevo archivo ordenado, con registros de longitud fija, con la diferencia de que contiene solo un par de datos del archivo original, y, por ende, es mucho más pequeño. En el peor caso, se plantea de nuevo la situación de ordenar este nuevo archivo en memoria secundaria, con alguno de los criterios planteados previamente en este capítulo. Una vez disponible el índice, la búsqueda de un dato se realiza primero en el índice (a partir de la clave), de allí se obtiene la dirección efectiva del archivo de datos y luego se accede directamente a la información buscada.

Quedan por analizar, entonces, la generación y el mantenimiento de esta nueva estructura. En primer lugar, se tratará el caso del índice creado a partir de la clave primaria, denominado índice primario.

A modo de ejemplo, se plantea el siguiente archivo de datos, organizado con registros de longitud variable:

Dir. Reg.	Cía.	Código	Título	Grupo musical compositor	Intérprete
15	WAR	23	Early Morning	A-ha	A-ha
36	SON	13	Just a Like a Corner	Cock Robin	Cock Robin
83	BMG	11	Selva	La Portuaria	La Portuaria
118	SON	15	Take on Me	A-ha	A-ha
161	VIR	2310	Land of Confusion	Genesis	Phil Collins
209	VIR	1323	Summer Moved On	A-ha	A-ha
248	AME	2323	Africa	Toto	Toto
275	RCA	1313	Leave of New York	REM	REM
313	ARI	2313	Here Come The Rain Again	Eurythmics	Annie Lennox

Al crearse el archivo, con clave primaria formada por los atributos Cia + Código , se crea el índice primario asociado. El índice y el archivo quedarían de la siguiente manera:

Clave	Ref.	Dir. reg.	Registro de datos
AME2323	248	15	WAR | 23 | Early Morning | A-ha | A-ha
ARI2313	313	36	SON | 13 | Just a Like a Corner | Cock Robin | Cock Robin
BMG11	83	83	BMG | 11 | Selva | La Portuaria | La Portuaria
RCA1313	275	118	SON | 15 | Take on Me | A-ha | A-ha
SON13	36	161	VIR | 2310 | Land of Confusion | Genesis | Phil Collins
SON15	118	209	VIR | 1323 | Summer Moved On | A-ha | A-ha
VIR1323	209	248	AME | 2323 | Africa | Toto | Toto
VIR2310	161	275	RCA | 1313 | Leave of New York | REM | REM
WAR23	15	313	ARI | 2313 | Here Come The Rain Again | Eurythmics | Annie Lennox

La tabla de la izquierda presenta el índice generado ordenado por la clave primaria del archivo de datos, en tanto que la tabla de la derecha muestra el archivo de datos propiamente dicho, utilizando registros de longitud variable. Es de notar que el archivo de datos no está ordenado.

Para realizar cualquier búsqueda de datos, por ejemplo, si se desea buscar el compositor del tema "Here Come The Rain Again", en primer lugar se busca la clave primaria (ARI2313) en el índice. Dado que el índice es un archivo de registros con longitud fija, es posible realizar sobre este una búsqueda binaria. Si además el índice cabe en memoria principal, puede ser transferido hacia allí y luego efectuarse la búsqueda.

Una vez hallada la referencia (313), correspondiente a la dirección del primer byte del registro buscado, se accede directamente al archivo de datos según lo indicado en dicha referencia.

Resumiendo, solo se realiza una búsqueda con potencial bajo costo en el índice y luego un acceso directo al archivo de datos, por lo que la cantidad de accesos a memoria secundaria está condicionada por la búsqueda en el índice. La mejor estructura y organización para los índices será un tema analizado en los capítulos siguientes.

Retomando la generación y el mantenimiento del índice, se analizarán brevemente estas operaciones.

Creación de índice primario

Al crearse el archivo, se crea también el índice asociado, ambos vacíos, solo con el registro encabezado.

Altas en índice primario

La operación de alta de un nuevo registro al archivo de datos consiste simplemente en agregar dicho registro al final del archivo. A partir de esta operación, con el NRR o la dirección del primer byte, según corresponda, más la clave primaria, se genera un nuevo registro de datos a insertar en forma ordenada en el índice. En caso de que el índice se encuentre en memoria principal, la inserción implica un costo relativamente bajo. Si se encuentra en memoria secundaria, el costo es mayor.

Modificaciones en índice primario

Se considera la posibilidad de cambiar cualquier parte del registro excepto la clave primaria. Si el archivo está organizado con registros de longitud fija, el índice no se altera.

Si el archivo está organizado con registros de longitud variable y el registro modificado no cambia de longitud, nuevamente el índice no se altera. Si el registro modificado cambia de longitud, particularmente agrandando su tamaño, este debe cambiar de posición, es decir, debe reubicarse. En este caso, esta nueva posición del registro es la que debe quedar asociada en la clave primaria respectiva del índice.

Bajas en índice primario

En este caso, eliminar un registro del archivo de datos implica borrar la información asociada en el índice primario. Así, se debe borrar física o lógicamente el registro correspondiente en el índice. Se debe notar que no tiene sentido recuperar el espacio físico en el índice con otro registro que se inserte, debido a que este archivo índice está ordenado y el elemento a insertar no debe alterar este orden.

Ventajas del índice primario

La principal ventaja que posee el uso de un índice primario radica en que, al ser de menor tamaño que el archivo asociado y tener registros de longitud fija, posibilita mejorar la *performance* de búsqueda. Además, se pueden realizar búsquedas binarias, mientras que en el archivo original esto quedaba condicionado a que contuviera registros de longitud fija. Más adelante, en los siguientes capítulos, se analizará la utilización de estructuras de datos para índices, más eficientes en términos de búsqueda, que permitan mejorar la *performance* de una búsqueda binaria.

Índices para claves candidatas

Las claves candidatas son claves que no admiten repeticiones de valores para sus atributos, similares a una clave primaria, pero que por cuestiones operativas no fueron seleccionadas como clave primaria. El tratamiento de un índice que soporte una clave candidata es similar al definido anteriormente para un índice primario. En los próximos capítulos, se ejemplificarán con más detalle el uso, la organización y las ventajas de estos índices.

Índices secundarios

La pregunta que seguramente surgió al buscar el compositor de la canción "Here Come The Rain Again" por la clave primaria (ARI2313) es cómo saber este dato, la clave primaria. No es natural ni intuitivo solicitar un dato por clave primaria, sino por el nombre de la canción o eventualmente por autor, que son atributos mucho más fáciles de recordar. Estos atributos, nombre de canción o autor, podrían contener valores repetidos en el archivo original. Por este motivo, no es posible pensarlos como parte de una clave primaria. La clave que soporta valores repetidos se denomina clave secundaria.

Por lo tanto, es necesario crear otro tipo de índice mediante el cual se pueda acceder a la información de un archivo, pero con datos fáciles de recordar. De esta manera surge el uso de índices secundarios.

Un índice secundario es una estructura adicional que permite relacionar una clave secundaria con una o más claves primarias, dado que, como ya se ha mencionado previamente, varios registros pueden contener la misma clave secundaria. Luego, para acceder a un dato deseado, primero se accede al índice secundario por clave secundaria; allí se obtiene la clave primaria, y se accede con dicha clave al índice primario para obtener finalmente la dirección efectiva del registro que contiene la información buscada. El lector podría preguntarse por qué motivo el índice secundario no posee directamente la dirección física de elemento de dato. La respuesta es simple: al tener la dirección física solamente definida para la clave primaria, si el registro cambia de lugar en el archivo, solo debe actualizarse la clave primaria. Suponga que para un archivo cualquiera se tienen definido un índice primario y cuatro secundarios. Si se modifica la posición física de un registro en el archivo de datos, los cinco índices deberían modificarse. Para mejorar esta situación, solo el índice primario tiene la dirección física y es el único que debe alterarse; todos los índices secundarios permanecen sin cambios. Solamente deberían ser

modificados si la clave primaria fuera alterada, situación que virtualmente no ocurre. Este tema será tratado en el Capítulo 12. Continuando con los datos del ejemplo antes presentado, en la siguiente tabla se muestra un índice secundario (índice de grupos de música).

Clave secundaria	Clave primaria
A-ha	SON15
A-ha	VIR1323
A-ha	WAR23
Cock Robin	SON13
Eurythmics	ARI2313
Genesis	VIR2310
La Portuaria	BMG11
REM	RCA1313
Toto	AME2323

En el ejemplo puede verse que, cuando la clave se repite, existe un segundo criterio de ordenación, establecido por la clave primaria.

Si se desea buscar un tema del grupo REM, primero se busca la clave REM en el índice secundario, se obtiene allí la clave primaria RCA1313 y finalmente se accede al índice primario para obtener luego la dirección física del archivo (275).

Nuevamente, el índice secundario está almacenado en un archivo con registros de longitud fija, y es de notar que es posible tener más de un índice secundario por archivo. Se deben analizar las operaciones básicas sobre el archivo que contiene al índice secundario: crear, agregar, modificar y eliminar.

Creación de índice secundario

Al implantarse el archivo de datos, se deben crear todos los índices secundarios asociados, naturalmente vacíos, solo con el registro encabezado. Esta operación es similar a la definida para índices primarios.

Altas en índice secundario

Cualquier alta en el archivo de datos genera una inserción en el índice secundario, que implica reacomodar el archivo en el cual se almacena. Esta operación es de bajo costo en términos de *performance* si el índice puede almacenarse en memoria principal, pero resulta muy

costosa si está en memoria secundaria, considerando además que potencialmente puede existir (como en el ejemplo visto) un segundo nivel de ordenación, determinado por la clave primaria.

Modificaciones en índice secundario

Aquí se deben analizar dos alternativas. La primera de ellas ocurre cuando se produce un cambio en la clave secundaria. En este caso, se debe reacomodar el índice secundario, con los costos que ello implica. El segundo caso ocurre cuando cambia el resto del registro de datos (excepto la clave primaria), no generando así ningún cambio en el índice secundario.

Bajas en índice secundario

Cuando se elimina un registro del archivo de datos, esta operación implica eliminar la referencia a ese registro del índice primario más todas las referencias en índices secundarios.

La eliminación a realizarse en el índice secundario, almacenado en un archivo con registros de longitud fija, puede ser física o lógica.

Una alternativa interesante es borrar solo la referencia en el índice primario. En este caso, el índice primario actúa como una especie de barrera de protección, que aísla a los índices secundarios de este tipo de cambios en el archivo de datos. El beneficio principal es que el índice secundario no se altera, pero existe un costo adicional representado en el espacio ocupado de datos que ya no existen. Si el archivo de datos fuese poco volátil, este no sería un gran problema, pero si fuera volátil, se deberían programar borrados físicos de los índices secundarios.

Alternativas de organización de índices secundarios

Se ha visto que el índice secundario debe reacomodarse con cada alta realizada sobre el archivo de datos, aun si se ingresa una clave secundaria ya existente. Esto se debe a que existe un segundo nivel de ordenación por la clave primaria asociada al registro ingresado.

La organización de índices presentada hasta el momento consiste en almacenar la misma clave secundaria en distintos registros, tantas como ocurrencias haya. Esto implica mayor espacio, generando una menor posibilidad de que el índice quepa en memoria.

La primera alternativa de mejora sobre el espacio ocupado por el índice secundario implica almacenar en un mismo registro todas las ocurrencias de la misma clave secundaria. Con el ejemplo presentado previamente, esta nueva organización consiste en que cada registro esté formado por la clave secundaria, más un arreglo de claves primarias correspondientes a dicha clave. En el ejemplo presentado:

A-ha	SON15	VIR1323	WAR23

En este caso, al agregar un nuevo registro al archivo de datos con la misma clave secundaria, no genera una inserción en el índice secundario. Solo se debe insertar en forma ordenada la clave primaria en el vector de claves primarias respectivo.

El problema inherente a esta alternativa es la elección del tamaño del registro, ya que conviene que este sea de longitud fija. Esto implica decidir el tamaño del vector, por lo que puede haber casos en que resulte insuficiente o que sobre espacio (provocando fragmentación). Por lo tanto, si bien es una alternativa razonable, la solución a un inconveniente produce potencialmente otro problema.

Otra alternativa es pensar en una lista de claves primarias asociada a cada clave secundaria. De esta manera, no se debe realizar reserva de espacio y puede existir cualquier número de claves primarias por cada clave secundaria.

Con el ejemplo que se ha presentado, el índice secundario quedaría organizado del siguiente modo:

NRR	Clave secundaria	Puntero	NRR	Clave primaria	Enlace
0	A-ha	2	0	VIR2310	–1
1	Cock Robin	3	1	BMG11	–1
2	Eurythmics	4	2	SON15	5
3	Genesis	0	3	SON13	–1
4	La Portuaria	1	4	ARI2313	–1
5	REM	6	5	VIR1323	8
6	Toto	7	6	RCA1313	–1
			7	AME2323	–1
			8	WAR23	–1

Analizando brevemente el ejemplo, se puede observar que, para acceder a las claves primarias asociadas a las secundarias, primeramente se accede al archivo que contiene solo las claves secundarias, y allí se obtiene la dirección (NRR) de acceso a la lista de claves primarias asociadas. Dicha lista, denominada lista invertida, se encuentra almacenada en otro archivo, el cual se recorre de acuerdo con el camino establecido por el atributo enlace, que indica cuál es el próximo registro asociado, que en caso de no existir, contendrá un puntero nulo (enlace = –1, en el ejemplo).

Ambos archivos están organizados con registros de longitud fija.

Esta opción tiene las siguientes ventajas:

- El único reacomodamiento en el índice se produce cuando se agrega una nueva clave primaria. Igualmente el índice es más pequeño, por lo que dicho reacomodamiento resulta menos costoso.
- Si se agregan o borran datos de una clave secundaria ya existente, solo se debe modificar el archivo que contiene la lista, y en cada caso solo se debe modificar una lista.
- Dado que se generan dos archivos, uno de ellos podría residir en memoria secundaria, liberando así memoria principal. No obstante, si existiesen muchos índices secundarios, el mantenimiento de dos archivos por cada índice podría resultar muy costoso.

Índices selectivos

Existe otra posibilidad que consiste en disponer de índices que incluyan solo claves asociadas a una parte de la información existente, es decir, aquella información que tenga mayor interés de acceso. En el ejemplo presentado, se podría tener un índice con las canciones del grupo musical A-ha solamente.

De esta forma, el índice incluye solo un subconjunto de datos asociado a los registros del archivo que interesa consultar, generando una estructura que actúa como filtro de acceso a la información de interés en el archivo.

Obviamente, en este caso, solo se deben considerar las modificaciones al archivo, vinculadas con los datos presentes en el índice.

Cuestionario del capítulo

1. ¿Por qué tiene tanta importancia la operación de búsqueda de información?
2. ¿Qué motiva a analizar solo la cantidad de accesos a disco cuando se habla de análisis de *performance* de operaciones de búsqueda?
3. ¿Cuál es el costo de tener un archivo físicamente ordenado?
4. ¿Qué es un índice? ¿Para qué sirve?
5. ¿Qué diferencias existen entre índices primarios, candidatos y secundarios?

Ejercitación

1. Suponga que tiene un archivo de datos desordenado con 1.600.000 registros de 200 bytes cada uno y una clave primaria que ocupa 20 bytes, y solo dispone de 4 Mb de memoria. Discuta cómo podría ordenar este archivo.
2. Sobre el ejercicio anterior se podrían haber generado particiones como parte del proceso de ordenación.
   a. ¿Cuántas particiones utilizando sort interno se pudieron generar?
   b. Discuta el costo de ordenar las particiones con un proceso clásico de merge que utilice todos los archivos intermedios generados simultáneamente.
   c. Discuta nuevamente el inciso anterior, proponiendo una intercalación en dos pasos.
   d. Resuelva el caso anterior con una intercalación en tres pasos.
   e. Compare los resultados obtenidos en los incisos b, c y d. ¿Qué puede concluir al respecto?

CAPÍTULO 6

Árboles. Introducción

Objetivo

En el Capítulo 5, se discutió sobre la necesidad de localizar rápidamente alguna información particular dentro de un archivo. Se debe tener en cuenta que, sobre una BD, aproximadamente 80% de las operaciones efectuadas consisten en búsqueda de información. Por lo tanto, es de gran importancia poder recuperar un dato requerido con la menor cantidad de accesos sobre disco. Es así que la organización física que se determine para el archivo resultará de suma importancia para la *performance* final de cada una de las operaciones de consulta.

Se discute en este capítulo la utilización de árboles binarios de búsqueda, como forma de organizar los datos en un archivo sobre un dispositivo de almacenamiento secundario, omitiéndose los conceptos básicos de la estructura árbol. Se presentan las ventajas y desventajas de esta política y, asimismo, se plantean los algoritmos que permiten implantar árboles binarios sobre disco.

Árboles binarios

El principal problema para poder administrar un índice en disco rígido lo representa la cantidad de accesos necesarios para recuperar la información. Como se discutió en el Capítulo 5, el proceso de búsqueda de información es costoso y, además, el mantenimiento de la información ante operaciones de altas, bajas y modificaciones también debe ser considerado como parte del proceso de actualización del archivo de datos.

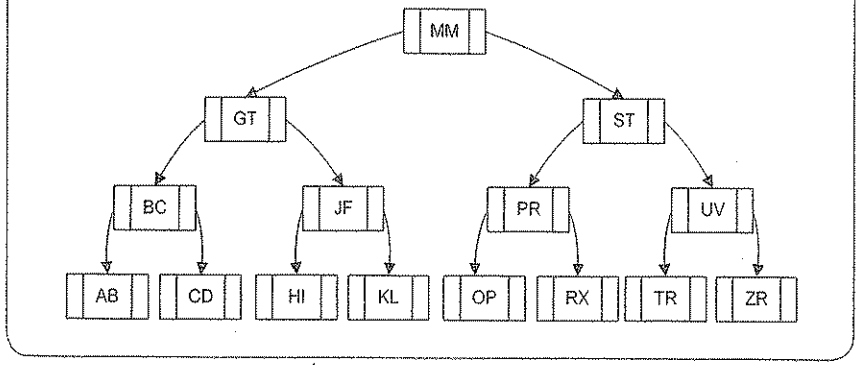
Las estructuras tipo árbol presentan algunas mejoras tanto para la búsqueda como para el mantenimiento del orden de la información. Esta aseveración será discutida a lo largo de este capítulo.

Los árboles binarios representan la alternativa más simple para la solución de este problema. Es probable que el lector haya estudiado temas relacionados con árboles binarios, e implantado soluciones recursivas o iterativas utilizando estructuras de datos dinámicas sobre memoria RAM. Se presentarán aquí las variantes que deben llevarse a cabo para que los algoritmos tradicionales puedan implantarse sobre estructuras que permitan almacenar la información en dispositivos secundarios.

> Un **árbol binario** es una estructura de datos dinámica no lineal, en la cual cada nodo puede tener a lo sumo dos hijos.

La estructura de datos árbol binario en general tiene sentido cuando está ordenado. En ese caso, como lo muestra la Figura 6.1, los elementos que se agregan en un árbol se insertan manteniendo un orden. Esto es, a la izquierda de un elemento se encuentran los elementos menores que él, y a la derecha, los mayores.

FIGURA 6.1



La búsqueda en este tipo de estructuras se realiza a partir del nodo raíz y se recorre, explorando hacia los nodos hoja. Se chequea un nodo; si es el deseado, la búsqueda finaliza o, en su defecto, se decide si la búsqueda continúa a izquierda o a derecha descartando la mitad de los elementos restantes. Por este motivo, la búsqueda de información en un árbol binario es de orden logarítmico. Encontrar un elemento es del orden $log_2 (N)$, siendo N la cantidad de elementos distribuidos en el árbol.

Es probable que el lector haya implementado algoritmos de árboles binarios sobre memoria RAM. Sin embargo, para poder utilizar estas ideas como soporte de índices de búsqueda, es necesario que los árboles binarios se implanten sobre almacenamiento secundario.

La Figura 6.2 presenta la definición de datos necesaria para generar un archivo de índices utilizando árboles binarios sobre disco. El lector puede observar que el registro de datos contiene tres campos. El campo `elemento_de_dato` contendrá la clave del índice, los campos `hijo_izquierda` e `hijo_derecha` definen la dirección del hijo menor y mayor, respectivamente. Mientras que sobre memoria RAM estos hijos se representaban mediante punteros, ahora el valor de cada uno es representado mediante un valor entero. Ese valor indica el NRR del hijo dentro del archivo de datos.

Figura 6.2

```
type   registroarbolbinario = record
             elemento_de_dato:  tipo_de_dato;
             hijo_izquierda,
             hijo_derecha     :  integer;
       end;

       indicebinario = file of registroarbolbinario;
```

La Figura 6.3 representa un formato posible del archivo de datos que se corresponde con el árbol presentado en la Figura 6.1. Es posible observar que cada elemento tiene la dirección de sus respectivos hijos y, en caso de tratarse de un nodo terminal u hoja, con un valor negativo se indica que no dispone de hijos.

Figura 6.3

Raíz → 0

	Clave	Hijo Izq	Hijo Der
0	MM	1	2
1	GT	3	4
2	ST	8	11
3	BC	5	6
4	JF	7	14
5	AB	-1	-1
6	CD	-1	-1
7	HI	-1	-1

	Clave	Hijo Izq	Hijo Der
8	PR	9	10
9	OP	-1	-1
10	RX	-1	-1
11	UV	12	13
12	TR	-1	-1
13	ZR	-1	-1
14	KL	-1	-1

El Ejemplo 6.1 muestra el seudocódigo del algoritmo que permite la creación/inserción de nuevos elementos de datos sobre un archivo que almacena un índice estructurado en un árbol binario.

EJEMPLO 6.1

```
Type
  Nodo = record
    Elemento: tipo_dato_elemento;
    Hijo_Izquierdo: integer;
    Hijo_Derecho:integer;
End;
Archivo: file of Nodo;

Procedure Insertar (var A: Archivo, elem:
                    tipo_dato_elemento)
Var
  Raiz, nodo_nuevo: Nodo;
  Pos_nuevo_nodo:   integer;
  Encontre_Padre:   boolean;

Begin
Reset(A);
With nodo_nuevo do
  Elemento   :=elem;
  Hijo_izquierdo    := -1;
  Hijo_Derecho      := -1;
End;

If Eof(A) Then {significa que es un árbol vacío y el
elemento es insertado como raíz}
  Write(A,nodo_nuevo);
Else
  Read(A, Raiz);
  Pos_nuevo_nodo := filesize(A);
  Seek(A, pos_nuevo_nodo); {posicionarse al final del
  archivo}
  Write(A, nodo_nuevo); {escribir el nuevo nodo al final}
  Encontre_Padre := false;

  {buscar al padre para agregar la referencia al nuevo
  nodo}
  While not (Encontre_Padre) do
  begin
      If (Raiz.elemento > nodo_nuevo.elemento) Then
        If (Raiz.hijo_izquierdo <> -1) Then
           Seek(A, Raiz.hijo_izquierdo);
           Read(A, Raiz);
        Else
           Raiz.hijo_izquierdo := Pos_nuevo_nodo;
           Encontre Padre:= true;
```

continúa >>>

```
        Else
          If (Raiz.hijo_derecho <> -1) Then
             Seek(A, Raiz.hijo_derecho);
             Read(A, Raiz);
          Else
             Raiz.hijo_derecho := Pos_nuevo_nodo;
             Encontre_Padre := true;
          end;
        end;
    end;
    {raiz es el padre y ya lo leí, debo volver a
    posicionarme}
    Seek(A, Filepos(A)-1);

    {guardo al padre con la nueva referencia}
    Write (A, raiz);
end.
```

Performance de los árboles binarios

El Ejemplo 6.2 presenta el proceso que permite buscar dentro del archivo anterior (que representa un árbol binario) un elemento particular.

EJEMPLO 6.2

```
Type
  Nodo = record
      Elemento: tipo_dato_elemento;
      Hijo_Izquierdo: integer;
      Hijo_Derecho:integer;
End;
Archivo : file of Nodo;

Function Buscar (var A:archivo, elem:tipo_dato_elemento):
integer
{retorna el NRR del nodo; si el árbol es vacio, retorna
-1}

Var
  Raiz: nodo;
  Encontre:boolean;
  Pos:integer;

Begin

  Reset(A);
  Encontre := False;
  Pos      := -1;
```

continúa >>>


```
If not eof(A) Then
 begin
     Read(A, Raiz);

     While not (encontre) and not eof(A) do
          'If(raiz.elemento > elem) then
             Pos := raiz.hijo_izquierdo;
             Seek(pos);
             Read(A, Raiz);
           Else
             If (Raiz.elemento < elem) then
                Pos:= raiz.hijo_derecho;
                Seek(pos);
                Read(A, Raiz);
             Else
                Encontre := true;

 End;

 {pos tiene por defecto -1 bsino el NRR del nodo donde
 está el elemento buscado}
 Buscar := pos;
End
```

Se puede observar que la búsqueda de un elemento de datos comienza siempre desde la raíz, recorriendo a izquierda o a derecha según el elemento que se desea ubicar. De esta forma, en cada revisión se descarta la mitad del archivo restante, donde el dato buscado seguro no se encuentra. Así, como se mencionó anteriormente, la *performance* para buscar un elemento es del orden $Log_2(N)$ accesos a disco, siendo N la cantidad de nodos (elementos de datos) que contiene el árbol. Esta organización es comparable a lo discutido en el Capítulo 5: el algoritmo de búsqueda sobre un archivo de índices ordenado por clave tiene el mismo orden de *performance*.

Una ventaja de la organización mediante árboles binarios está dada en la inserción de nuevos elementos. Mientras que un archivo se desordena cuando se agrega un nuevo dato, si la organización se realiza con la política de árbol binario, la operatoria resulta más sencilla en términos de complejidad computacional. La secuencia de pasos para insertar un nuevo elemento es la siguiente:

1. Agregar el nuevo elemento de datos al final del archivo.
2. Buscar al padre de dicho elemento. Para ello se recorre el archivo desde la raíz hasta llegar a un nodo terminal.
3. Actualizar el padre, haciendo referencia a la dirección del nuevo hijo.

Se puede observar la *performance* desde el punto de vista de accesos a disco: se deben realizar $Log_2(N)$ lecturas, necesarias para localizar el padre del nuevo elemento, y dos operaciones de escritura (el nuevo elemento y la actualización del padre). De este modo, si se compara este método con reordenar todo el archivo, como se presentó en el Capítulo 5, esta opción resulta mucho menos costosa en términos de *performance* final.

La operación de borrado presenta un análisis similar. Para quitar un elemento de un árbol, este debe ser necesariamente un elemento terminal. Si no lo fuera, debe intercambiarse el elemento en cuestión con el menor de sus hijos mayores. Suponga que se desea quitar el elemento ST del árbol de la Figura 6.1; al no ser un elemento terminal, debe ser reemplazado por TR, que representa al menor de sus hijos mayores.

Esta operatoria de borrado, en términos de acceso a disco, significa reemplazar la llave TR por la llave ST, en el registro 2 del archivo de datos de la Figura 6.3, y, posteriormente, borrar el registro 12. Así, deberán realizarse nuevamente $Log_2(N)$ lecturas (necesarias para localizar tanto a ST como TR) y dos escrituras. La primera de ellas, para dejar a TR en el registro 2, y la segunda, para marcar como borrado al registro 12. Nótese que sobre el archivo se pueden aplicar las técnicas de recuperación de espacio discutidas en el Capítulo 4.

Como conclusión, los árboles binarios representan una buena elección en términos de inserción y borrado de elementos, que supera ampliamente los resultados obtenidos en el Capítulo 5. En lo que se refiere a operaciones de búsqueda, los árboles binarios tienen un comportamiento similar al comportamiento obtenido al disponer de un archivo ordenado.

Archivos de datos vs. índices de datos

Suponga el lector que se dispone de un archivo de datos que contempla los clientes de cierta compañía. La estructura del archivo se presenta en la Figura 6.4. Sobre este archivo se necesita realizar búsquedas de información por código de cliente, por nombre de cliente, por tarjeta de identificación o por fecha de nacimiento, indistintamente.

La pregunta es: ¿por qué criterio se ordena el archivo? El árbol binario que se genera ¿qué tipo de orden va a contemplar? ¿Se van a generar cuatro copias del archivo de manera que cada una de ellas esté ordenada por un criterio diferente?

Figura 6.4

```
type  clientes = record
        código           :  integer;
        nombre           :  string;
        dni              :  string;
        fecha_nacimiento:  fecha;
      end;
      archivocliente = file of clientes;
```

La respuesta a las preguntas anteriores es que son necesarios los cuatro ordenamientos y, por ende, generar cuatro árboles binarios diferentes. Lo que el lector debe contemplar en este momento es que cada estructura ordenada solo necesita el atributo por el cual ordena. Así, un árbol binario que ordena por tarjeta de identidad solo debe tener esta información.

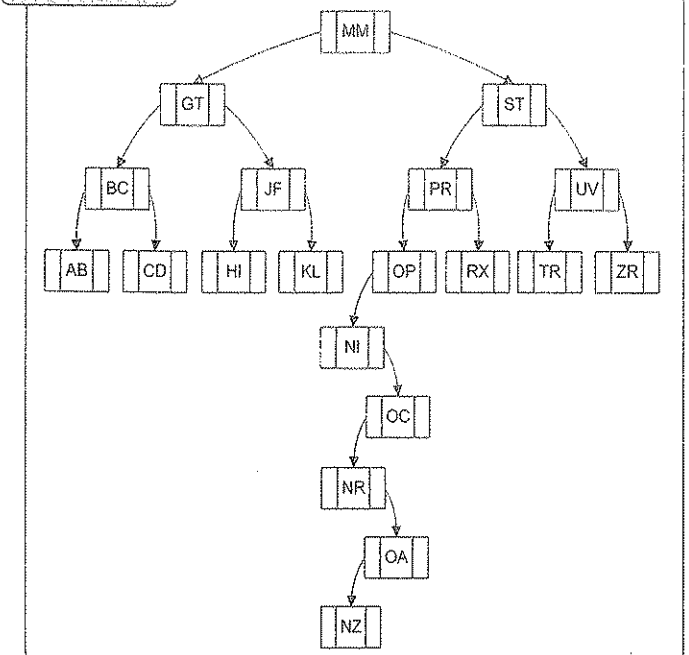
De este modo, y siguiendo lo discutido en el Capítulo 5, se debe separar el archivo de datos de los índices de dicho archivo. El archivo de datos se trata como un archivo serie, donde cada elemento es insertado al final del archivo y no hay orden físico de datos. Se genera, además, un archivo por cada árbol binario que necesite implantar un índice diferente. Cada archivo de índice tendrá la estructura presentada en la Figura 6.3, con la inclusión de la dirección (NRR) del registro completo en el archivo de datos.

Problemas con los árboles binarios

El desempeño de la búsqueda en un árbol binario –y, en particular, en el representado en la Figura 6.1– es bueno porque el árbol se encuentra balanceado. Se entiende por árbol balanceado a aquel árbol donde la trayectoria de la raíz a cada una de las hojas está representada por igual cantidad de nodos. Es decir, todos los nodos hoja se encuentran a igual distancia del nodo raíz.

¿Qué sucedería si, sobre el árbol de la Figura 6.1, se insertaran en el siguiente orden las claves `NI, OC, NR, OA, NZ`? La Figura 6.5 presenta este caso. Se puede observar que ahora el árbol se encuentra desbalanceado.

Figura 6.5



La *performance* de búsqueda ya no puede considerarse más en el orden logarítmico. El caso degenerado de un árbol binario transforma al mismo en una estructura tipo lista y, en ese caso, la *performance* de búsqueda decae, transformándose en orden lineal.

La pregunta a responder entonces es: ¿en qué situaciones un árbol binario resulta una buena organización para un índice? La respuesta es: cuando el árbol binario está balanceado. La correcta elección de la raíz del árbol determinará si el mismo permanecerá balanceado o no. No obstante, cuando se genera un archivo que implanta un índice de búsqueda, es imposible a priori determinar cuál es la mejor raíz, dado que dependerá de los elementos de datos que se inserten.

Árboles AVL

La situación planteada anteriormente está resuelta. Los árboles balanceados en altura son árboles binarios cuya construcción se determina respetando un precepto muy simple: la diferencia entre el camino más corto y el más largo entre un nodo terminal y la raíz no puede diferir en

más que un determinado delta, y dicho delta es el nivel de balanceo en altura del árbol. Otra definición posible plantea que un árbol balanceado en altura es aquel en el que la diferencia máxima entre las alturas de cualquiera de dos subárboles que tienen raíz común no supera un delta determinado.

Así, un árbol AVL es un árbol balanceado en altura donde el delta determinado es uno, es decir, el máximo desbalanceo posible es uno. En el caso del árbol de la Figura 6.5, al insertar la clave NI, el árbol aún se mantiene balanceado como AVL(1). Sin embargo, cuando se intenta insertar OC, se viola el precepto establecido. Se debe llevar a cabo, en esta situación, un rebalanceo de este árbol. No es propósito de este libro definir con detalle este proceso. Se debe comprender que, si bien este algoritmo es relativamente sencillo de implementar, los costos computacionales de acceso a disco aumentan considerablemente, por lo que su implantación deja de ser viable.

Entonces, se dispone de una estructura capaz de mantener el balanceo acotado, pero asumiendo mayores costos en las operaciones de inserción y borrado. Por lo tanto, los árboles binarios y los AVL no representan una solución viable para los índices del archivo de datos.

Paginación de árboles binarios

En capítulos anteriores, se discutió el concepto de *buffering*. Cuando se transfiere desde o hacia el disco rígido, dicha transferencia no se limita a un registro, sino que son enviados un conjunto de registros, es decir, los que quepan en un *buffer* de memoria. Las operaciones de lectura y escritura de datos de un archivo utilizando *buffers* presentan una mejora de *performance*. Este concepto es de utilidad cuando se genera el archivo que contiene el árbol binario. Dicho árbol se divide en páginas, es decir, se pagina, y cada página contiene un conjunto de nodos, los cuales están ubicados en direcciones físicas cercanas, tal como se muestra en la Figura 6.6.

Así, cuando se transfieren datos, no se accede al disco para transferir unos pocos bytes, sino que se transmite una página completa.

Una organización de este tipo reduce el número de accesos a disco necesarios para poder recuperar la información. En el árbol de la Figura 6.6 se puede notar que, al transferir la primera página, se están recuperando siete nodos del árbol, los que representan los primeros tres niveles del árbol. Así, y siguiendo dicha figura, con solo dos accesos a disco es posible recuperar un nodo específico entre 63. Esto es considerando que para transferir una página completa, si bien se accede a direcciones físicas diferentes, tales direcciones son cercanas y los desplazamientos realizados para su acceso son despreciables.

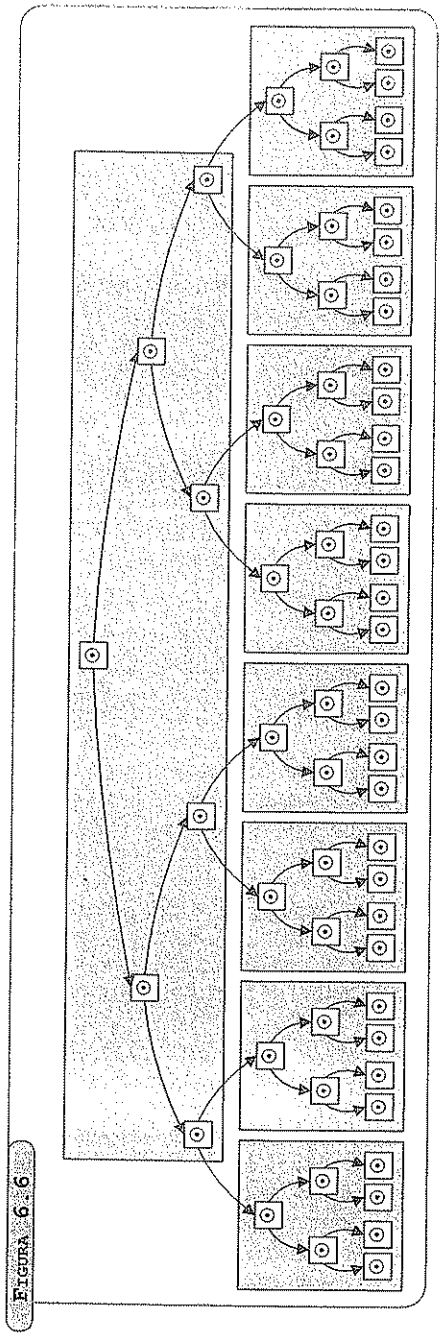


FIGURA 6.6

Al dividir un árbol binario en páginas, es posible realizar búsquedas más rápidas de datos almacenados en memoria secundaria. Se puede observar que, en el ejemplo expuesto, cada página contiene siete elementos, pero es solo para presentar la situación. En una situación más realista, una página puede contener 255 o 511 nodos del árbol, con lo cual la altura relativa decrece rápidamente, permitiendo recuperar un elemento del árbol en muchos menos accesos que los planteados originalmente ($log_2(N)$). Ahora, para analizar la *performance* resultante, deberían tenerse en cuenta la cantidad de nodos que caben en una página. Así, suponiendo que en un *buffer* caben 255 elementos, el tamaño de cada página sería entonces de 255 nodos, resultando la *performance* final de búsqueda del orden de $log_{256}(N)$, es decir, $log_{k+1}(N)$, siendo N la cantidad de claves del archivo y K la cantidad de nodos por página.

La cuestión a analizar, ahora, es cómo generar un árbol binario paginado. Para que el uso de las páginas tenga sentido, los siete elementos correspondientes a la misma (de acuerdo con la Figura 6.6) deberían llegar consecutivos para almacenarse en el mismo sector del disco. En su defecto, sería imposible que una página contuviese los elementos relacionados. Como esta opción es imposible, no se puede condicionar la llegada de elementos al archivo de datos, y es menester encontrar otra solución. Así, cada página deberá quedar incompleta si al llegar un elemento no estuviese relacionado con los anteriores. Es decir, un elemento deberá estar contenido en la página que le corresponde, en lugar de quedar en la primera disponible. La Figura 6.7 presenta gráficamente la forma que tendría un árbol binario paginado de acuerdo con las claves que se insertan en él.

Para lograr esta construcción, hay que pensar en páginas, los elementos que caben en cada una de ellas, la forma en que se construye un árbol binario y, además, los temas relacionados con el balanceo. Dividir el árbol en páginas implica un costo extra necesario para su reacomodamiento y para mantener su balanceo interno. Un algoritmo que soporte esta construcción será muy costoso de implementar y luego también en cuanto a *performance*.

Aquí se plantea una disyuntiva, el paginado de árboles binarios con su beneficio y el costo que ello trae aparejado. La solución para este problema consiste en adoptar la idea de manipular más de un registro (es decir, la página) y tratar de disponer de algoritmos a bajo costo para construir un árbol balanceado.

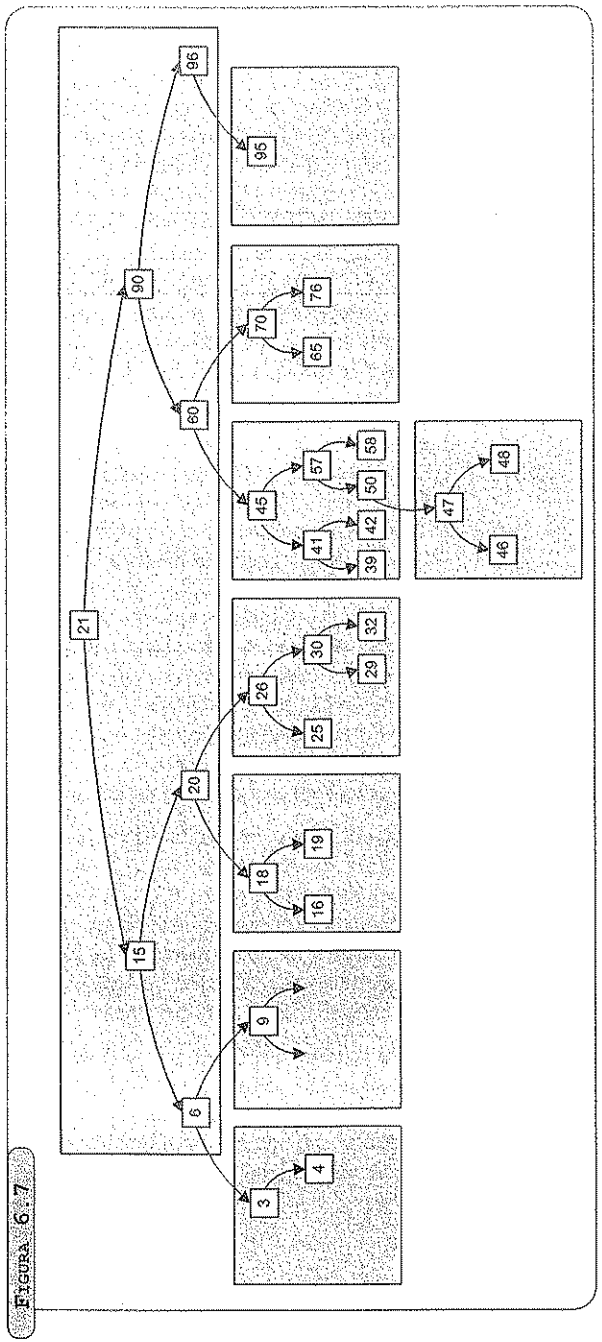


Figura 6.7

Árboles multicamino

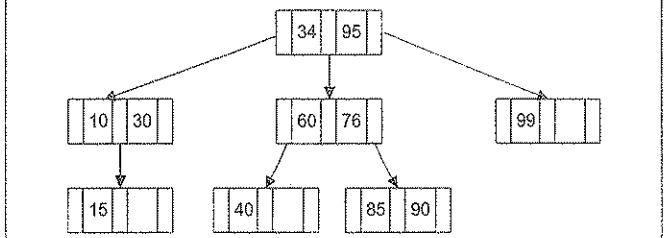
Los árboles binarios son un tipo de estructura de datos que resulta muy simple, pero luego de presentar el concepto de árboles binarios paginados se deberían replantear las siguientes ideas: ¿por qué los árboles deben ser binarios?, ¿por qué no es posible tener un árbol ternario?, ¿qué sería tener un árbol ternario? y, por último, ¿cómo generalizar estas ideas?

Anteriormente se definió a un árbol binario como una estructura de datos en la cual cada nodo puede tener cero, uno o dos hijos. Siguiendo esta línea se podría definir a un árbol ternario como aquel en el cual cada nodo puede tener cero, uno, dos o tres hijos.

La Figura 6.8 presenta un árbol ternario. En el nodo raíz hay dos elementos, el 34 y el 95; a la izquierda del 34, como primer puntero se referencia a un nodo que contiene los elementos menores que 34. Los valores comprendidos entre 34 y 95 se referencian por un puntero existente entre ambos, en tanto que a la derecha del 95 se apunta hacia el nodo que contiene los elementos mayores.

Debe notarse que no todos los nodos están "completos" en cuanto a elementos y en punteros, es decir, hay nodos con lugares libres y con la posibilidad de tener más hijos. En definitiva y en esencia, un árbol ternario es similar a uno binario, solo que cada nodo tiene la posibilidad de poseer hasta dos elementos y hasta tres hijos.

Figura 6.8



La Figura 6.7 presentó un árbol binario paginado, donde cada página contiene siete nodos. ¿Qué sucede ahora si, en lugar de pensar en un árbol binario paginado, que contiene siete nodos repartidos en tres niveles, se construye un árbol que contenga en un nodo los siete elementos? La Figura 6.9 presenta la estructura que debería tener dicho nodo. Se puede observar que la misma contendrá ocho punteros hacia los hijos de ese nodo padre.

Figura 6.9

Un **árbol multicamino** es una estructura de datos en la cual cada nodo puede contener *k* elementos y *k*+1 hijos.

Se define el concepto de **orden de un árbol multicamino** como la máxima cantidad de descendientes posibles de un nodo.

Así, el orden de un árbol binario es dos (máxima cantidad de descendientes), el orden de un árbol ternario será tres, y, en general, un árbol multicamino de orden *M* contendrá un máximo de *M* descendientes por nodo.

Un árbol multicamino representa otra forma de resolver el concepto de página vertido anteriormente. En este caso, el orden del árbol dependerá del tamaño de la página y de los elementos que se coloquen en ella.

Sin embargo, queda latente aún el problema del balanceo. Como muestra la Figura 6.8, el árbol ternario generado mantiene este problema, el cual puede generalizarse a los árboles multicamino. Se ha dado respuesta parcial entonces a los inconvenientes planteados. Es posible redefinir la estructura de un árbol binario paginado para hacerla manejable como un árbol multicamino; sin embargo, deberá plantearse alguna solución al problema del balanceo. El Capítulo 7 aborda dicho problema.

Cuestionario del capítulo

1. ¿Qué ventaja presenta una organización tipo árbol para representar los índices de un archivo de datos?
2. ¿Por qué un árbol binario resulta insuficiente para organizar el índice?
3. ¿Los árboles AVL representan una solución viable? ¿Por qué motivo?
4. ¿Qué características ponderan los árboles binarios paginados como estructura alternativa?
5. En función de sus respuestas a la Pregunta 2, ¿los árboles multicamino solucionan dichos inconvenientes?

Ejercitación

1. A partir de los ejemplos en seudocódigo del capítulo, implemente el algoritmo que permita borrar un elemento de datos de un archivo que contiene un árbol binario.
2. Discuta la estructura de datos necesaria para almacenar un árbol ternario.
3. Discuta los algoritmos de creación, búsqueda y eliminación de elementos sobre un árbol ternario.
4. Discuta la estructura de datos necesaria para almacenar un árbol multicamino.
5. Sabiendo que la *performance* de búsqueda sobre un árbol binario balanceado es de orden $Log_2(N)$, discuta cuál sería la *performance* de búsqueda sobre un árbol ternario balanceado.
6. Generalice la respuesta anterior para un árbol multicamino.
7. ¿Por qué un árbol debe permanecer balanceado para asegurar las respuestas obtenidas en los Ejercicios 5 y 6?
8. Suponga que dispone de páginas de 4 Kb y que los elementos del árbol ocupan, cada uno, 20 bytes. Si se desea generar un árbol multicamino, ¿cuál podría ser el orden del mismo?

CAPÍTULO 7

Familia de árboles balanceados

Objetivo

En este capítulo se plantea la utilización de árboles balanceados como solución a la implantación de índices para controlar el acceso a la información contenida en archivos de datos. Los primeros tipos de árboles balanceados presentados son los árboles *B*; se discuten sus características, operaciones y *performance*. Luego, se presentan los árboles B^* como alternativa, nuevamente analizando sus características y *performance*.

Por último, se presentan y discuten los árboles *B+*, estructuras que permiten optimizar la búsqueda de información, ya que otorgan además un acceso secuencial eficiente.

Árboles *B* (balanceados)

Los **árboles *B*** son árboles multicamino con una construcción especial que permite mantenerlos balanceados a bajo costo.

Un árbol *B* de orden *M* posee las siguientes propiedades básicas:

1. Cada nodo del árbol puede contener, como máximo, *M* descendientes y *M*-1 elementos.
2. La raíz no posee descendientes directos o tiene al menos dos.
3. Un nodo con *x* descendientes directos contiene *x*-1 elementos.
4. Los nodos terminales (hojas) tienen, como mínimo, [*M*/2] – 1 elementos, y como máximo, *M*-1 elementos.
5. Los nodos que no son terminales ni raíz tienen, como mínimo, [*M* / 2] elementos.
6. Todos los nodos terminales se encuentran al mismo nivel.

Analizando las propiedades anteriores, la primera de ellas establece que si el orden de un árbol *B* es, por ejemplo, 256, esto indica que la máxima cantidad de descendientes de cada nodo es 256, porque este valor determina la cantidad de punteros disponibles.

La segunda propiedad posibilita que la raíz no disponga de descendientes, pero cuando sea necesario deberá contar con dos, como mínimo. En ningún caso, la raíz puede tener un solo descendiente.

La tercera característica determina que cualquier nodo que posea una cantidad determinada de hijos, por ejemplo, 256, necesariamente debe contener 255 elementos, es decir, uno menos.

La cuarta propiedad establece la mínima cantidad de elementos contenidos en un nodo terminal. Al ser *M* el orden del árbol, la máxima cantidad admisible de elementos será *M*-1, pero, además, la mínima cantidad de elementos es uno menos que la parte entera de la mitad del orden. Así, si 256 fuera el orden establecido, un nodo terminal deberá poseer 127 elementos como mínimo y 255 como máximo.

La quinta propiedad implica una restricción parecida a la anterior: cualquier nodo que no sea terminal o raíz debe tener una cantidad mínima de descendientes. Siguiendo con un árbol de orden 256, cada nodo deberá tener un mínimo de 128 descendientes.

La última propiedad establece el balanceo del árbol. La distancia desde la raíz hasta cada nodo terminal debe ser la misma. Esta opción permite analizar la *performance* de un árbol *B* y saber que esta *performance* será respetada sin importar cómo se construya, a partir de la llegada de los elementos de datos.

Hasta el momento, se dispone de una definición de árbol *B* y de seis propiedades. Se deberá ahora determinar la estructura de datos que brinde soporte a estos árboles y discutir las operaciones que permitan crear, buscar, borrar y mantener la información contenida en esta estructura de datos.

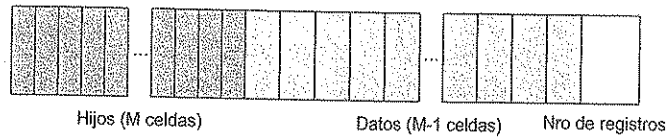
En el Capítulo 6, cuando se presentó una estructura tipo árbol como implementación posible para los índices de un archivo de datos, se discutió que dichas estructuras contendrían solamente la información para permitir el acceso eficiente a estos archivos. Así, por un lado, se manipula el archivo de datos como un archivo serie, y por otro, cada uno de los archivos que implementaban a los índices respectivos, con una referencia al registro completo en el archivo de datos. En este capítulo, se continúa en ese sentido. Los archivos que contendrán a los índices representados como estructuras de tipo árbol *B*, B^* o *B+* solamente contendrán información que permitirá el ordenamiento y la búsqueda eficiente, quedando el registro completo en el archivo de datos.

La Figura 7.1 presenta el formato de un nodo posible para un árbol *B* de orden *M*. Dicha figura se corresponde con la definición de la estructura de datos siguiente:

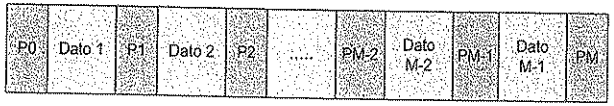
```
Const orden = 255;
Type reg_arbol_b = record;
          Hijos: array [0..orden] of integer;
          Claves: array [1..orden] of tipo_de_dato;
          Nro_registros: integer;
     End;
```

Donde `hijos` es un arreglo que contiene la dirección de los nodos que son descendientes directos, `claves` es un arreglo que contiene las claves que forman el índice del árbol y `nro_registros` indica la dimensión efectiva en uso de cada nodo, es decir, la cantidad de elementos del nodo. Nótese que el vector de `Hijos` es un elemento más grande que el vector de `Claves`. Nótese además que el orden del árbol en este caso es 256.

FIGURA 7.1



Formato del nodo para archivo del índice árbol b



Formato gráfico del nodo del índice árbol B

Operaciones sobre árboles *B*

En este apartado, se describen las operaciones `Crear/Insertar`, `Buscar`, `Borrar` y `Modificar` sobre un árbol *B*. Además, en cada caso, se analizará la *performance* de cada una de dichas operaciones. La primera operación a discutir indica la forma en que debe ser creado un árbol *B*.

Creación de árboles *B*

Un árbol balanceado tiene como principal característica que todos los nodos terminales se encuentran a la misma distancia del nodo raíz. Para poder lograr esta construcción, no es posible concebir un árbol *B* de la misma forma que un árbol "tradicional". Cuando no se dispone de suficiente espacio y para preservar el principio de altura constante, es la raíz la que debe alejarse de los nodos terminales.

El proceso de creación comienza con una estructura vacía. Para permitir que el lector comprenda adecuadamente el proceso de creación, se resolverá simultáneamente el problema en forma gráfica, sobre un árbol, y también se presentará la generación del archivo de índices asociado.

A fines prácticos, las claves que compondrán el árbol *B* serán pares de letras y, además, el orden del árbol se establecerá en 4. Este orden del árbol tan pequeño se determina para permitir, en pocos pasos, presentar el mecanismo de creación.

En el momento inicial, el único nodo existente en el árbol será un nodo raíz, el cual no contendrá ningún elemento. Asimismo, el archivo del índice se encontrará vacío. La Figura 7.2 y la Tabla 7.1 (representación gráfica del registro del archivo índice) presentan esta situación.

FIGURA 7.2

Construcción gráfica del árbol

TABLA 7.1

Nodo Raiz: Nulo		
Punteros	*Datos*	*Nro Datos*

Cuando llega el primer elemento, `FG`, este se inserta en la primera posición libre del nodo raíz; este proceso se muestra en la Figura 7.3 y en la Tabla 7.2. El proceso detecta que, en el archivo de índice, el nodo raíz es nulo; por lo tanto, se inserta el primer registro del

archivo en el primer lugar de los datos, la clave ingresada FG, el número de lugares ocupados se establece en 1 y, al no tener este nodo raíz descendientes, los punteros a los hijos se establecen en nulo. Nótese que se utiliza –1 para indicar valor nulo; cuando una dirección es válida, debe registrar el nodo siguiente. El nodo siguiente es otro registro del archivo; por ese motivo se selecciona –1 como indicador de nulo. El primer registro del archivo está ubicado en la posición cero.

Figura 7.3

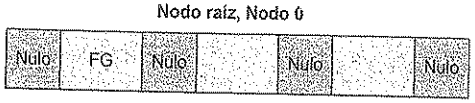


Tabla 7.2

	Nodo Raiz: 0						
	Punteros				Datos		Nro.Datos
0	-1	-1			FG		1

Los elementos siguientes comenzarán siempre el proceso de inserción a partir del nodo raíz. Si el nodo raíz tiene lugar, se procederá a insertarlos en este nodo teniendo en cuenta que los elementos de datos deben quedar ordenados dentro del nodo. Se debe notar que este proceso no resulta costoso, dado que el registro que contiene al nodo raíz está almacenado en memoria; por lo tanto, su ordenación solo debe contabilizar accesos a memoria RAM.

De esta forma, si llegan los elementos TR y AD, la Figura 7.4 y la Tabla 7.3 muestran cómo queda el índice construido.

Figura 7.4

Tabla 7.3

	Nodo Raiz: 0							
	Punteros				Datos			Nro Datos
0	-1	-1	-1	-1	AD	FG	TR	3

La llegada de un nuevo elemento provocará un *overflow* en el nodo. Este *overflow* significa que en el nodo no hay capacidad disponible para almacenar un nuevo elemento de datos.

Supóngase que la siguiente clave es ZT; cuando se intenta insertar la clave en el registro 0, se determina el *overflow*. Ante la ocurrencia de *overflow*, el proceso es el siguiente:

1. Se crea un nodo nuevo.
2. La primera mitad de las claves se mantienen en el nodo viejo.
3. La segunda mitad de las claves se trasladan al nodo nuevo.
4. La menor de las claves de la segunda mitad se promociona al nodo padre.

En el ejemplo planteado, se manipulan cuatro claves: AD, FG, TR, ZT. En este caso, el registro 0 mantendrá a AD y FG, en el registro 1 se dejará ZT, y TR se promocionará al nodo padre. Como el nodo dividido, el nodo cero, era hasta el momento la raíz del árbol, se debe generar un nuevo registro, el 2, que será considerado el nuevo nodo raíz. (Ver Figura 7.5 y Tabla 7.4). En la Tabla 7.4, se puede observar que la dirección del nuevo nodo raíz fue modificada y ahora apunta al nodo 2.

Figura 7.5

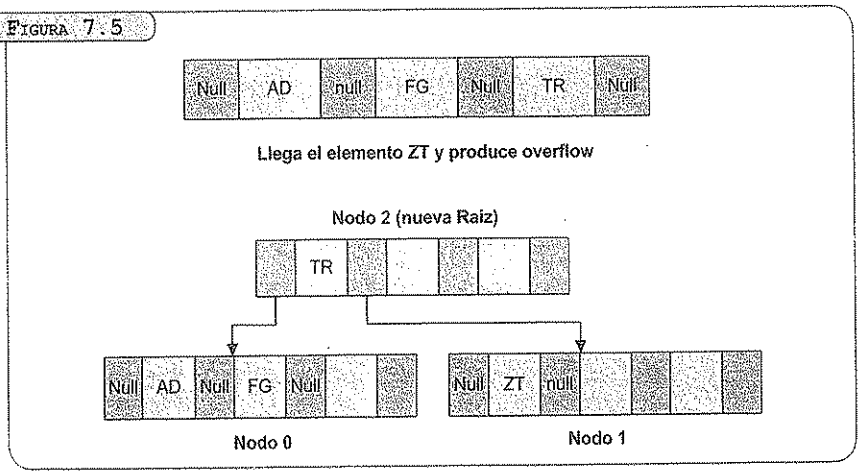


Tabla 7.4

	Nodo Raiz: 0							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	FG		2
1	-1	-1			ZT			1
2	0	1			TR			1

Siguiendo con el ejemplo, se intenta insertar la clave WA; algoritmicamente el proceso comienza desde el nodo raíz, el nodo 2. WA es mayor que la clave actual del nodo TR, la dirección a derecha del primer elemento del nodo es 1; esto significa que no es un nodo terminal, y como los elementos deben insertarse en un nodo terminal, se recupera desde memoria secundaria el nodo 1. Aquí se determina que la clave WA es menor que ZT, y como la dirección a izquierda de ZT es nula, se está ante un nodo terminal que tiene capacidad para almacenar WA. La clave es ubicada en dicho nodo. (Ver Figura 7.6 y Tabla 7.5).

Figura 7.6

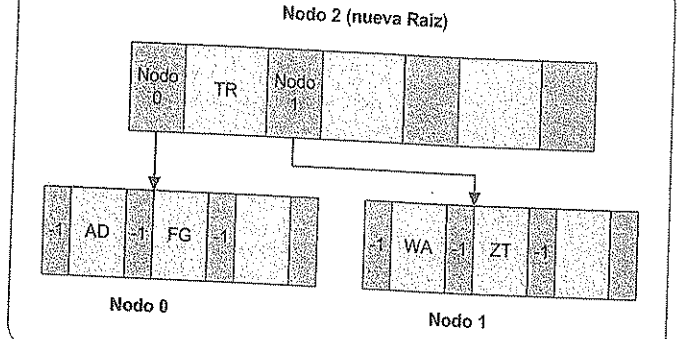


Tabla 7.5

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	FG		2
1	-1	-1	-1		WA	ZT		2
2	0	1			TR			1

Supóngase que ahora se deben insertar las claves BH y XF; procediendo igualmente que el paso anterior, cada elemento es insertado en el nodo 0 y el nodo 1, respectivamente, sin generar *overflow* sobre dichos nodos. La Figura 7.7 y la Tabla 7.6 dan cuenta de dicha situación.

Figura 7.7

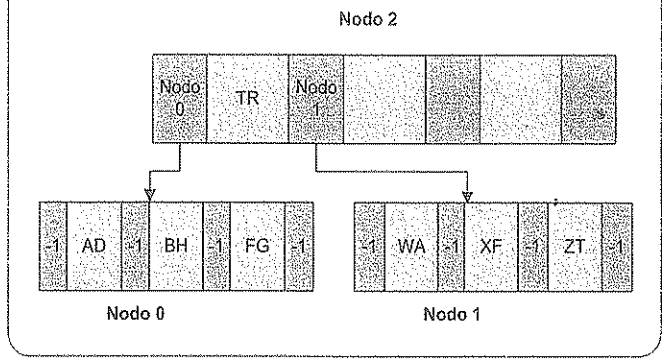


Tabla 7.6

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1	-1	AD	BH	FG	3
1	-1	-1	-1	-1	WA	XF	ZT	3
2	0	1			TR			1

La siguiente clave que se debe insertar en el árbol es MN; el proceso nuevamente comienza desde el nodo raíz y, desde este, se accede al nodo 0. El nodo 0 es un nodo terminal pero no dispone de capacidad para almacenar más elementos. Esta situación produce un *overflow* en el nodo 0, por lo que se debe generar un nuevo nodo, el 3 en este caso. El lector debe notar que las claves que producen *overflow* son AD, BH, FG y MN. En el nodo 0 quedarán las claves AD y BH, en el nodo 3 quedará MN, y FG será promocionada al nodo raíz junto con la dirección del nuevo nodo generado. En el nodo 2 se producirá un reacomodamiento para permitir que FG sea insertado en el orden respectivo. (Ver Figura 7.8 y Tabla 7.7).

Figura 7.8

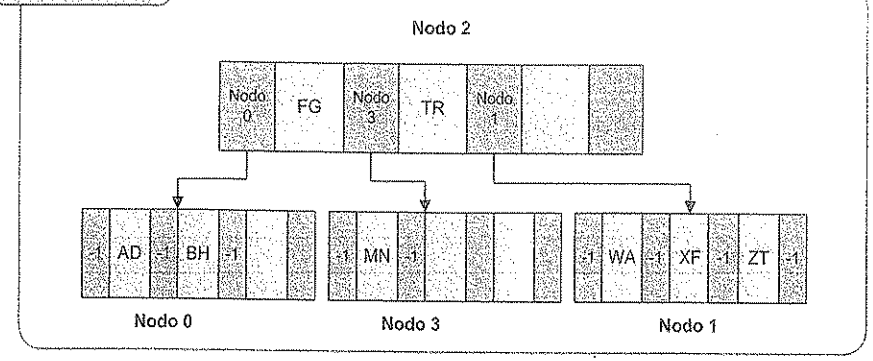


Tabla 7.7

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	BH		2
1	-1	-1	-1	-1	WA	XF	ZT	3
2	0	3	1		FG	TR		2
3	-1	-1			MN			1

La clave a insertar ahora es VU. Procediendo de la misma forma que en el paso anterior, se producirá *overflow* sobre el nodo 1. Se genera un nuevo nodo, el 4, y luego de realizar las divisiones respectivas quedarán las claves VU y WA en el nodo 1, ZT en el nodo 4 y XF se promocionará al nodo raíz. Esta situación se presenta en la Figura 7.9 y en la Tabla 7.8.

Figura 7.9

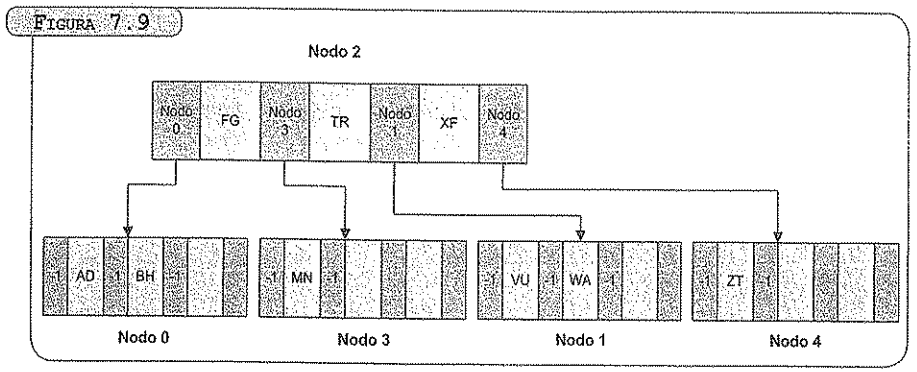


Tabla 7.8

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	BH		2
1	-1	-1	-1		VU	WA		2
2	0	3	1	4	FG	TR	XF	3
3	-1	-1			MN			1
4	-1	-1			ZT			1

Se deben introducir en el árbol las claves CD y UW. Las mismas son insertadas en los nodos 0 y 1, respectivamente, sin producir *overflow*. (Ver Figura 7.10 y Tabla 7.9).

Figura 7.10

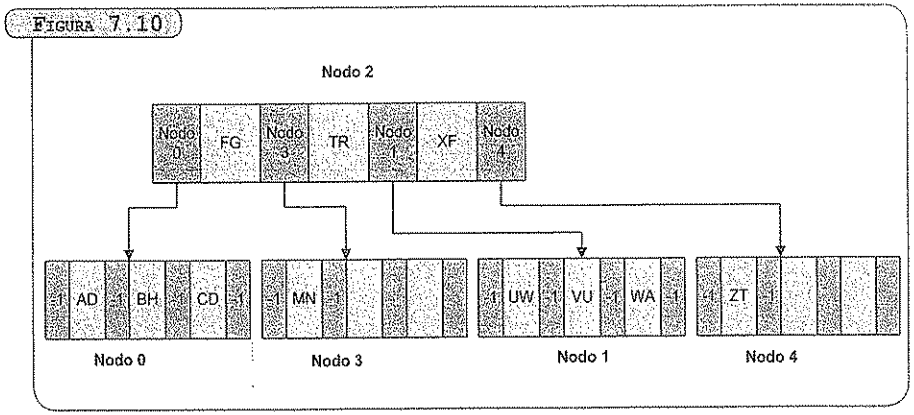


Tabla 7.9

	Nodo Raiz: 2							
	Punteros				Datos			Nro Datos
0	-1	-1	-1	-1	AD	BH	CD	3
1	-1	-1	-1	-1	UW	VU	WA	3
2	0	3	1	4	FG	TR	XF	3
3	-1	-1			MN			1
4	-1	-1			ZT			1

Por último, se inserta la clave DI en el árbol. Desde el nodo raíz se accede al nodo 0, donde debería insertarse la clave. Pero, en este nodo, nuevamente se produce *overflow*. Se crea, entonces, el nodo 5, las claves AD y BH quedan en el nodo 0, y DI pasa al nodo 5, promocionando al nodo padre la clave CD. Se debe notar que ahora en el nodo 2 se produce *overflow*. En este caso se procede nuevamente dividiendo dicho nodo, y se genera de esta forma el nodo 6. Las claves CD y FG quedan en el nodo 2; XF pasa al nodo 6; en tanto TR debe promocionarse al nodo raíz. Este nodo raíz debe generarse dado que hasta el momento el nodo 2 actuaba como tal. Se crea el nodo 7 (nueva raíz) que contendrá a la clave TR y a la dirección de sus nuevos hijos: el nodo 2 y el nodo 6. Obsérvese que, en la Tabla 7.10, se modifica la dirección del nodo raíz, siendo ahora el nodo 7.

El proceso continúa con la inserción de elementos en el árbol de la misma forma; cuando el árbol crece en altura, es la raíz la que se aleja de los nodos terminales, de modo tal que siempre el árbol crezca de manera balanceada. En todo momento, durante el proceso de creación del árbol, las propiedades descriptas anteriormente fueron respetadas.

En el presente libro no se describe el algoritmo que permite generar un árbol *B*. Este algoritmo se deja como ejercicio intelectual para el lector, considerando que se encuentra disponible en diversos sitios web, con versiones en diferentes lenguajes de programación.

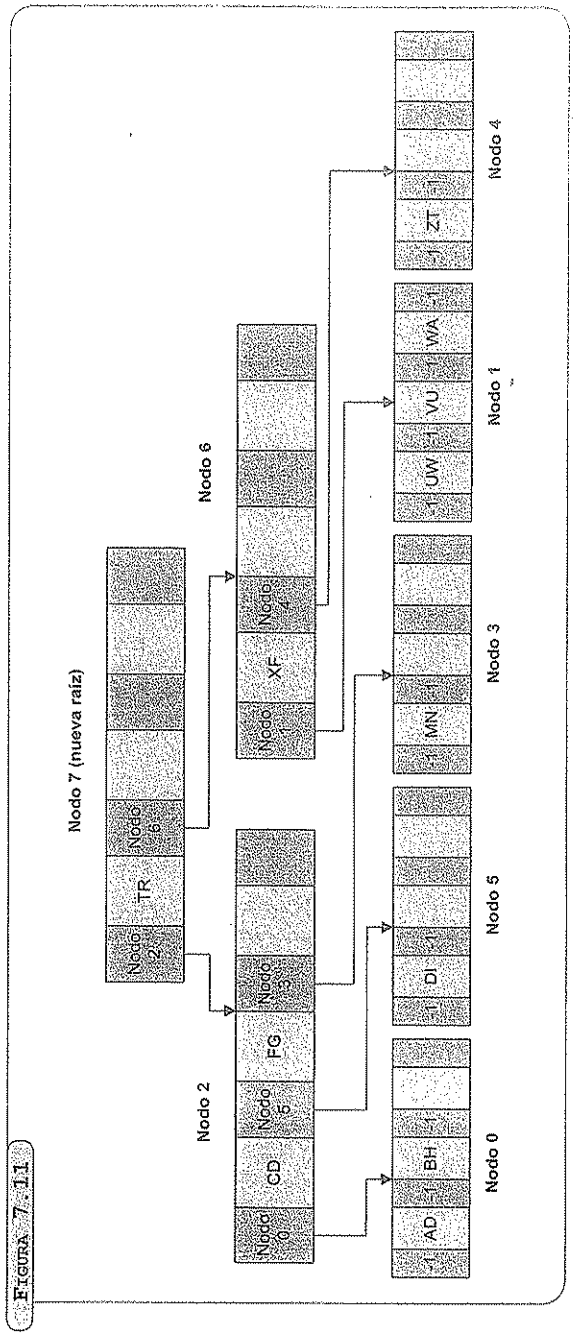


Figura 7.11

Tabla 7.10

	Nodo Raiz: 7							
	Punteros				Datos			Nro Datos
0	-1	-1	-1		AD	BH		2
1	-1	-1	-1	-1	UW	VU	WA	3
2	0	5	3		CD	FG		2
3	-1	-1			MN			1
4	-1	-1			ZT			1
5	-1	-1			DI			1
6	1	4			XF			1
7	2	6			TR			1

Búsqueda en un árbol *B*

En el apartado anterior, se describió mediante un ejemplo el mecanismo de inserción de elementos en un árbol *B*. El proceso de búsqueda coincide con la primera parte del proceso mostrado.

El primer paso de la inserción consiste en localizar el nodo que debería contener al nuevo elemento. El proceso de búsqueda realiza la misma operación; comenzando desde el nodo raíz, se procede a buscar el elemento en cuestión. En caso de encontrarlo en dicho nodo, se retorna una condición de éxito (esto implica retornar la dirección física en memoria secundaria, asociada al registro que contiene la clave encontrada). Si no se encuentra, se procede a buscar en el nodo inmediato siguiente que debería contener al elemento, procediendo de esa forma hasta encontrar dicho elemento, o hasta encontrar un nodo sin hijos que no incluya al elemento.

Tomando como ejemplos la Figura 7.11 y la Tabla 7.10, si se busca el elemento `DI`, comenzando desde el nodo 7 (nodo raíz), allí no se encuentra tal elemento y, siendo `DI` una clave menor que `TR`, se procede a recuperar el nodo inmediato izquierdo (el nodo 2). En dicho nodo, nuevamente no se encuentra la clave `DI`, y como esta es mayor que `CD` y menor que `FG`, se bifurca al nodo 5, donde se localiza la clave buscada, y, por lo tanto, el proceso retorna éxito.

Supóngase ahora que la clave buscada es `DR`; procediendo de la misma forma que en el párrafo anterior, se recupera el nodo 5 donde no se encuentra la clave buscada. Como `DR` es mayor que `DI`, se intenta recuperar el nodo derecho. El puntero indica que no hay más nodos disponibles. Por lo tanto, el elemento buscado no se encuentra en el árbol balanceado, y el proceso no retorna éxito dado que el elemento no está en el árbol.

Eficiencia de búsqueda en un árbol *B*

El proceso algorítmico de búsqueda de un elemento en un árbol balanceado no difiere demasiado del mismo proceso en un árbol binario común. Comienza la búsqueda en el nodo raíz y se va bifurcando hacia los nodos terminales en la medida en que el elemento no sea localizado.

La **eficiencia de búsqueda en un árbol *B*** consiste en contar los accesos al archivo de datos, que se requieren para localizar un elemento o para determinar que el elemento no se encuentra.

El resultado es un valor acotado en el rango entero [1, *H*], siendo *H* la altura del árbol tal como fuera definida previamente en el Capítulo 6. Si el elemento se encuentra ubicado en el nodo raíz, la cantidad de accesos requeridos es 1. En caso de localizar al elemento en un nodo terminal (o que el elemento no se encuentre), serán requeridos *H* accesos.

La cuestión por resolver, a fin de poder comparar la eficiencia de búsqueda contra los métodos discutidos en capítulos anteriores, consiste en poder comparar *H* (altura, o cantidad de niveles del árbol) con *N* (cantidad de registros en el árbol). Se debe recordar que, en casos anteriores, la eficiencia x se calculó siempre en función de los registros que contiene el archivo.

Para poder comparar resultados, es necesario medir la eficiencia en variables similares. Para ello, se debe acotar el valor de *H*. Una cota para *H* en función de los registros que el árbol contiene (*N*) permitirá realizar la comparación requerida.

Antes de comenzar con el análisis, se tendrá en cuenta, aunque no se demostrará en el presente libro, el siguiente axioma: "Un árbol *B* asociado a un archivo de datos que contiene *N* registros contendrá un total de *N*+1 punteros nulos en sus nodos terminales". A partir de la Figura 7.11 o de la Tabla 7.10, se deduce que la cantidad de registros almacenados en el archivo es 12, y que la suma de punteros nulos (–1) en los nodos terminales es 13. Este axioma permitirá a continuación acotar *H*.

Para poder realizar una cota, se debe analizar siempre la peor situación posible; de esa forma se garantiza que dicha cota se cumplirá siempre. Para definir la cota, se deben tomar en cuenta las propiedades básicas de un árbol *B*. Una de ellas indica que, en un árbol *B*, el nodo raíz tiene dos descendientes como mínimo o, en su defecto, ninguno. Si ocurriera este último caso, el árbol tendría un solo nivel, y por ende, con un acceso se recupera el elemento buscado o se

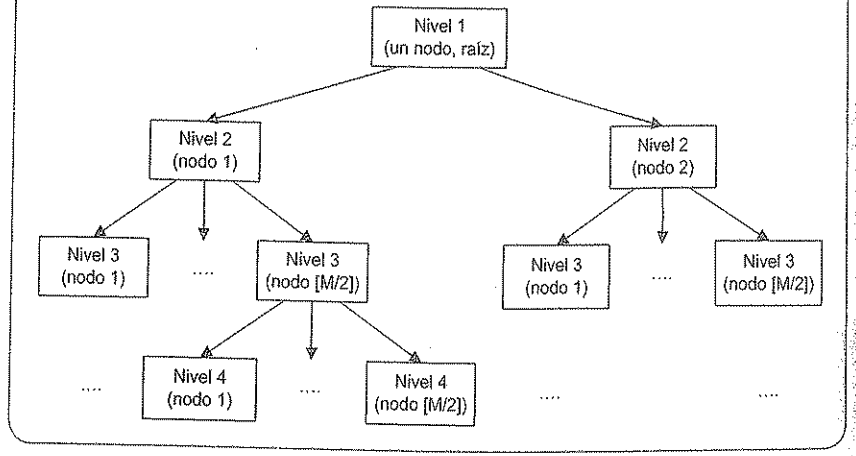
detecta su inexistencia. Ahora bien, el peor caso posible es tener un nodo raíz con dos descendientes (un solo elemento de datos en el nodo raíz). Esta es la peor situación posible, dado que con solo dos descendientes se está forzando la mayor cantidad de niveles posible, y esto permite establecer una cota.

Siguiendo este análisis y teniendo en cuenta otra de las propiedades del árbol *B*, un nodo que no es terminal ni raíz, es decir, un nodo interno, tiene un mínimo de [*M*/2] descendientes, siendo *M* el orden del árbol *B*.

Resumiendo lo anterior, la raíz tiene un mínimo de dos descendientes y cualquier nodo no terminal tiene un mínimo de [*M*/2] descendientes.

Por lo tanto, del primer nivel se dispone de dos nodos descendientes; del segundo nivel se dispone de [*M*/2] nodos descendientes por cada uno de los nodos de ese nivel (dos nodos); del tercer nivel, de [*M*/2] nodos descendientes por cada uno de los nodos de ese nivel (2 * [*M*/2] nodos), y así sucesivamente. (Ver Figura 7.12).

Figura 7.12



Generalizando (valores de cota)

Nivel 1 (raíz)	2 descendientes
Nivel 2	2 * [*M*/2] descendientes
Nivel 3	2 * [*M*/2] * [*M*/2] descendientes
...	
Nivel *H*	2 * ([*M*/2])$^{H-1}$ descendientes

Se debe tener en cuenta ahora el axioma presentado anteriormente: en el último nivel del árbol *B* (nodos terminales) existen en total *N*+1 punteros nulos, si *N* es la cantidad de registros que contiene el árbol.

Por lo tanto, la cantidad de punteros nulos es mayor que la cota de descendientes encontrada anteriormente:

$N+1 >= 2 * ([M/2])^{H-1}$

A partir de un simple análisis de la inecuación anterior, se acotará la cantidad de niveles del árbol (*H*) en función del orden de dicho árbol (*M*) y la cantidad de registros contenidos (*N*).

$(N+1) / 2 > ([M/2])^{H-1}$

Aplicando logaritmos a la fórmula anterior, se obtiene:

$Log_{[M/2]} ((N + 1) / 2) > H - 1$

Y luego:

$H < 1 + Log_{[M/2]} ((N + 1) / 2)$

Arribando así a la cota deseada.

Si se le asigna un valor posible al orden del árbol –por ejemplo, 512–, y se supone un árbol que contiene 1.000.000 de registros, en el peor caso su altura será:

$H < 3.34$

En este ejemplo, en el peor de los casos el árbol tendrá cuatro niveles, es decir que serán necesarios a lo sumo cuatro accesos para recuperar cualquier elemento. Si se compara este resultado estimativo con los presentados en capítulos anteriores, se puede observar una notable mejora en el proceso de búsqueda de información sobre un árbol *B*.

En el apartado siguiente, se analizará la eficiencia del proceso de inserción.

Eficiencia de inserción en un árbol *B*

Tomando como base el análisis previo, que permitió establecer una cota para la búsqueda de información en un árbol *B*, se procederá a analizar el caso de una inserción.

Se debe tener en cuenta que otra de las propiedades de los árboles *B* determina que todos los elementos de datos se insertan en los nodos terminales. Por ese motivo es necesario realizar *H* lecturas para poder encontrar el nodo donde el elemento será almacenado (comenzando en la raíz y avanzando hacia el nodo terminal).

Aquí pueden surgir dos alternativas. Si el nodo a realizar la operación dispone de lugar, la inserción del nuevo elemento no produce *overflow*, solo será necesaria una escritura en el nodo terminal con el nuevo elemento y de esta forma se da por finalizado el proceso de alta.

La alternativa consiste en que se produzca un *overflow* (se debe recordar cómo se trata esta situación, la cual fue explicada anteriormente). El peor caso es aquel donde el *overflow* se propaga hacia la raíz, haciendo aumentar en uno el nivel del árbol. Analizando detalladamente, si el nodo terminal se completa, entonces deberá realizarse un proceso de división, lo que motivará la escritura de dos nodos en el nivel terminal. Se propaga una clave hacia el nivel superior, la cual, como se considera la peor situación, también genera *overflow*; se debe dividir el nodo y realizar dos nuevas escrituras. Esto se repite hacia la raíz, la cual también se divide generando dos escrituras. Por último, se genera una nueva raíz. Todo este proceso genera dos escrituras por nivel y, además, aumenta en uno el nivel del árbol.

Resumiendo, la *performance* de la inserción en un árbol *B* está compuesta por:

H lecturas

1 escritura (mejor caso)

H lecturas

(2 * *H*) + 1 escrituras (peor caso)

Estudios empíricos realizados han demostrado que, generando árboles *B* de orden *M* = 10, la cantidad de *overflows* es de 25% aproximadamente. En tanto, si el orden se eleva a *M* = 100, la cantidad de *overflows* se reduce a 2%. Analizando esta última situación, 98 de cada 100 inserciones se tratan por el mejor caso, en tanto que solamente dos no están dentro de esta consideración, lo que no significa necesariamente que se trate del peor caso.

Por lo tanto, la inserción de nuevos elementos en el árbol *B* requiere un número considerablemente bajo de operaciones de entrada-salida para lograr mantener el orden en la estructura. Queda aún analizar el proceso de baja y su eficiencia.

Eliminación en árboles *B*

Se han tenido en cuenta hasta el momento las operaciones de búsqueda y creación/inserción sobre un árbol *B*. Estas dos operaciones, si bien pueden ser consideradas las más importantes desde un punto de vista de utilización (recuérdese que, en promedio, 80% de las operaciones sobre un archivo son de consultas y que, del 20% restante, la mayoría la representan operaciones de inserción), no son las únicas operaciones posibles.

Otra operación a tener presente consiste en el proceso de baja de información, tal cual como fuera discutido en capítulos anteriores. Más allá de trabajar utilizando borrados lógicos o físicos, o de las técnicas de recuperación de espacio que se podrían llevar adelante, en este apartado se discutirá en abstracto el proceso de baja sobre un árbol *B*.

Para poder avanzar al respecto, se debe tener en cuenta una consideración que se hereda desde los árboles binarios: para poder borrar un elemento, el mismo debe estar localizado en un nodo terminal. Respetando esta consideración, si en la Figura 7.11 se desea eliminar el elemento `TR`, el mismo deberá ser intercambiado por otro elemento residente en un nodo terminal, sin que esto afecte el recorrido del árbol generado. Entonces, algoritmicamente se deberá intercambiar `TR` con la mayor de sus claves menores (`MN`), o la menor de sus claves mayores (`UW`). Se ha demostrado empíricamente que la conveniencia en este reemplazo está dada por la menor de las claves mayores.

Así, el primer paso para eliminar `TR` será intercambiarlo con `UW`, quedando el árbol temporalmente como se observa en la Figura 7.13. Si bien no se muestra gráficamente, a continuación, `TR` se elimina del nodo terminal (situación a analizar en los párrafos siguientes).

Como ejercicio intelectual y tomando nuevamente la Figura 7.11, suponga ahora que desea eliminar el elemento `FG`, ¿cuál sería la opción de cambio? En este caso, se debería intercambiar `FG` con `MN`.

Como se presentó en el capítulo anterior, en un árbol binario, el proceso de baja finaliza liberando el espacio que el elemento ocupa e indicando su padre una dirección nula. La diferencia con un árbol balanceado radica en que un nodo terminal contiene un conjunto de elementos, entre ellos el que se debe borrar. Luego de quitar este elemento del nodo terminal, se deben seguir cumpliendo las propiedades antes descriptas para árboles *B*.

Existen entonces dos situaciones en el proceso de baja. La primera situación se presenta cuando, al borrar el elemento del nodo terminal,

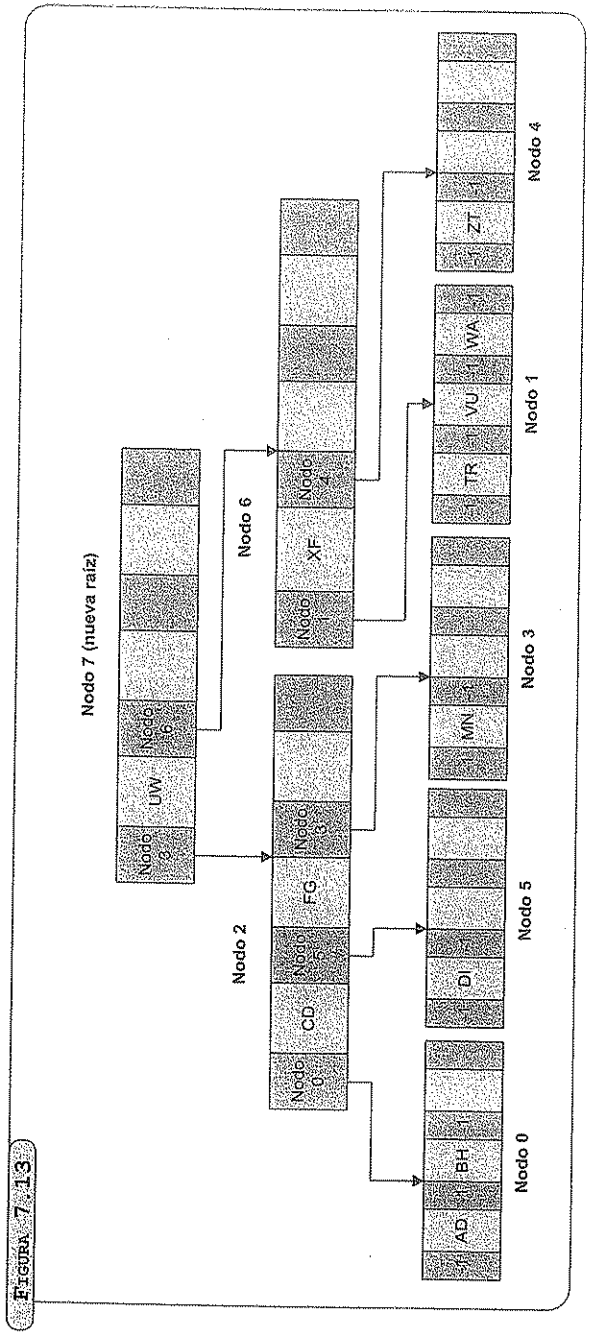
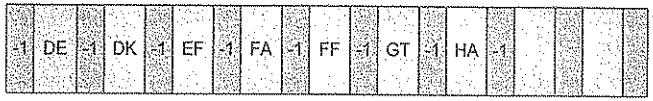


Figura 7.13

la cantidad de elementos restantes no está por debajo de la cantidad mínima ([*M*/2]–1). En este caso, no se genera un *underflow* en el nodo, y el proceso de baja finaliza.

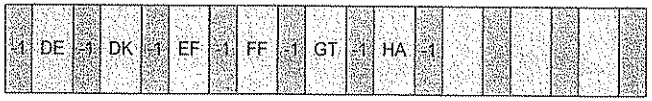
La Figura 7.14 a) representa un nodo terminal de un árbol balanceado de orden 10; sobre ese nodo se desea eliminar el elemento FA, el cual una vez eliminado deja al nodo como lo muestra la Figura 7.14 b). Con el orden 10 indicado y como el nodo contenía siete elementos, luego de borrar FA quedan seis elementos. Como se sigue cumpliendo la propiedad indicada, el proceso de baja finaliza.

Figura 7.14



Nodo Terminal

Figura 7.14 a)



Nodo Terminal

Figura 7.14 b)

La segunda situación posible es aquella que genera una situación de *underflow*. En este caso, el nodo al que se le quita un elemento deja de cumplir la condición de contener al menos [*M*/2] –1 elementos. A fin de poder ilustrar esta situación, es necesario presentar algunas definiciones extra.

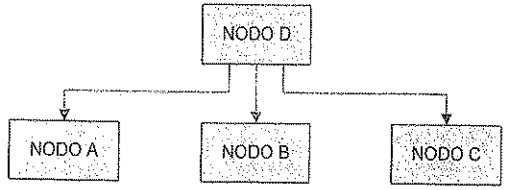
Se denomina **nodos hermanos** a aquellos nodos que tienen el mismo nodo padre.

Se denomina **nodos adyacentes hermanos o nodos hermanos adyacentes** a aquellos nodos que, siendo hermanos, son además dependientes de punteros consecutivos del padre.

La Figura 7.15 ilustra las definiciones anteriores. Los tres nodos hijos –*A*, *B* y *C*– dependen del mismo padre, *D*; por ende, son nodos hermanos. Ahora, el nodo *A* y el nodo *B* son apuntados por punteros

consecutivos, entonces son nodos adyacentes hermanos. Lo mismo ocurre entre *B* y *C*. Se debe notar que los nodos *A* y *C* solo son hermanos, puesto que, al no ser apuntados por punteros consecutivos en el padre, no pueden considerarse adyacentes.

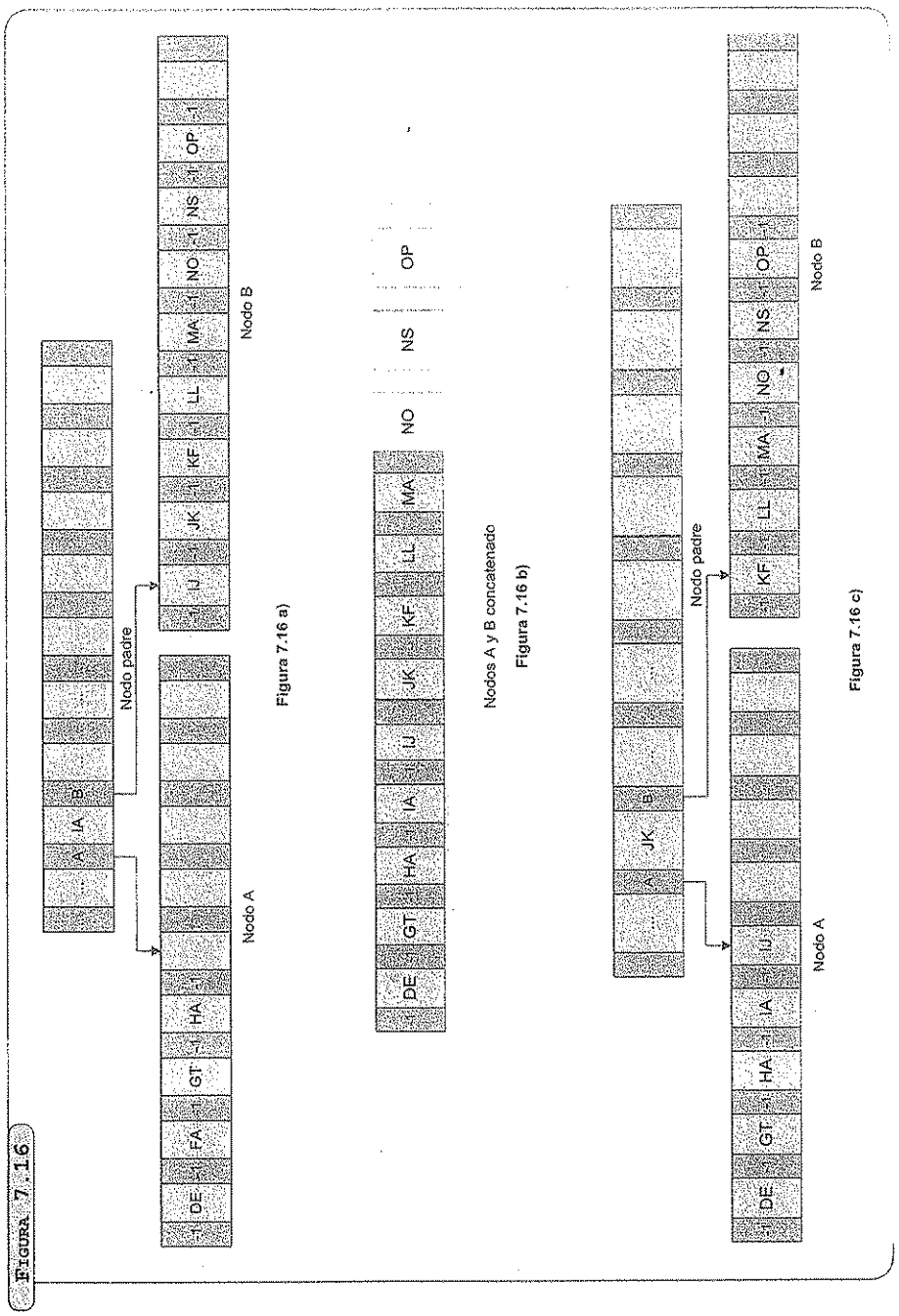
Figura 7.15



Retomando el proceso de baja, ya se mostró que la división de nodos garantiza el cumplimiento de las propiedades de un árbol *B* cuando se insertan nuevos elementos en caso de *overflow*. Ahora, se debe garantizar el mismo cumplimiento de estas propiedades cuando se eliminen las claves y se produzca *underflow* sobre un nodo. Aquí, las acciones que es posible llevar a cabo varían de acuerdo con las situaciones que el árbol plantee.

Supóngase la siguiente situación: en la Figura 7.16 a) se muestra un árbol de orden 10, sobre el nodo *A* se elimina el elemento FA, y como solamente quedan tres elementos en el nodo, no se cumple la propiedad con respecto a la cantidad de elementos mínimos de un nodo, dado que en este ejemplo la mínima cantidad necesaria de elementos es cuatro. Entonces se debe llevar a cabo algún tipo de acción para corregir esta situación. Todas las acciones que sea posible realizar deben afectar a la menor cantidad posible de nodos del árbol, con la finalidad de mantener acotados los cambios y, por ende, la eficiencia del proceso. Así, se debe trabajar con los nodos adyacentes hermanos para poder subsanar el problema.

Figura 7.16



La acción más inmediata que se puede plantear consiste en la opuesta a la división en un caso de *overflow*, la cual se puede denominar concatenación. Siguiendo este análisis, en la Figura 7.16 b), la opción consiste en concatenar el nodo *A* con el nodo *B*. Se debe notar que el nodo *B* contiene ocho elementos; por lo tanto, si se realiza la concatenación de ambos nodos, el resultado será un nodo que contendrá 12 elementos (tres del nodo *A*, ocho del nodo *B* y uno del nodo padre). Esta acción no es válida en este caso, ya que se superaría la máxima cantidad de elementos posibles para un nodo.

La alternativa, denominada redistribución, plantea utilizar algún nodo adyacente hermano del nodo conflictivo, permitiendo que dicho nodo adyacente hermano ceda elementos al nodo que presenta insuficiencia. Analizando la Figura 7.16 b), el nodo *B* dispone de ocho elementos; de aquí que es posible que le ceda al nodo *A* alguno de ellos. Este proceso significará que el nodo padre sea afectado. Se puede observar que si el elemento `IJ` pasara al nodo *A*, el elemento `IA`, contenido en el padre y utilizado como separador entre *A* y *B*, no sería de utilidad. Luego, como lo muestra la Figura 7.16 c), el proceso consiste en redistribuir entre los nodos *A* y *B* los elementos que conforman *A*, *B* y el padre. Así los cinco primeros (`DE, GT, HA, IA, IJ`) quedarán en el nodo *A*, los seis últimos quedarán en el nodo B (`KF, LL, MA, NO, NS, OP`) y el restante será ubicado en el padre (`JK`). Generalizando lo anterior: en una redistribución, la mitad de los elementos queda en un nodo, la otra mitad, en el segundo nodo, y se toma un elemento que actúe como separador y quede ubicado en el nodo padre.

Una ventaja que presenta la redistribución es que la cantidad de nodos no se ve afectada. Entonces, con solo cambiar de posición elementos entre tres nodos (dos hijos y el padre), el cambio sobre el árbol *B* queda acotado sin posibilidad de propagar cambios al resto de dicho árbol.

No obstante, la redistribución no siempre es posible. Para redistribuir se debe disponer de un nodo adyacente hermano con suficientes elementos para compartir. Obsérvese el caso planteado en la Figura 7.17, partiendo de la base del mismo árbol original de la Figura 7.16. Aquí se borrará nuevamente el elemento `FA` del nodo *A*, pero ahora el nodo *B* solo dispone de cuatro elementos. Al intentar redistribuir, el nodo *A* o el nodo *B* mantendrán su insuficiencia. En este caso, no es posible realizar una redistribución. La única operación viable resulta la concatenación, es decir, tomar los elementos del nodo *A*, juntarlos con los elementos del nodo *B* y traer a ese nuevo nodo el elemento del padre, dado que al juntar *A* y *B* no será necesario utilizar un separador. La Figura 7.17 b) plantea el caso descripto.

Se debe notar que la operación de concatenación puede propagar cambios a lo largo del árbol. En el ejemplo planteado, el nodo padre (que no es la raíz del árbol) deja de tener un elemento. Entonces puede ocurrir que el nodo padre genere insuficiencia. Este caso deberá ser resuelto como se planteara anteriormente. Si hubiera insuficiencia en el nodo padre, se procederá a redistribuir con algún adyacente hermano de este, y si no se pudiera realizar esta operación de redistribución, se aplicará la operación de concatenación. Así, la concatenación puede propagarse hacia arriba en el árbol *B*, obligando a modificar la raíz, disminuyendo en uno la cantidad de niveles del árbol.

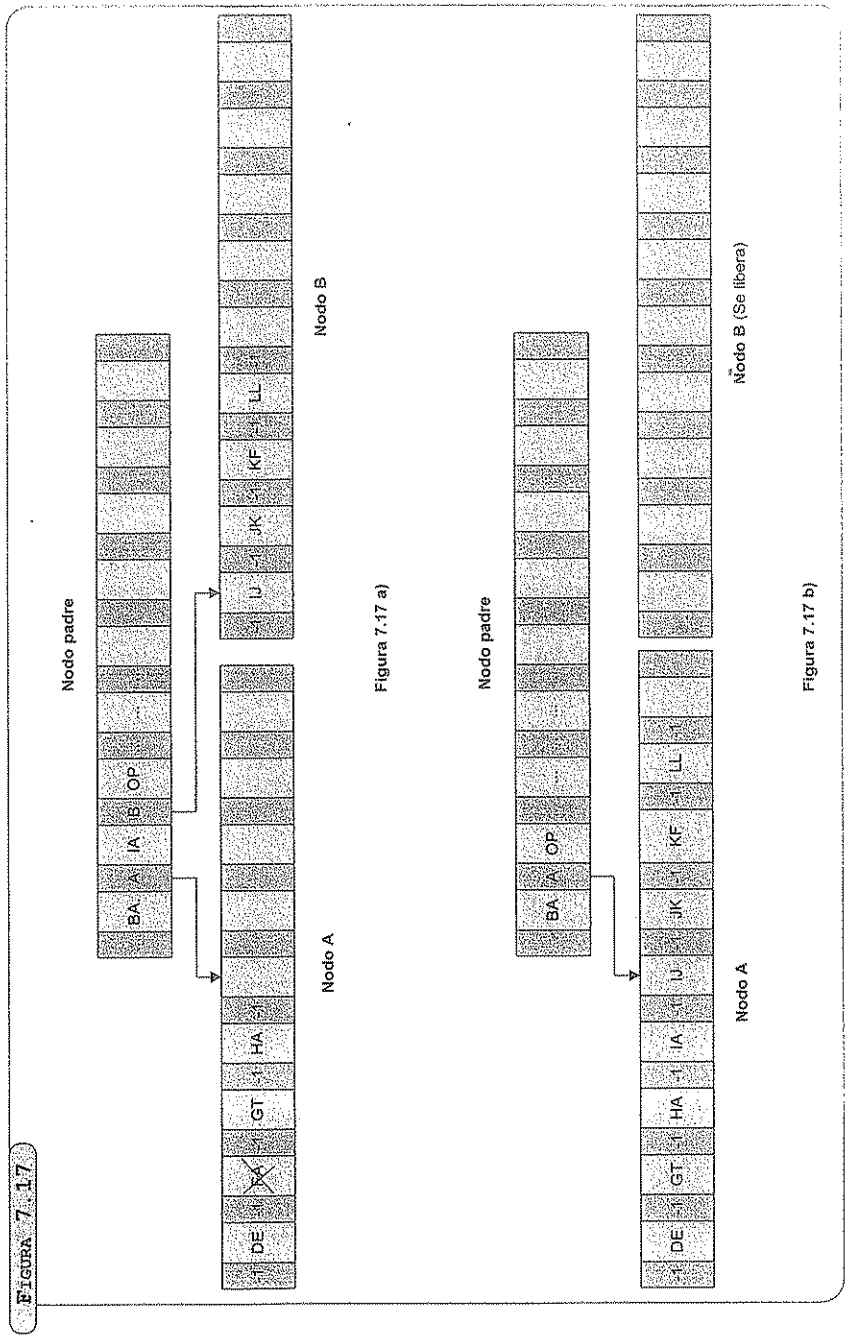


Figura 7.17

Eficiencia de eliminación en un árbol *B*

De la misma forma que se realizó el estudio de eficiencia en la inserción y la búsqueda, se presenta el análisis de eficiencia en la eliminación.

Aquí se puede comenzar por el mejor caso. Así, se intenta eliminar un elemento que está en un nodo terminal y cuyo borrado no genera insuficiencia. Entonces serán necesarias tantas lecturas como niveles tenga el árbol y una sola escritura (la correspondiente al nodo terminal sin el elemento borrado).

El peor caso quedará representado cuando la operación de borrado necesite concatenar (recuerde el lector que la redistribución acota los efectos a tres nodos). La concatenación implica leer un nodo adyacente hermano. Por cada nivel que se deba concatenar habrá dos lecturas, salvo en el nodo raíz, el cual carece de hermanos. La cantidad de escrituras se limita a una por nivel.

Resumiendo (siendo *H* la cantidad de niveles):

	Mejor caso	Peor caso
Cantidad de lecturas	*H*	$2 * H - 1$
Cantidad de escrituras	1	$H - 1$

A partir del análisis de eficiencia de la inserción, y teniendo en cuenta lo discutido en el Capítulo 4 sobre la cantidad de operaciones de baja que se registran sobre archivos, el estudio de eficiencia de la eliminación plantea una situación muy beneficiosa.

Modificación en árboles *B*

La restante operación que se debe analizar es la operación de modificación de un elemento contenido en un árbol *B*. Hasta aquí, se analizó la eficiencia de operaciones de inserción y eliminación, y se determinó que los resultados obtenidos son positivos.

La opción más simple para tratar un caso de modificación es proceder tomando un cambio como una baja del elemento anterior y un alta del nuevo elemento. Esta consideración no requiere mayor análisis de operatividad y tiene como resultado, en el análisis de *performance*, el obtenido de sumar una operación de baja seguida de una de alta. En el peor caso, dicha situación se mantiene acotada en el tiempo de respuesta.

Algunas conclusiones sobre árboles *B*

De acuerdo con lo planteado hasta el momento, los árboles balanceados representan una buena solución como estructura de datos, para implementar el manejo de índices asociados a archivos de datos.

Se debe notar que cualquier operación (consulta, alta, baja o modificación) se realiza en términos aceptables de *performance*. Se puede considerar, entonces, que este tipo de estructura representa una solución viable.

Se debe tener siempre en cuenta que las estructuras de árboles *B* serán utilizadas para administrar los índices asociados a claves primarias, candidatas o secundarias. Dichos archivos de índices contendrán la estructura planteada, clave, hijos y referencia del resto del registro en el archivo de datos original. El archivo de datos original se plantea como un archivo serie donde cada elemento se inserta siempre al final.

Como se definió en el Capítulo 5 para índices secundarios, tanto estos índices como aquellos que almacenan claves candidatas referencian a la clave primaria, en lugar de referenciar directamente a la posición física respectiva en el archivo de datos. Esto se debe a cuestiones de *performance*.

En caso de modificar la posición física de un registro, solo debe modificarse un índice, el correspondiente a la clave primaria.

De modificarse la clave primaria, deben modificarse el resto de los índices, pero como se explicará con detalle en el Capítulo 12, al diseñar un modelo de datos, en general las claves primarias se definen bajo la pauta de "nunca serán modificadas".

Árboles *B**

Los árboles *B** representan una variante sobre los árboles *B*. La consideración efectuada para avanzar sobre una estructura *B** parte del análisis realizado en el proceso de baja de información. Resumiendo este caso, el tratamiento de una saturación en un árbol *B* genera una sola operación, la división. El tratamiento de una insuficiencia genera dos operaciones, redistribución y/o concatenación. Claramente se puede observar que división y concatenación son operaciones opuestas. Entonces, el proceso de tratamiento de saturaciones en un árbol *B* no plantea una operación equivalente a la redistribución en el proceso de baja.

Precisamente, la algorítmica que planteara Knuth define una alternativa para los casos de *overflow*. Así, antes de dividir y generar nuevos nodos se dispone de una variante, redistribuir también ante una saturación. Esta acción demorará la generación de nuevos nodos y, por ende, tendrá el efecto de aumentar en forma más lenta la cantidad de niveles del árbol. Si la cantidad de niveles del árbol crece más lentamente, la *performance* final de la estructura es mejor.

Si se aplica el concepto de redistribuir, cuando un nodo se completa, reubica sus elementos utilizando un nodo adyacente hermano. Cuando no sea posible esta redistribución, se estará ante una situación donde tanto el nodo que genera *overflow* como su adyacente hermano están completos. Esto abre la posibilidad de dividir partiendo de dos nodos completos y generando tres nodos completos en dos terceras partes (2/3).

El aspecto más importante de la división planteada para un árbol *B** es que produce nodos completos en 2/3, en vez de nodos con solo la mitad de los elementos como planteaba un árbol *B*.

Un **árbol *B**** es un árbol balanceado con las siguientes propiedades especiales:

- Cada nodo del árbol puede contener, como máximo, *M* descendientes y *M*-1 elementos.
- La raíz no posee descendientes o tiene al menos dos.
- Un nodo con *x* descendientes contiene *x*-1 elementos.
- Los nodos terminales tienen, como mínimo, [(2*M*-1)/3] -1 elementos, y como máximo, *M*-1 elementos.
- Los nodos que no son terminales ni raíz tienen, como mínimo, [(2*M*-1) / 3] descendientes.
- Todos los nodos terminales se encuentran al mismo nivel.

Si se analizan las propiedades anteriores, se puede observar que cuando el árbol *B** tiene un solo nivel (la raíz) y este nivel se completa, no está disponible la posibilidad de redistribuir, dado que la raíz no tiene hermanos. Entonces, la propiedad que indica la mínima cantidad de elementos tiene una excepción de tratamiento, al dividir el nodo raíz y generar dos hijos. En este caso, los dos nodos hijos generados solamente completarán su espacio a la mitad. Esta situación se considera como la única excepción admitida para el tratamiento de las propiedades de árboles *B**.

La operación de búsqueda sobre un árbol *B** es similar a la presentada anteriormente para árboles *B*. La naturaleza de ambos árboles para localizar un elemento no presenta diferencias de tratamiento. La búsqueda comienza en el nodo raíz y se avanza hasta encontrar el elemento, o hasta llegar a un nodo terminal y no poder continuar con el proceso.

La operación de baja resulta nuevamente similar, tanto para los casos donde no se genera *underflow* como para aquellos donde sí se genera insuficiencia. En este último caso, la primera opción consiste en redistribuir, o en su defecto se deberá concatenar, basados en los principios definidos para árboles *B*. Si bien existen algunas variantes posibles, no es objetivo de este libro profundizar su tratamiento.

Operaciones de inserción sobre árboles *B**

La operación de inserción debe ser discutida con detalle. Las diferentes variantes de inserción tienen que ver, básicamente, con alternativas propuestas en los casos de redistribución.

El proceso de inserción en un árbol *B** puede ser regulado de acuerdo con tres políticas básicas: política de un lado, política de un lado u otro lado, política de un lado y otro lado.

Así, cada política determina, en caso de *overflow*, el nodo adyacente hermano a tener en cuenta. La política de un lado determina que el nodo adyacente hermano considerado será uno solo, definiendo la política de izquierda, o en su defecto, la política de derecha. En caso de completar un nodo, intenta redistribuir con el hermano indicado. En caso de no ser posible porque el hermano también está completo, tanto el nodo que genera *overflow* como dicho hermano son divididos de dos nodos llenos a tres nodos dos tercios llenos.

Figura 7.18

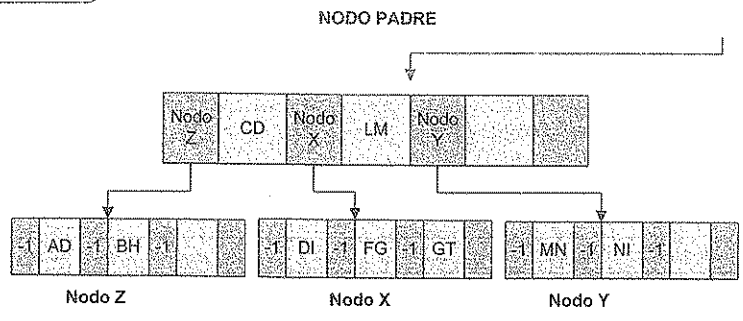


Figura 7.18 a)

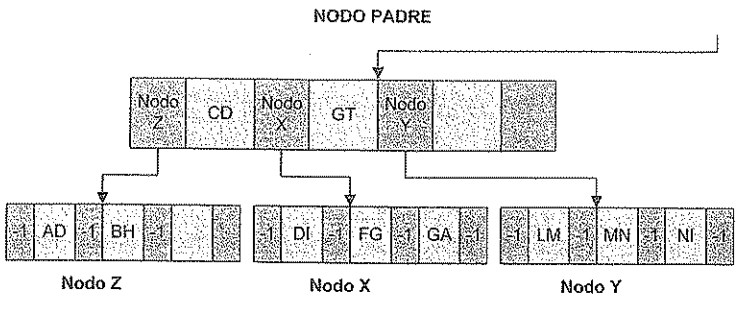
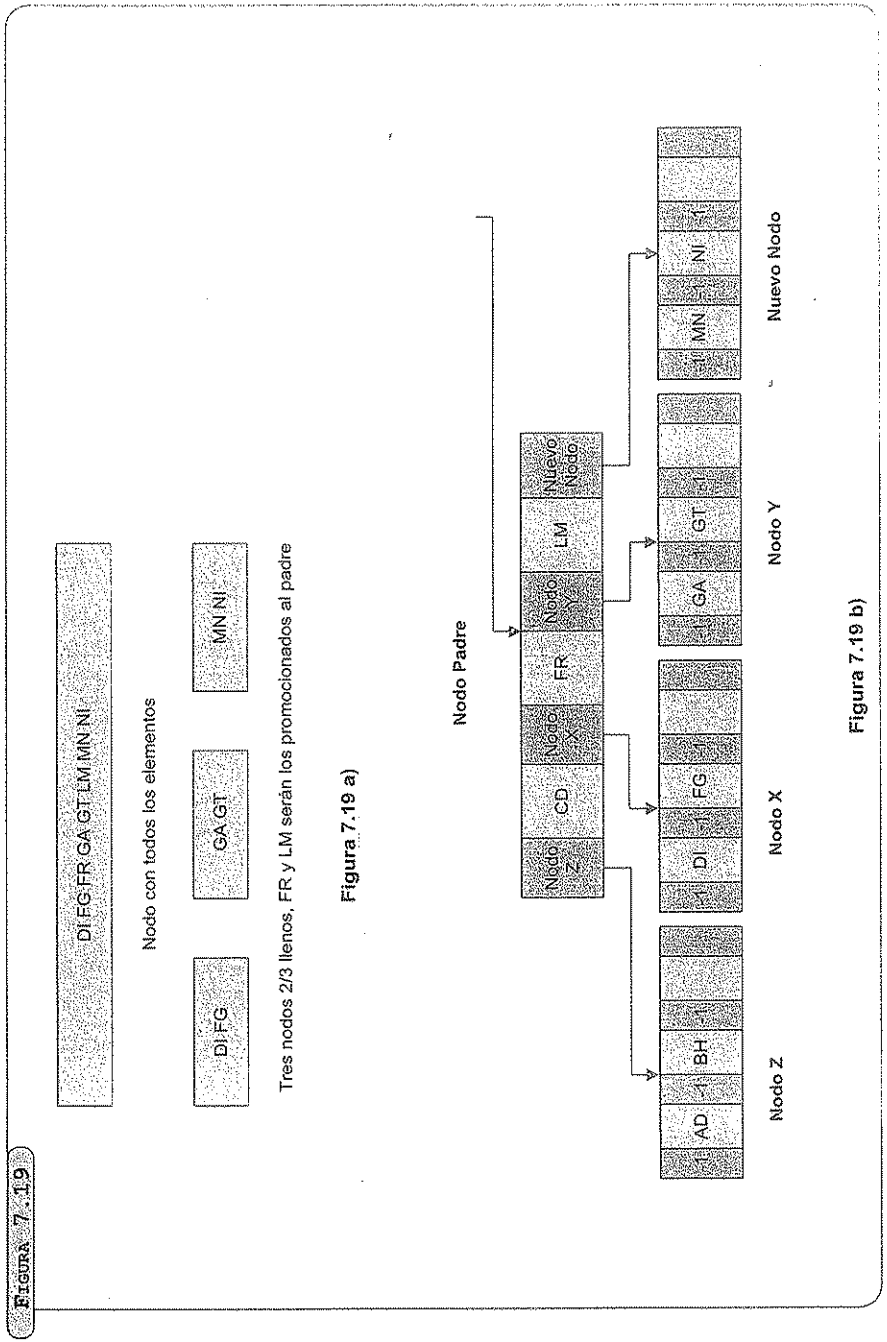


Figura 7.18 b)

La Figura 7.18 presenta un ejemplo resumido de la política de un lado, utilizando el hermano de la derecha. Aquí, teniendo un árbol *B** de orden 4, se puede observar que el nodo *X* está completo [Figura 7.18 a)] y se debe insertar el elemento GA; como dicho elemento no tiene lugar, se intenta redistribuir con el nodo adyacente hermano *Y*, como lo muestra la Figura 7.18 b). Se debe tener en cuenta que se han reacomodado los elementos de los nodos *X* e *Y*, así como el elemento separador en el nodo padre.

Utilizando el mismo ejemplo se pueden hacer varias consideraciones. Si llegase el elemento ZT, el nodo *Y* se encuentra completo, generando un *overflow*. El inconveniente que se genera está dado porque el nodo *Y* es el nodo ubicado más a la derecha del padre. Entonces, dicho nodo carece de hermano derecho. En esta situación, y solo en esta situación, es válido aplicar redistribución con el hermano izquierdo, pero no es posible aplicar dicha redistribución pues el nodo *X* está completo. En este caso, nuevamente tanto el nodo que genera *overflow* como dicho hermano son divididos de dos nodos llenos a tres nodos dos tercios llenos.

Figura 7.19



Otra situación para ser considerada es la siguiente: suponga que el elemento a insertar en el árbol de la Figura 7.18 b) es FR. El nodo *X* está completo, no se puede redistribuir con el nodo hermano *Y* pues también está completo y, si bien el nodo *Z* dispone de lugar para redistribuir, al tratarse de la política de un lado, derecha, algoritmicamente no se puede redistribuir con *Z*. Entonces, como lo muestra la Figura 7.19 a), a partir de los nodos *X*, *Y* y de los elementos separadores ubicados en el nodo padre, se genera temporalmente un nodo con suficiente capacidad para contener a todos los elementos. Este nodo se debe dividir en tres nodos con dos tercios llenos cada uno. Por último, a partir de la división mostrada en la Figura 7.19 a), el árbol queda como el presentado en la Figura 7.19 b).

La política de un hermano adyacente o el otro hermano adyacente representa una alternativa al caso anterior. Aquí, en caso de producirse una saturación en un nodo, se intenta primero redistribuir con un adyacente hermano. De no ser posible la redistribución, se intenta con el otro adyacente hermano. Si nuevamente la redistribución no es posible, la alternativa es dividir de dos nodos llenos a tres nodos dos tercios llenos. La Figura 7.20 a) expone, bajo el ejemplo definido anteriormente, la situación planteada; la política a utilizar es derecha o izquierda. En este caso, el nodo *X* contiene tres elementos y se debe insertar el elemento GA, se genera una saturación y, como la política planteada es primero derecha luego izquierda, se intenta redistribuir con el nodo *Y* (adyacente hermano de la derecha), como lo presenta la Figura 7.20 b). Suponga que luego se inserta el elemento TY; como el nodo *Y* tiene capacidad suficiente, no se genera saturación.

La Figura 7.20 c) muestra cómo queda el árbol *B** luego de introducir la clave EM; el nodo *X* produce saturación, el nodo *Y* no acepta redistribuir y, entonces, se redistribuye con el nodo *Z* (adyacente hermano de la izquierda).

Suponga ahora que se debe insertar DZ; los tres nodos terminales *X*, *Y* y *Z* están completos. Por lo tanto, no es posible redistribuir. Esta política determina que se debe tomar el nodo *X* y uno de sus adyacentes hermanos y proceder con la división. Siguiendo la política de derecha o izquierda presentada en este problema, se realizaría una división como se explicó en la Figura 7.19, utilizando los nodos *X* e *Y*.

La última política alternativa plantea ambos adyacentes hermanos. Aquí, la forma de trabajo es similar al caso anterior. Primero, redistribuir hacia un lado y, si no es posible, con el otro hermano. La diferencia aparece cuando los tres nodos están completos. Aquí se toman los tres nodos y se generan cuatro nodos con tres cuartas partes completas cada uno. Si bien esta alternativa completa más cada nodo del

Figura 7.20

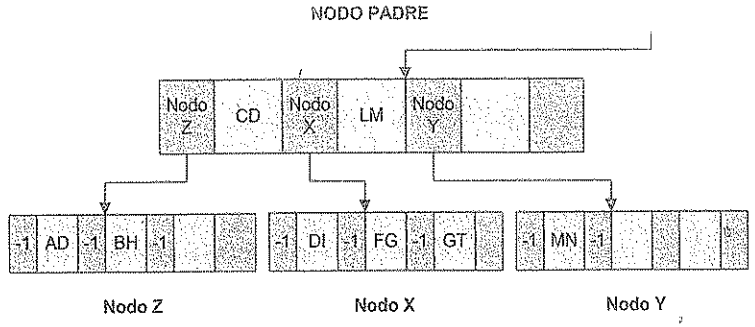


Figura 7.20 a)

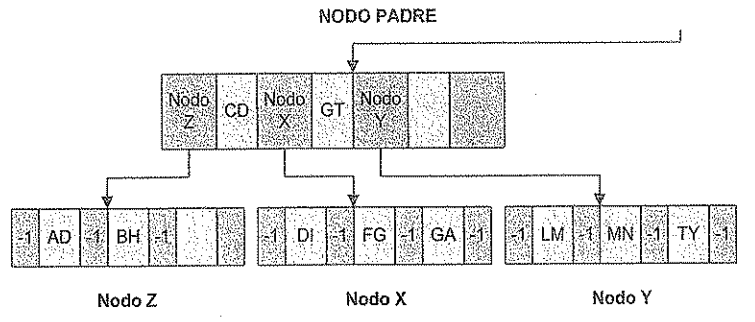


Figura 7.20 b)

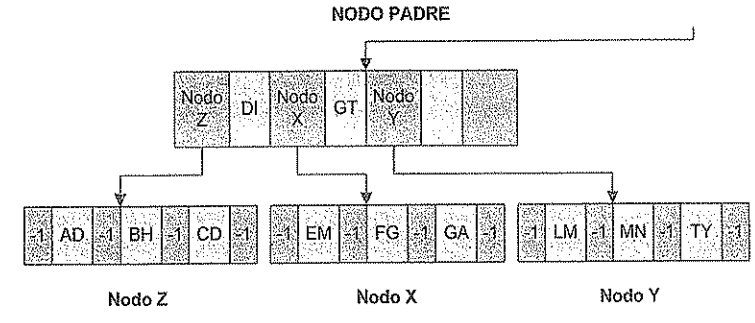


Figura 7.20 c)

árbol, y por consiguiente, genera árboles de menor altura, necesita de mayor número de operaciones de entrada-salida sobre disco para poder ser implementada. No es objetivo del presente material profundizar al respecto.

Análisis de *performance* de inserción en árboles *B**

La *performance* resultante de la inserción sobre árboles *B** dependerá de cada política. Así, ante la ocurrencia de *overflow*, como mínimo cada una de las políticas requiere dos lecturas (el nodo que se satura y un adyacente hermano) y tres escrituras (cada nodo hijo y el padre). Se debe considerar que no se contabiliza la lectura en el nodo padre, ya que una inserción se realiza sobre nodos terminales y para acceder a los mismos necesariamente se hace a través del padre.

En caso de necesitar realizar una división, la política de un lado necesita a su vez realizar cuatro escrituras (el nodo padre, los dos nodos que estaban completos y el nuevo que se genera). Es la misma situación generada por la política de un lado o el otro lado, dado que en este caso se divide nuevamente de dos nodos completos a tres nodos.

Por último, la política de un lado y el otro lado genera cuatro escrituras, dado que, además de involucrar a los tres nodos terminales, se deberán tener en cuenta el nodo padre y el nuevo generado.

Manejo de *buffers*. Árboles *B* virtuales

Como se discutió hasta el momento, los árboles *B* representan una muy eficiente estructura de almacenamiento para índices. El análisis de *performance* de las operaciones determina que el resultado obtenido depende directamente de la altura del árbol. Se ha mostrado que es posible generar árboles que contengan un número importante de elementos manteniendo la altura acotada.

Se debe tener en cuenta, además, que si un árbol *B* tiene *H* niveles, esto no significa que se necesite esa misma cantidad de accesos para recuperar un elemento.

Cuando se decide la capacidad de cada nodo, esta capacidad está determinada básicamente por la cantidad de información que podrá manejar el SO, a través del uso de *buffers*. Así, el concepto de *buffer* y el de nodo están íntegramente relacionados.

Se debe tener en cuenta que un SO administra un número importante de *buffers* y que, de acuerdo con la política de trabajo, es posible que

aquellos *buffers* más utilizados se mantengan en memoria principal, a fin de minimizar el número de accesos a disco. Una de las políticas más utilizadas se conoce como LRU, que, por sus siglas en inglés, significa "último más recientemente utilizado". Con esta política se mantienen en memoria principal los nodos que más se han visitado en forma reciente.

Un árbol *B* que administre un índice de acceso muy frecuente significará para el SO que el *buffer* que contenga el nodo raíz sea muy utilizado. Entonces, el SO tenderá a mantener dicho *buffer* en memoria, disminuyendo en uno el número de accesos a disco, requerido para recuperar la información.

En general, empíricamente es posible afirmar que si se dispone de un árbol de tres niveles, la primera búsqueda de un elemento demandará a lo sumo tres accesos a disco; la segunda búsqueda demandará a lo sumo dos accesos a disco (en este caso, la raíz ya se mantiene en memoria principal). Ahora, buscar cinco elementos requerirá en promedio 1.71 accesos al disco, disminuyendo la cantidad de accesos requeridos a medida que se intente localizar mayor información.

Esta última apreciación, si bien es solamente un análisis empírico, muestra una característica extra para los árboles *B* que hace más eficiente su utilización.

Acceso secuencial indizado

Se denomina **archivo con acceso secuencial indizado** a aquel que permite dos formas para visualizar la información:

1. **Indizada:** el archivo puede verse como un conjunto de registros ordenados por una clave o llave.
2. **Secuencial:** se puede acceder secuencialmente al archivo, con registros físicamente contiguos y ordenados nuevamente por una clave o llave.

Este nuevo tratamiento propuesto para archivos intenta compatibilizar el orden físico de los elementos con un acceso indizado, de acuerdo con lo definido para árboles *B*.

Suponga el lector que se dispone de un archivo con *N* registros que fueron organizados físicamente por orden de llegada, pero que se creó una estructura de índice utilizando un árbol *B* o *B** que permite la localización rápida de la información. Si se desea un tratamiento

secuencial de los elementos del archivo utilizando el orden de la llave, la única forma de extraerlos en orden es utilizando el índice (árbol *B*). Para recuperar los *N* registros en orden, es necesario recorrer todos los nodos del árbol a través de los punteros definidos. Esto significa acceder a un nodo terminal, volver a su padre y acceder al nodo terminal siguiente, repitiendo este proceso para todo el árbol. Esto resulta en un algoritmo mucho menos eficiente que tener la misma estructura ordenada físicamente.

Si el archivo requiere mucho acceso secuencial utilizando el índice (almacenado en un árbol *B*, por ejemplo, para obtener algún tipo de listado o presentar la información en pantalla), la solución resulta muy ineficiente.

Por otro lado, si se plantea la alternativa de mantener el archivo físicamente ordenado, se soluciona el problema de acceso anterior pero resulta inaceptable, al momento de acceder para recuperar un dato, realizar una inserción o una eliminación.

Estos problemas necesitan una solución alternativa que permita compatibilizar ambos accesos.

Archivos físicamente ordenados a bajo costo

El problema definido tiene claramente dos partes establecidas: el acceso secuencial indizado (árbol) y el orden físico de los elementos. Hasta el momento, en el capítulo se han discutido opciones para resolver el acceso de manera eficiente. Este apartado presentará una alternativa de orden lógico secuencial a bajo costo.

En el Capítulo 5, se analizó el costo del ordenamiento físico de un archivo y el mismo resultó extremadamente ineficiente. Por este motivo, la solución a los archivos físicamente ordenados tiene que provenir de su creación y administración, sin forzar el reordenamiento ante cada operación de inserción.

Como se discutió anteriormente, el nodo pasa a ser la unidad de entrada-salida, es decir, todos los elementos de un nodo son leídos consecutivamente y el nodo luego es almacenado en un *buffer*, o viceversa, se toman los elementos del nodo de un *buffer* y se escriben dichos elementos consecutivamente a disco. Por este motivo, cada vez que se accede a un nodo, debe contabilizarse un acceso a disco, independientemente de la cantidad de registros contenidos en él.

La Figura 7.21 a) muestra el contenido de un nodo o bloque, con capacidad para cinco registros, donde se almacenan los nombres de cinco alumnos de una cátedra de la Facultad de Informática. Supóngase

ahora que dicho archivo tiene dos nodos; como lo muestra la Figura 7.21 b), para poder mantener el orden entre los elementos, ambos nodos deben estar enlazados.

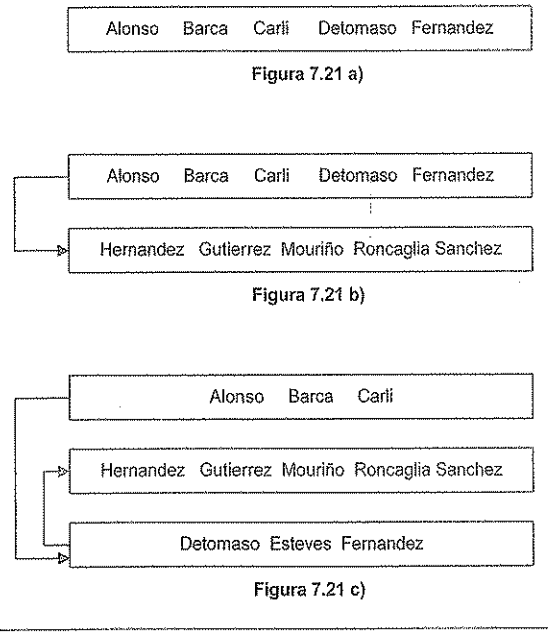
La pregunta a realizar es: ¿cómo se llegó a tener el archivo ordenado?, o ¿qué sucede si llega una clave nueva, por ejemplo, `Estevez`, a insertar en el primer nodo y no tiene espacio?

La respuesta a la pregunta es tratar cada nodo o bloque como si fuera un nodo terminal de un árbol *B*. Así, al llegar Estevez, debería insertarse entre Detomaso y Fernandez, para mantener el orden. Al no tener capacidad suficiente, el nodo debe ser dividido, de acuerdo con lo planteado en la algorítmica de árboles *B*.

La Figura 7.21 c) presenta dicha situación; se crea un nuevo nodo, donde se redistribuyen los elementos. Para mantener el orden del archivo, se modifican los enlaces entre los nodos.

Se debe notar que la *performance* de esta operación se limita a dos escrituras, el nodo existente modificado y el nuevo nodo, similar a la *performance* de una división con árboles *B*.

Figura 7.21



La eliminación de registros dentro del archivo puede ocasionar que un nodo tenga "pocos" elementos. Nuevamente, si se piensa en la operación de baja sobre un árbol *B*, un nodo que presenta insuficiencia puede ser concatenado o fusionado con el nodo hermano adyacente.

Esta solución presenta una ventaja concreta. Si bien el archivo no se encuentra físicamente ordenado, cada bloque sí lo está. Y, además, como cada nodo se encuentra enlazado con el siguiente, es posible realizar un recorrido secuencial ordenado a muy bajo costo en términos de accesos a disco.

El último análisis a realizar tiene que ver con la elección del tamaño del nodo. Nuevamente, dicho tamaño dependerá de la capacidad del *buffer* de entrada-salida del SO, y de la información que se desea almacenar para cada registro (la llave o clave más la dirección física del resto de la información del registro en el archivo de datos).

Árboles *B+*

Se ha discutido una solución de bajo costo, para recuperar los elementos en forma ordenada de un archivo, sin necesidad de reacomodamientos físicos costosos.

Ahora se debe encontrar un mecanismo que permita localizar los datos contenidos en los nodos, a bajo costo. La solución que presentaban los árboles *B* y *B** debería poder aplicarse en este entorno.

La estructura intermedia resultante se denomina árbol *B+* e incorpora las características discutidas para árboles *B*, además del tratamiento secuencial ordenado del archivo. Así, se podrán realizar búsquedas aleatorias rápidas de información, en conjunto con acceso secuencial eficiente.

Un **árbol *B+*** es un árbol multicamino con las siguientes propiedades:

- Cada nodo del árbol puede contener, como máximo, *M* descendientes y *M*-1 elementos.
- La raíz no posee descendientes o tiene al menos dos.
- Un nodo con *x* descendientes contiene *x*-1 elementos.
- Los nodos terminales tienen, como mínimo, ([*M*/2] – 1) elementos, y como máximo, *M*-1 elementos.
- Los nodos que no son terminales ni raíz tienen, como mínimo, [*M*/2] descendientes.
- Todos los nodos terminales se encuentran al mismo nivel.
- Los nodos terminales representan un conjunto de datos y son enlazados entre ellos.

Si se compara la definición anterior con la resultante para árboles *B*, se encontrarán similitudes, excepto en la última propiedad.

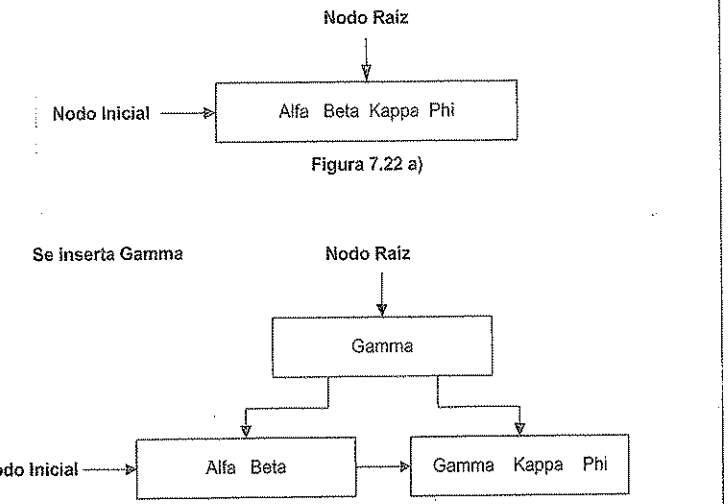
Es esta última propiedad la que establece la principal diferencia entre un árbol *B* y un árbol *B+*. Para poder realizar acceso secuencial ordenado a todos los registros del archivo, es necesario que cada elemento (clave asociada a un registro de datos) aparezca almacenado en un nodo terminal. Así, los árboles *B+* diferencian los elementos que constituyen datos de aquellos que son separadores.

Al comenzar la creación de un árbol *B+*, el único nodo disponible, el nodo raíz, actúa tanto como punto de partida para búsquedas como para acceso secuencial [Figura 7.22 a)].

En el momento en que el nodo se satura (en la figura, el árbol se supone de orden 5), se produce una división. Se genera un nuevo nodo terminal donde se redistribuyen los elementos.

Como lo muestra la Figura 7.22 b), ahora se dispone de tres nodos, los nodos terminales contendrán todos los elementos y el nodo raíz contendrá el elemento que actúa como separador. Se debe notar que la clave utilizada como separador es la misma que está contenida en el nodo terminal, es decir, se utiliza una copia del elemento y no el elemento en sí.

Figura 7.22



El proceso de creación del árbol *B+* sigue los lineamientos discutidos anteriormente para árboles *B*.

Los elementos siempre se insertan en nodos terminales. Si se produce una saturación, el nodo se divide y se promociona una copia (aquí está la diferencia) del menor de los elementos mayores, hacia el nodo padre. Si el padre no tiene espacio para contenerlo, se dividirá nuevamente.

Se debe notar que, en caso de dividir un nodo no terminal, se debe promocionar hacia el padre el elemento en sí y no una copia del mismo, es decir, solo ante la división de un nodo terminal se debe promocionar una copia.

Para borrar un elemento de un árbol *B+*, siempre se borra de un nodo terminal, y si hubiese una copia de ese elemento en un nodo no terminal, esta copia se mantendría porque sigue actuando como separador.

Árboles *B+* de prefijos simples

El agregado de la opción `prefijos simples` a un árbol *B+* intenta aprovechar mejor el uso de espacio físico.

Se puede observar en la Figura 7.22 b) que al producirse una división se ha promocionado la menor de las claves mayores, Gamma. También, puede observarse que en este caso no es necesario promocionar Gamma, ya que con la letra G alcanzaría para actuar como separador.

Un **árbol *B+* de prefijos simples** es un árbol *B+* donde los separadores están representados por la mínima expresión posible de la clave, que permita decidir si la búsqueda se realiza hacia la izquierda o hacia la derecha.

Las Figuras 7.23 a) y b) presentan dos ejemplos de división donde el árbol tiene la política de prefijos simples. En un caso, alcanza con la letra G como separador, pero en el otro es necesario que el separador sea Gon. El separador siempre tiene la menor cantidad de caracteres posible, de modo de poder actuar como tal entre los nodos hijos.

Figura 7.23

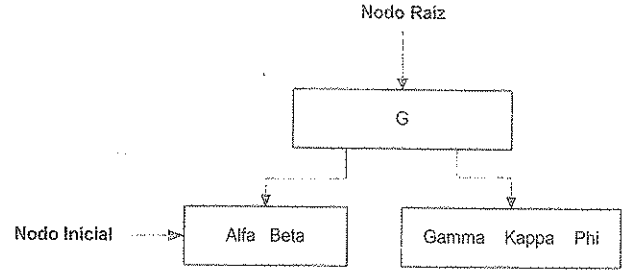


Figura 7.23 a)

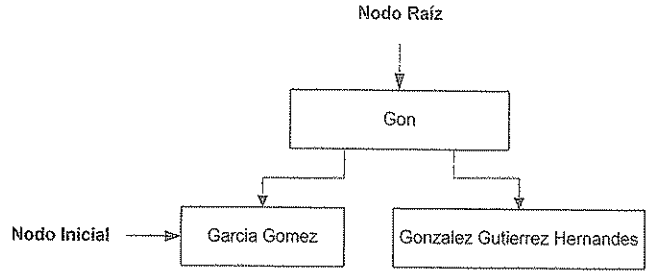


Figura 7.23 b)

Árboles balanceados. Conclusiones

La familia de los árboles balanceados son estructuras muy poderosas y flexibles para la administración de índices asociados a archivos de datos, con un nivel de *performance* muy interesante. Sin embargo, no deben considerarse como la única solución posible a todos los problemas.

La mejor estructura para un archivo va a depender del archivo en sí y del propósito de uso de dicho archivo. Es decir, si el archivo de datos se limita a unos pocos registros que pueden estar contenidos en un nodo, utilizar un árbol no es necesario, dado que agrega un nivel de indirección por cada índice generado. Tampoco es necesario si el archivo no requiere consultarse con frecuencia.

Como se discutirá más adelante, hay metodologías para organizar archivos que presentan un mejor tiempo de respuesta. Entonces, ante determinadas necesidades de *performance*, puede ocurrir que los árboles balanceados no cubran las expectativas de un problema particular.

Sin embargo, la familia de árboles *B* tiene una gran aplicabilidad en situaciones en las que resulta necesario acceder a un archivo tanto para búsquedas aleatorias eficientes como para acceso secuencial.

Las características compartidas entre los árboles de la familia *B* son las siguientes:

- Manejo de bloques o nodos de trabajo; esto facilita las operaciones de entrada-salida y mantiene acotado el número de accesos para realizar dichas operaciones.
- Todos los nodos terminales están a la misma distancia del nodo raíz, es decir, estos árboles están balanceados.
- Crecen de abajo hacia arriba, a diferencia de los árboles binarios comunes.
- Son "bajos" y "anchos", a diferencia de los binarios, que son "altos" y "delgados".
- Es posible implementar algoritmos que manipulen registros de longitud variable para, de esta forma, mejorar aun más la *performance* y el uso del espacio.

En particular, los árboles *B** mejoran la eficiencia de los árboles *B*, aumentando la cantidad mínima de elementos en cada nodo y, por consiguiente, ayudando a disminuir la altura final del árbol. El costo que se debe pagar en este caso es que las operaciones de inserción resultan un poco más lentas.

Por último, en los árboles *B+*, toda la información de las claves está contenida en los nodos terminales. Los nodos no terminales actúan como separadores, y poseen una copia de los elementos contenidos en los nodos terminales.

Cuestionario del capítulo

1. ¿Qué características principales presenta una estructura de árbol balanceada sobre un árbol binario?
2. ¿Por qué es importante el concepto de orden en un árbol multicamino? ¿La respuesta anterior es la misma si el árbol es multicamino balanceado?
3. ¿Por qué la cantidad de niveles de un árbol es importante para el estudio de la *performance* de las operaciones de búsqueda, inserción y borrado?
4. ¿Cómo afecta el orden del árbol a la cantidad de niveles de dicho árbol?
5. ¿Qué características diferentes presenta un árbol *B** respecto de un árbol *B*?
6. ¿Cómo mejora la *performance* de búsqueda en un árbol *B** respecto de un árbol *B*?
7. ¿Cuáles son los costos de generar una estructura *B** en lugar de una estructura *B*?
8. ¿Qué aspectos hacen necesario utilizar árboles *B+*?
9. ¿Qué se puede decir de la *performance* de búsqueda en un árbol B+ respecto de las estructuras *B** o *B* común?
10. ¿Qué significa utilizar un árbol *B+* de prefijos simples?

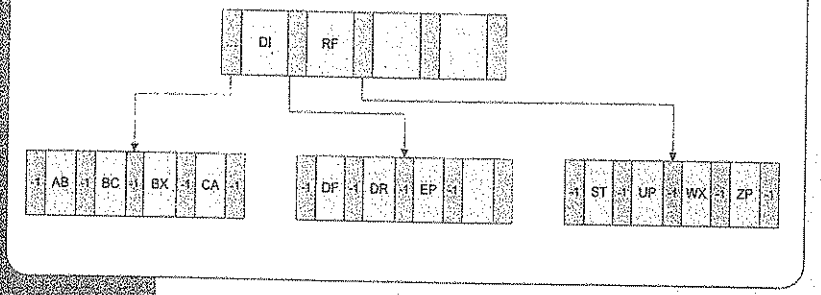
Ejercitación

1. Dado el siguiente conjunto de claves, genere un árbol *B* de orden 5 y muestre cómo se construye el archivo respectivo: `QW RT NB AS FG DF JH OP ZX BB YT LA CA WS MZ UI AW PN FR HL XC BT`.
2. Genere, a partir de las claves definidas en el ejercicio anterior, un árbol *B**:
   a. Con política de derecha.
   b. Con política de izquierda o derecha.
   c. Con política de derecha e izquierda.
3. A partir de la Figura 7.24, y sabiendo que corresponde a un árbol B de orden 5:
   a. Inserte `CV`. b. Inserte `XX`. c. Borre `HI`.

Ejercitación

4. Suponga ahora que la Figura 7.24 representa a un árbol B* de orden 5.

Figura 7.24



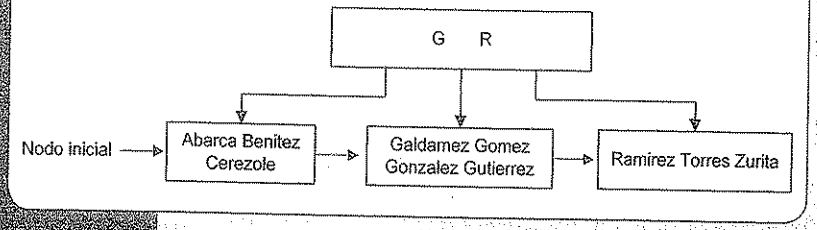
Resuelva los problemas del ejercicio anterior suponiendo que:

a. Se trabaja con la política de izquierda.

b. Se trabaja con la política de izquierda o derecha.

5. Dado el árbol de la Figura 7.25 y sabiendo que el mismo es un árbol B+ de prefijos simples de orden 4, inserte las siguientes claves: García, Hernández, Molinari, Perez.

Figura 7.25



Capítulo 8

Dispersión (*hashing*)

Objetivo

Un archivo directo, o con acceso directo, es un archivo en el cual cualquier registro puede ser accedido sin acceder antes a otros registros, es decir, cualquier registro puede ser accedido directamente. En un archivo serie, en cambio, un registro está disponible solo cuando el registro predecesor fue procesado. Resulta interesante pensar a un archivo directo como un arreglo de registros, en tanto que un archivo serie puede pensarse como una lista de registros.

Los archivos directos son almacenados en disco en diferentes formas respecto de los archivos serie. Esta forma de almacenamiento debe permitir una rápida recuperación de la información contenida en el archivo, que es el objetivo principal que se persigue con un archivo directo.

El mecanismo que trata de asegurar una recuperación rápida de registros, en un solo acceso promedio, lleva el nombre de dispersión o *hashing*. En este capítulo, se presentan las alternativas más comunes de dispersión: dispersión con espacio de direccionamiento estático y dispersión con espacio de direccionamiento dinámico. Además, se analizan con detalle las propiedades fundamentales del método y se desarrolla un estudio numérico que permite asegurar la localización rápida de un registro en particular.

Conceptos de dispersión. Métodos de búsqueda más eficientes

Suponga que se necesita acceder a un archivo de datos, y que para la rápida localización de los registros allí contenidos se genera un árbol *B+* de prefijos simples, como los estudiados en el Capítulo 7. Siguiendo el análisis numérico efectuado, en promedio son necesarios entre tres y cuatro accesos para recuperar un registro. Si bien esta *performance* fue considerada muy interesante en el capítulo anterior, se podría presentar alguna situación en la práctica donde sea ineficiente consumir cuatro accesos a memoria secundaria para recuperar un registro.

Entonces, ¿cuál es la alternativa disponible? Debe existir un mecanismo que permita reducir el número de accesos a disco.

Hashing o dispersión es un método que mejora la eficiencia obtenida con árboles balanceados, asegurando en promedio un acceso para recuperar la información.

Se presentan a continuación definiciones para el método:

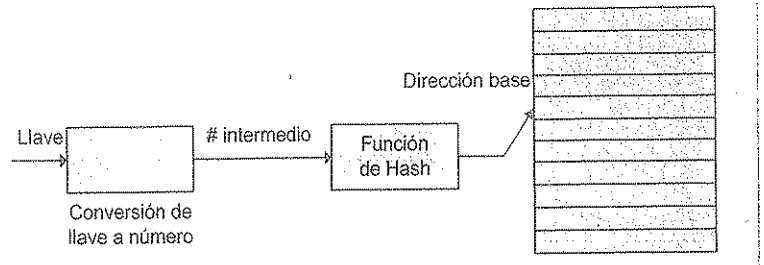
- Técnica para generar una dirección base única para una clave dada. La **dispersión** se usa cuando se requiere acceso rápido mediante una clave.
- Técnica que convierte la clave asociada a un registro de datos en un número aleatorio, el cual posteriormente es utilizado para determinar dónde se almacena dicho registro.
- Técnica de almacenamiento y recuperación que usa una función para mapear registros en direcciones de almacenamiento en memoria secundaria.

La primera definición plantea la dispersión como una técnica que permite generar la dirección donde se almacena un registro a partir de la llave o clave de dicho registro. La segunda definición agrega la conversión de la clave en un número aleatorio que será la base para determinar dónde el registro se almacenará. Ambas definiciones están en concordancia.

Por último, la tercera definición menciona que la técnica utiliza una función, que será responsable de obtener la dirección física de almacenamiento del registro.

Las tres definiciones anteriores pueden resumirse en la Figura 8.1. Aquí puede observarse cómo la llave primero se convierte en un número, sobre el cual puede aplicarse la función de *hash*, que permite obtener la dirección donde el registro se almacenará dentro del archivo.

Figura 8.1



La técnica de dispersión presenta una serie de atributos. Estos pueden resumirse en los siguientes:

- No se requiere almacenamiento adicional. Esto significa que cuando se elige la opción de dispersión como método de organización de archivos, es el archivo de datos el que resulta disperso. Esta dispersión se genera a partir de la clave o llave primaria del archivo, y el registro que contiene la información relacionada con la clave es ubicado en el espacio resultante de aplicar la función de *hash*. Así, no es necesario tener una estructura auxiliar que actúe como soporte para poder acceder rápidamente a la información.
- Facilita la inserción y eliminación rápida de registros en el archivo. Como se analizará más adelante en este capítulo, el proceso de inserción o borrado de información se realiza de una manera más eficiente en términos de accesos a disco. En general, con un solo acceso a disco, un registro puede ser insertado o eliminado del archivo. La ventaja de este método reside básicamente en este punto.
- Localiza registros dentro del archivo con un solo acceso a disco. Como corolario del punto anterior, otra ventaja del método de dispersión consiste en ubicar cada elemento de datos, en promedio, con un acceso a disco. Si bien no es posible asegurar que cualquier registro sea encontrado en un solo acceso, la gran mayoría de las búsquedas serán efectivamente satisfechas con un acceso a disco. Resta estudiar, en este capítulo, bajo qué condiciones no se respeta esta característica.

Sin embargo, el método de dispersión presenta limitaciones. Estas indican situaciones donde el método no es aplicable, o donde, a partir de su aplicación, no es posible lograr otras características que podrían ser deseadas para el archivo de datos. Estas limitaciones son las siguientes:

- No es posible aplicar la técnica de dispersión en archivos con registros de longitud variable. Esto se debe a que cada dirección física obtenida debe tener capacidad para almacenar un registro de tamaño conocido. Es decir, cuando se obtiene una dirección de almacenamiento, asociada a esta hay un espacio de almacenamiento conocido y acotado. No puede ocurrir que el registro a almacenar no quepa en dicho espacio.
- No es posible obtener un orden lógico de los datos. Utilizando índices como metodología de acceso a datos, no solo la búsqueda es eficiente, sino que además presenta la característica de mantener los registros ordenados. Bajo la técnica de *hashing* o dispersión no es posible preservar la propiedad de orden. Los registros son esparcidos en el archivo de datos, de acuerdo con la dirección que se obtiene con el uso de la función de *hash*.
- No es posible tratar con claves duplicadas. Así, no es aplicable la función de *hash* sobre una clave secundaria. Si se analiza con detalle dicha situación, se podrá observar a qué se debe este impedimento. Una clave secundaria puede repetirse; por lo tanto, dos registros diferentes con la misma clave secundaria, aplicando la función de *hash*, tendrían como resultado la misma dirección de memoria. Aquí ocurriría una contradicción con el principio básico del método de dispersión. Cada registro debe, *a priori*, residir en direcciones diferentes del archivo. Si se aceptara trabajar con claves secundarias, esta propiedad no se cumpliría.

Tipos de dispersión

El método de *hashing* presenta dos alternativas para su implantación: tratamiento de espacio en forma estática o tratamiento de espacio en forma dinámica.

Se denomina ***hashing* con espacio de direccionamiento estático** a aquella política donde el espacio disponible para dispersar los registros de un archivo de datos está fijado previamente. Así, la función de *hash* aplicada a una clave da como resultado una dirección física posible dentro del espacio disponible para el archivo.

Se denomina ***hashing* con espacio de direccionamiento dinámico** a aquella política donde el espacio disponible para dispersar los registros de un archivo de datos aumenta o disminuye en función de las necesidades de espacio que en cada momento tiene el archivo. Así, la función de *hash* aplicada a una clave da como resultado un valor intermedio, que será utilizado para obtener una dirección física posible para el archivo. Estas direcciones físicas no están establecidas *a priori* y son generadas de manera dinámica.

En el resto del capítulo, se presentarán ambas políticas con detalle. En primer término, será abordado el *hashing* con espacio de direccionamiento estático. De esta forma se podrán analizar el funcionamiento del método, sus principales características y el tratamiento de condiciones especiales. Luego, se analizarán los casos problemáticos y cómo estos pueden ser solucionados con espacio de direccionamiento variable.

Parámetros de la dispersión

El método de dispersión, cuando utiliza espacio de direccionamiento estático, presenta cuatro parámetros esenciales que definen su comportamiento. En este apartado, se abordan estos parámetros y se analiza con detalle la influencia que ejercen sobre el método.

Los cuatro parámetros a estudiar son los siguientes:

- Función de *hash*.
- Tamaño de cada nodo de almacenamiento.
- Densidad de empaquetamiento.
- Métodos de tratamiento de desbordes (*overflow*).

Función de *hash*

El primer parámetro a analizar es la función de *hash*. Esta función puede verse como una caja negra que recibe como entrada una clave, y produce una dirección de memoria donde almacenar el registro asociado a la clave en el archivo de datos.

Una **función de *hash* o dispersión** es una función que transforma un valor, que representa una llave primaria de un registro, en otro valor dentro de un determinado rango, que se utiliza como dirección física de acceso para insertar un registro en un archivo de datos.

La definición anterior determina que la función de *hash* debe retornar como resultado una dirección que se encuentre en un determinado

rango. Sin embargo, una vez aplicada dicha función sobre una clave, el valor resultante puede ser cualquiera.

Se debe tener presente que el conjunto de direcciones físicas sobre la estructura de almacenamiento secundario (disco rígido) es un elemento finito, es decir, el método cuenta *a priori* con un conjunto fijo y finito de direcciones disponibles.

Como la función de *hash* podría retornar cualquier valor, es necesario, previo a retornar el resultado final, mapear dicho valor dentro del rango de valores posibles.

Un ejemplo sencillo para una función de *hash* puede ser aquella que sume los valores ASCII de la clave y luego mapee el resultado dentro del rango de direcciones disponibles.

La Figura 8.2 presenta el algoritmo de dicha función de *hash*. Se puede observar en la figura que la función recibe como parámetro de entrada la clave a convertir y su dimensión, y que retorna la dirección física donde el registro será almacenado. Además, en la función de *hash* se conoce la cantidad de direcciones físicas disponibles, lo que permite mapear el resultado a un rango previamente establecido. Se debe asumir que la función `ASCII` retorna el código correspondiente al carácter indicado.

Figura 8.2

```
function  hash_ascii ( llave:  string, tamaño: integer ) :  integer;
var  valor, i : integer;
begin
     valor := ASCII( llave[1] );
     for i = 2 to tamaño
          valor := valor + ASCII( llave[i] );
     hash_ascii := valor / numero_direcciones_fisicas;
end;
```

Pueden observarse de la función anterior su sencillez, su facilidad de implementación y su velocidad de ejecución. Así, la función de *hash* aplicada sobre cualquier llave resolverá rápidamente la dirección física donde almacenar el registro asociado a la clave.

Sin embargo, esta sencillez tiene asociado un inconveniente. Suponga el lector dos claves diferentes: `DACA` y `CADA`. Si se aplica la función anterior, el valor ASCII resultante será el mismo; por lo tanto, los registros asociados a ambas claves se deberían almacenar en el mismo lugar de memoria. Esta situación genera una colisión entre ambos registros. En este caso, las llaves `DACA` y `CADA` se denominan sinónimos.

Las colisiones o desbordes ocasionan problemas. No es posible almacenar dos registros en el mismo espacio físico. Así, es necesario encontrar una solución a este problema. Las alternativas en este caso son dos:

- Elegir un algoritmo de dispersión perfecto, que no genere colisiones. Este tipo de algoritmo debe asegurar que dadas dos claves diferentes siempre se obtendrán dos direcciones físicas diferentes. Se ha demostrado que obtener este tipo de funciones resulta extremadamente difícil. En general, intentar generar estos algoritmos no es una opción válida cuando se decide trabajar con el método de dispersión.
- Minimizar el número de colisiones a una cantidad aceptable, y de esta manera tratar dichas colisiones como una condición excepcional. Aquí se debe tener en cuenta que uno de los parámetros que afectan la eficiencia del *hash*, y que aún no se han discutido, es precisamente el método de tratamiento de colisiones o desbordes. Existen diferentes modos de reducir el número de colisiones; las alternativas disponibles son:
  - Distribuir los registros de la forma más aleatoria posible. Las colisiones se presentan cuando dos o más claves compiten por la misma dirección física de memoria. Para ello, se debe buscar una función de dispersión que distribuya su resultado de la forma más aleatoria posible.
  - Utilizar más espacio de disco. Si se intentan distribuir 10 registros en 10 lugares, significa que hay un lugar disponible para cada registro, con lo cual las posibilidades de colisión son altas. Ahora bien, si se dispone de 100 lugares para cada uno de los 10 registros, se tendría un archivo con 10 direcciones posibles para cada registro. Esto, sin duda, debería disminuir el número de colisiones que se producen. La desventaja asociada con utilizar más espacio del necesario es, básicamente, el desperdicio de espacio. No hay una única respuesta para la pregunta acerca de cuánto espacio "vacío" debe dejarse o tolerarse en un archivo para minimizar las colisiones; a lo largo del capítulo, se analizarán alternativas.
  - Ubicar o almacenar más de un registro por cada dirección física en el archivo. Esto es, que cada dirección obtenida por la función de *hash* sea la dirección de un nodo o sector del disco donde es posible almacenar más de un registro. Así, dos claves, aun en situación de colisión, podrían ser almacenadas en la dirección física asignada por la función de *hash*, pues el nodo tiene mayor capacidad. Esto significa que las claves sinónimos para una función de *hash* determinada podrán albergarse en la misma

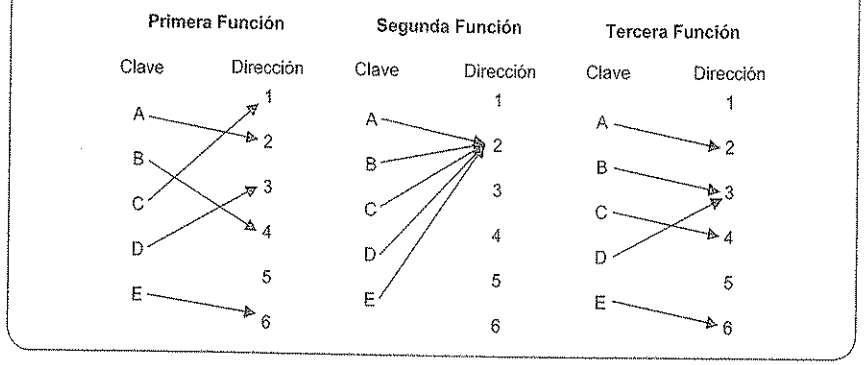
dirección física. No obstante, la capacidad de un nodo es limitada, es decir, puede albergar una cantidad (previamente fijada) de registros asociados a claves sinónimos. Si, a pesar de la mayor de capacidad del nodo, un registro ya no cabe en esa dirección, se dice que el nodo está saturado o en *overflow* (tema que será analizado posteriormente).

Uno de los objetivos fundamentales cuando se utiliza el método de dispersión o *hashing* es la selección de la función de *hash*. Esta función debe esparcir los registros de la manera más uniforme posible, es decir, que a cada clave se le asigne una dirección física distinta.

El algoritmo definido en la Figura 8.2 dista en mucho de ser un algoritmo uniforme, como ya se analizó.

La Figura 8.3 presenta el resultado de tres funciones de *hash* diferentes esparciendo un conjunto de registros.

Figura 8.3



Se puede observar que la primera función esparce los registros de manera uniforme; hay disponibles seis direcciones y los cinco registros distribuidos ocupan lugares diferentes. La segunda función es la peor posible; los cinco registros se ubican en el mismo lugar del disco. En tanto, la tercera función, si bien no es perfecta, puede considerarse una función aceptable. Allí, los registros *A* y *D* se ubican en el mismo sector, en tanto que el resto se asigna a diferentes lugares.

Aunque la distribución aleatoria de registros entre direcciones no representa la situación ideal, es una alternativa razonable, dado que es muy poco factible encontrar una función que logre una distribución realmente uniforme.

En bibliografía especializada del tema es posible analizar distintas alternativas de funciones de *hash*.

Tamaño de cada nodo de almacenamiento

Dentro de los parámetros que afectan la eficiencia del método de dispersión está presente el tamaño o capacidad de cada nodo.

En el apartado anterior, cuando se discutió la función de *hash*, fue presentada como alternativa a las colisiones la posibilidad de que un nodo tenga capacidad de albergar más de un registro.

Para determinar el tamaño ideal de un nodo, se realiza el mismo análisis que el efectuado en el Capítulo 7 para árboles balanceados. Es así que la capacidad del nodo queda determinada por la posibilidad de transferencia de información en cada operación de entrada/salida desde RAM hacia disco, y viceversa.

Posteriormente, se analizarán situaciones de poseer nodos con mayor capacidad y sus beneficios. Estos beneficios tienen que ver con la cantidad de colisiones producidas.

Densidad de empaquetamiento

El primer parámetro analizado, la función de *hash*, mostró las ventajas que se obtienen a partir de una elección que minimice las colisiones. También se analizó que con el uso de memoria adicional es posible reducirlas.

Se analizará un nuevo concepto, la densidad de empaquetamiento, como alternativa de evaluación.

> Se define la **Densidad de Empaquetamiento (DE)** como la relación entre el espacio disponible para el archivo de datos y la cantidad de registros que integran dicho archivo.

La siguiente fórmula describe la DE cómo la razón entre la cantidad de registros que componen un archivo (*r*) y el espacio disponible para almacenar ese archivo.

El espacio disponible se define como la cantidad de nodos direccionables (*n*) por la función de *hash*, y la cantidad de registros que cada nodo puede almacenar, **Registros por Nodo (RPN)**.

$$DE = \frac{r}{RPN * n}$$

Si se debieran esparcir 30 registros entre 10 direcciones con capacidad de cinco registros por cada dirección, la DE sería de 0.6 o 60% (30 / 5 * 10).

Cuanto mayor sea la DE, mayor será la posibilidad de colisiones, dado que en ese caso se dispone de menos espacio para esparcir registros. Por el contrario, si la DE se mantiene baja, se dispone de mayor espacio para esparcir registros y, por ende, disminuye la probabilidad de colisiones.

Por otra parte, cuando la DE se mantiene baja, se desperdicia espacio en el disco, dado que se utiliza menor espacio que el reservado, generando fragmentación.

La DE no es constante. Al inicio, la DE será baja. Suponga que se crea un archivo, al cual inicialmente se le agrega un registro, y que la función de *hash* puede direccionar 100 lugares de disco y cada uno tiene capacidad para 10 registros. Esto significa que la DE inicial es (1 / 100 * 10), lo que puede considerarse como un número muy bajo, con un alto desperdicio de espacio en disco. Sin embargo, a medida que el archivo es utilizado, se van incorporando nuevos registros. Luego de un tiempo de uso, el archivo puede contener 500 registros, y la DE aumenta a 50%. El archivo puede continuar "creciendo" hasta tener 990 registros almacenados, con una DE de 99%. Note el lector que, en este caso, se debería considerar aumentar el espacio disponible, dado que el archivo está próximo a su límite de espacio.

La acción de aumentar el espacio de direcciones (por ejemplo, de 100 a 200 direcciones) implica reubicar a todos los registros ya almacenados. Si la función de *hash* en uso esparce entre 100 direcciones, al aumentar el número base de direcciones, la función de *hash* debe cambiar para adaptar sus resultados al nuevo espacio disponible y, por ende, todos los registros esparcidos deben ser reubicados.

Métodos de tratamiento de desbordes (*overflow*)

Un desborde u *overflow* ocurre cuando un registro es direccionado a un nodo que no dispone de capacidad para almacenarlo. Cuando esto ocurre, deben realizarse dos acciones: encontrar lugar para el registro en otra dirección y asegurarse de que el registro posteriormente sea encontrado en esa nueva dirección. Para efectivizar estas dos acciones, existen diferentes métodos. Más adelante, en este mismo capítulo, estos métodos serán tratados junto con la *performance* de cada uno de ellos.

En el apartado siguiente, se realizará un estudio con respecto a la probabilidad de ocurrencia de *overflow*.

Estudio de la ocurrencia de *overflow*

Como ya se planteó en este capítulo, una de las características principales del método de dispersión es asegurar que los registros sean rápidamente localizados en el archivo, a tal punto que se plantea como acceso directo a la información, lo que implica encontrar los registros en un solo acceso a disco. Si bien esto es posible, no siempre se puede cumplir.

En este apartado se analizará la *performance* del método de dispersión, y se estudiará bajo qué circunstancias se puede garantizar acceso directo y cuándo esto no es posible.

Dado que generalmente no se puede lograr una distribución uniforme de los registros en el espacio de direcciones disponible, se debe estudiar la probabilidad de distribución. Esto permite analizar cuáles direcciones quedarán saturadas o completas y cuáles no.

Al intentar ubicar un registro, la función de *hash* retorna la dirección física donde el registro debe residir. Esta dirección física es la dirección de un nodo dentro del espacio de direccionamiento definido. Una vez dispersados todos los registros de un archivo, las preguntas que sería interesante responder son las siguientes:

- ¿Cuántos nodos no recibieron ningún registro?
- ¿Cuántos nodos recibieron solo un registro?
- En general, ¿cuántos nodos recibieron *i* registros?

Para poder realizar este análisis, será necesario considerar algunas nociones mínimas de probabilidades.

En el problema que se analizará a continuación, se definen una serie de variables que lo identifican:

N representa el número de direcciones de nodos disponibles en memoria secundaria.

K determina la cantidad de registros a dispersar.

i determina la cantidad de registros que contendrá un nodo en un momento específico.

C determina la capacidad de cada nodo.

Luego, es necesario poder determinar la probabilidad de que un nodo reciba *i* registros.

Se demostrará que:

$$P(i) = \frac{K! * (1/N)^i * (1 - 1/N)^{K-i}}{i! * (K-i)!}$$

Se denominará $P(A)$ a la probabilidad de direccionar un nodo específico de los N disponibles, y $P(B)$, a la probabilidad de no direccionar ese nodo específico, es decir, direccionar cualquiera de los N-1 restantes.

Si se arroja una moneda al aire, la probabilidad de obtener cara o ceca es igual, 1/2 en cada caso.

Si se arroja un dado, la probabilidad de obtener un número específico es 1/6, debido a que hay seis resultados posibles.

Generalizando el caso anterior, $P(A)$ indica la probabilidad de que una clave sea direccionada a un nodo específico dentro de los N disponibles, entonces:

$P(A) = 1/N$

Como $P(B)$ complementa a $P(A)$,

$P(A) + P(B) = 1$

Entonces, $P(B) = 1 - P(A)$; la probabilidad de no ir a un nodo específico es direccionar cualquiera de los restantes.

Reemplazando $P(A)$:

$P(B) = 1 - P(A) = 1 - 1/N = (N - 1) / N$

¿Cuál es la probabilidad de que dos claves cualesquiera ocupen un nodo específico?

Siguiendo el razonamiento anterior, se intenta analizar la posibilidad de $P(AA)$, es decir, que dos claves se ubiquen en el mismo nodo.

Cuando los eventos controlados por las probabilidades resultan independientes, ocurre que:

$P(AA) = P(A) * P(A) = (1/N) * (1/N)$

En este caso, la dirección obtenida para una clave no condiciona a la obtenida para otra clave. Es decir, dos claves diferentes no se condicionan entre sí, y por lo tanto, se pueden considerar independientes.

Continuando con el mismo razonamiento, la probabilidad de que dos claves no ocupen un nodo específico será:

$P(BB) = P(B) * P(B) = (1 - 1/N) * (1 - 1/N)$

Por último, la probabilidad de que una clave se direccione a un nodo específico, y otra clave, a un nodo diferente, será:

$P(AB) = P(A) * P(B) = (1/N) * (1 - 1/N)$

Con el mismo criterio, con tres claves, se puede observar que:

$P(AAA) = P(A) * P(A) * P(A) = (1/N) * (1/N) * (1/N) = (1/N)^3$

A partir de tres claves se pueden analizar varias alternativas: $P(ABB)$, $P(BBB)$, $P(BBA)$, etcétera.

Generalizando lo anterior, ¿cuál sería la probabilidad de que un nodo reciba i registros?

$P(A...A) = P(A) *...* P(A) = (1/N)^i$ (se define $A...A$ como la ocurrencia de A i veces)

El problema original consistía en analizar la probabilidad de que i llaves, de las K disponibles, se direccionaran hacia un nodo específico, es decir, analizar $P(A)$ i veces y $P(B)$ K-i veces.

$P(A...AB...B) = P(A) *...* P(A) * P(B) *...* P(B) = P(A)^i * P(B)^{K-i} = (1/N)^i * (1 - 1/N)^{K-i}$

El resultado anterior corresponde tanto a $P(A...AB...B)$ como a $P(B...BA...A)$, con la ocurrencia de A i veces y B K-i veces.

Queda por analizar aún cuántas combinaciones de A y B producen el mismo resultado. Se puede observar que de las K llaves disponibles interesa la ubicación de i claves. La forma de selección de i llaves, a partir de las K disponibles, responde al número combinatorio K tomado de i. En términos matemáticos, esto se expresa como:

$K! / (i! * (K-i)!)$

Entonces, $P(i)$; la probabilidad de que un nodo reciba i claves entre las K disponibles será:

$$P(i) = \frac{K! * (1/N)^i * (1 - 1/N)^{K-i}}{i! * (K-i)!}$$

Este resultado corresponde a la fórmula inicial planteada.

Poisson demostró (esto es obviado) que la probabilidad anterior se puede acotar a partir de la siguiente fórmula:

$$P(i) = \frac{(K/N)^i * e^{-(K/N)}}{i!}$$

Ejemplo 1

Corresponde ahora estudiar numéricamente el problema. Suponga que se dispone de 10.000 direcciones donde se deben almacenar 10.000 llaves, y que cada dirección solo puede almacenar un registro. En este caso:

$N = 10.000$, $K = 10.000$, $C = 1$

La DE, de acuerdo con la fórmula antes definida, será:

$DE = K / (N * C) = 10.000 / 10.000 = 1$

El cálculo de la probabilidad de que un nodo no reciba ninguna clave (i = 0), utilizando la función de Poisson, será:

$$P(0) = \frac{(10.000 / 10.000)^0 * e^{-(10.000/10.000)}}{0!} = 0.3679$$

Esto significa que 36,79% de las direcciones disponibles no serán direccionadas por la función de *hash* aplicada a cualquiera de las 10.000 llaves. Es decir que 3.679 (36.79% de 10.000) nodos quedarán sin registros.

Si se calcula la probabilidad de que un nodo reciba un solo registro ($i = 1$):

$$P(1) = \frac{(10.000 / 10.000)^1 * e^{-(10.000/10.000)}}{1!} = 0.3679$$

Nuevamente, 3.679 nodos contendrán solo un registro.

$P(2) = 0.1839$, lo que significa que 1.839 nodos recibirán dos registros. Se debe notar que en este caso se tendrán 1.839 registros en saturación u *overflow*. El primer registro cabe en el nodo, y como el nodo solamente puede almacenar un solo registro, el segundo registro produce *overflow*.

$P(3) = 0.0613$ significa 613 direcciones con tres registros. En este caso, se tendrán 1.226 registros en saturación. La función de *hash* intenta asignar tres registros a 613 direcciones; el primero cabe, no así los dos restantes (613 * 2 = 1.226).

Generalizando el problema, hasta el momento se esperan 3.065 (1.839 + 1.226) registros en *overflow*, una cantidad importante. Es decir, hasta aquí, más de 30% de los registros dispersos en el archivo no serán asignados a la dirección base obtenida por la función de *hash*.

Ejemplo 2

Suponga ahora $K = 10.000$, $N = 20.000$, $C = 1$. La DE en este caso se reduce a 50%.

La siguiente tabla muestra la probabilidad de tener nodos sin registros asignados, o con uno, dos, tres o cuatro registros asignados en cada caso.

i	P(*i*)	*N* * P(*i*)	Saturación (*N* * P(*i*) * (*i*-1))
0	0.6065	12.130	0 registros
1	0.3032	6.065	0 registros
2	0.0758	1.516	1.516 registros
3	0.0126	252	504 registros
4	0.0016	32	96 registros

N * P(*i*) muestra el número esperable de nodos que tendrán asignado cero, uno, dos, tres o cuatro registros. En el caso de no tener registros o solamente contener uno, no se produce saturación. Los casos restantes generan *overflow*.

Se puede observar que, en este caso, la saturación es de 2.116 registros, lo que significa 21% de los registros esparcidos.

El resultado es mejor que el presentado anteriormente, ya que con una DE del 100%, la probabilidad de saturación es del orden de 36%.

Por lo tanto, si se baja la DE a 50%, el *overflow* se reduce a 21%.

De aquí se pueden obtener dos conclusiones. La positiva es que, a menor tasa de empaquetamiento, se reduce la condición de *overflow*. La conclusión negativa es que, con baja tasa de DE, aún existe un alto porcentaje de saturación.

Ejemplo 3

Bajo las mismas condiciones de DE del ejemplo anterior (DE = 50%), suponga que $K = 10.000$, $N = 10.000$, $C = 2$.

Se debe notar que la DE no varía pues el denominador sigue siendo el mismo, pero la relación entre *K* y *N* ahora es uno.

i	P(*i*)	*N* * P(*i*)	Saturación (*N* * P(*i*) * (*i*-1))
0	0.3678	3.678	0 registros
1	0.3678	3.678	0 registros
2	0.1839	1.839	0 registros
3	0.0613	613	613 registros
4	0.0153	153	306 registros

Se debe notar que ahora, cuando un nodo recibe dos registros, no genera saturación ($C = 2$). Por lo tanto, se tendrá un registro en *overflow* con $i = 3$, y dos registros en dicha situación con $i = 4$. En este caso, se tendrán 919 registros en saturación, menos de 10% de la cantidad total.

Se puede concluir que, si se aumenta la capacidad de cada nodo, el porcentaje de *overflow* se reduce.

La siguiente tabla resume el estudio anterior. Se presentan cálculos de saturación con diferentes DE y diferentes capacidades para los nodos (que pueden almacenar 1, 2, 5, 10 o 100 registros).

DE	1	2	5	10	100
10%	4.8	0.6	0.0	0.0	0.0
20%	9.4	2.2	0.1	0.0	0.0
30%	13.6	4.5	0.4	0.0	0.0
40%	17.6	7.3	1.1	0.1	0.0
50%	21.3	10.4	2.5	0.4	0.0
60%	24.8	13.7	4.5	1.3	0.0
70%	28.1	17.0	7.1	2.9	0.0
75%	29.6	18.7	8.6	4.0	0.0
80%	31.2	20.4	10.3	5.3	0.1
90%	34.1	23.8	13.8	8.9	0.8
100%	36.8	27.1	17.6	12.5	4.0

Es posible observar que, a medida que la capacidad de cada nodo aumenta, la probabilidad de saturación disminuye.

Para el caso de disponer de una DE de 75% con nodos que pueden almacenar hasta 100 registros, la probabilidad de colisiones es de menos de 0,09%.

Resolución de colisiones con *overflow*

Aunque la función de *hash* sea eficiente y aun con DE relativamente baja, es probable que las colisiones produzcan *overflow* o saturación. Por este motivo, se debe contar con algún método para reubicar a aquellos registros que no pueden ser almacenados en la dirección base obtenida a partir de la función de *hash*.

Se presentan cuatro de estos métodos: saturación progresiva, saturación progresiva encadenada, doble dispersión y área de desbordes por separado.

Saturación progresiva

El método consiste en almacenar el registro en la dirección siguiente más próxima al nodo donde se produce saturación.

Suponga que tiene un caso como la Figura 8.4 a). El problema simplificado trata con nodos de capacidad 2. En el nodo 50, se encuentran los registros correspondientes a `Alvarez y Gonzales`. Se inserta un nuevo registro, Perez, y la función de *hash* retorna como dirección base al nodo 50. Esta situación genera una saturación, debido a que el nodo 50 se encuentra completo. La primera dirección del nodo posterior que tiene capacidad para contener a Perez es la dirección 52. El registro, por lo tanto, es almacenado en dicho lugar, y la situación queda como la indicada en la Figura 8.4 b).

Figura 8.4

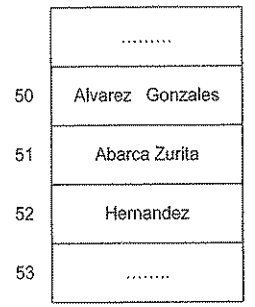


Figura 8.4 a)

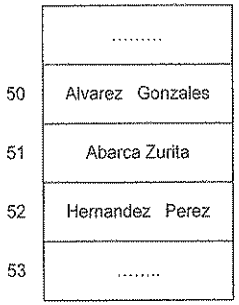


Figura 8.4 b)

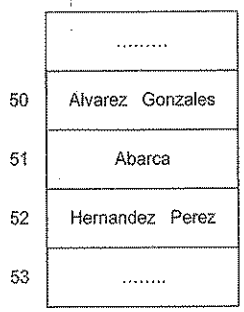


Figura 8.4 c)

El proceso de búsqueda de información debe, ahora, sufrir algún cambio. Si se busca a Perez, la dirección base que determina la función de *hash* seguirá siendo la dirección 50. Como el elemento no se encuentra en dicho lugar y el nodo está completo, se debe continuar la búsqueda en los nodos subsiguientes hasta encontrar el elemento, o hasta encontrar un nodo que no esté completo.

En el caso del ejemplo planteado, no se encuentra a Perez en el nodo 50, tampoco en el nodo 51 y sí es localizado en el nodo 52. Se debe notar la simpleza del método, pero además su limitada eficiencia. En el ejemplo planteado fueron necesarios tres accesos hasta encontrar al registro.

Suponga ahora que, con base en el ejemplo de la Figura 8.4 b), se busca al registro cuya clave es Rodríguez. El resultado de la función de *hash* retorna la dirección 50. Se busca al elemento en el nodo 50, luego en el 51 y en el 52 hasta que, cuando se busca en el 53, se encuentra un nodo no completo y sin el elemento Rodriguez. Es posible decidir, en ese momento, que el elemento no está en el archivo.

El método podría requerir chequear todas las direcciones disponibles en un caso extremo, para poder localizar un registro.

Asimismo, esta técnica necesita una condición excepcional adicional. Suponga ahora que en la Figura 8.4 b) se elimina el registro correspondiente a Zurita, y el gráfico queda como lo muestra la Figura 8.4 c). En este caso, si se intenta localizar a Perez, se presenta un inconveniente. Al chequear la dirección 51, esta no se encuentra completa, por lo que la búsqueda se detendría, sin permitir que Perez sea localizado.

El método necesita indicar que si una dirección estuvo completa anteriormente, debe ser marcada como dirección ya saturada a fin de no impedir la búsqueda potencial de registros. En este caso, el nodo 51, si bien no está saturado, al haberlo estado mantiene esa condición. Se podría utilizar # para indicar dicha situación en el nodo 51 (de ese modo, el nodo quedaría con Abarca #).

Saturación progresiva encadenada

La saturación progresiva encadenada presenta otra variante al tratamiento de la saturación. En líneas generales, el método funciona igual a su predecesor. Un elemento que se intenta ubicar en una dirección completa es direccionado a la inmediata siguiente con espacio disponible. La diferencia radica en que, una vez localizada la nueva dirección, esta se encadena o enlaza con la dirección base inicial, generando una cadena de búsqueda de elementos.

La Figura 8.5 a) presenta dicha situación. Se puede observar que ahora cada nodo tiene además la dirección siguiente. Inicialmente, las direcciones siguientes contienen –1, indicando que no direccionan a ningún nodo con saturación. Cuando se inserta Perez [Figura 8.5 b)], el registro se almacena en la dirección 52 y, por ende, se genera el enlace respectivo.

Figura 8.5

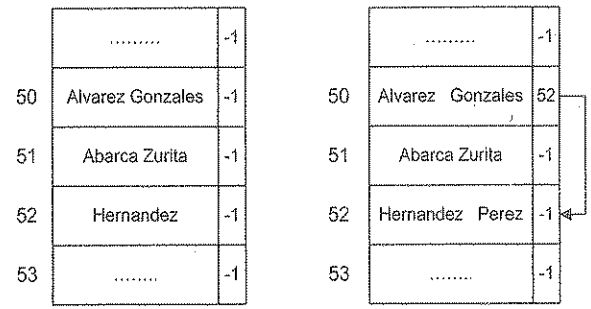


Figura 8.5 a) Figura 8.5 b)

A partir del ejemplo anterior, es posible observar que con saturación progresiva fueron necesarios tres accesos para recuperar a Perez, en tanto que con saturación progresiva encadenada solamente se requirieron dos accesos. Si bien la *performance* final de cada método dependerá del orden de llegada de las llaves, en líneas generales puede establecerse que el método de saturación progresiva encadenada presenta mejoras de *performance* respecto de su predecesor. Sin embargo, requiere que cada nodo manipule información extra: la dirección del nodo siguiente.

Cabe considerar que el enlace entre nodos sirve tanto para la búsqueda como para la inserción de nuevas claves. Si, por ejemplo, se desea insertar una clave a la cual la función de *hash* le asigna el nodo 50, al estar este nodo lleno se intenta insertar en el siguiente según el enlace (nodo 52). Nuevamente, al estar completo el nodo 52, se busca en el próximo nodo indicado en el enlace, que como en este caso tiene valor –1, hace que se deba buscar el nodo siguiente más próximo con espacio libre.

Existen algunas otras variantes y consideraciones respecto de este método de tratamiento de desbordes, que quedan por fuera del contenido del presente material.

Doble dispersión

El problema general que presenta la saturación progresiva es que, a medida que se producen saturaciones, los registros tienden a esparcirse en nodos cercanos. Esto podría provocar un exceso de saturación sectorizada. Existe un método alternativo denominado doble dispersión.

El método consiste en disponer de dos funciones de *hash*. La primera obtiene a partir de la llave la dirección de base, en la cual el registro será ubicado.

De producirse *overflow*, se utilizará la segunda función de *hash*. Esta segunda función no retorna una dirección, sino que su resultado es un desplazamiento. Este desplazamiento se suma a la dirección base obtenida con la primera función, generando así la nueva dirección donde se intentará ubicar al registro. En caso de generarse nuevamente *overflow*, se deberá sumar de manera reiterada el desplazamiento obtenido, y así sucesivamente hasta encontrar una dirección con espacio suficiente para albergar al registro.

La doble dispersión tiende a esparcir los registros en saturación a lo largo del archivo de datos, pero con un efecto lateral importante. Los registros en *overflow* tienden a ubicarse "lejos" de sus direcciones de base, lo cual produce un mayor desplazamiento de la cabeza lectora/grabadora del disco rígido, aumentando la latencia entre pistas y, por consiguiente, el tiempo de respuesta.

Área de desbordes por separado

Ante la ocurrencia de *overflow*, los registros son dispersados en nodos que no se corresponden con su dirección base original. Así, a medida que se completa un archivo por dispersión, pueden existir muchos registros ocupando direcciones que originalmente no les correspondían, disminuyendo la *performance* final del método de *hashing* utilizado.

Para evitar estas situaciones, se sugiere como alternativa el uso del área de desbordes por separado. Aquí se distinguen dos tipos de nodos: aquellos direccionables por la función de *hash* y aquellos de reserva, que solo podrán ser utilizados en caso de saturación pero que no son alcanzables por la función de *hash*.

Figura 8.6

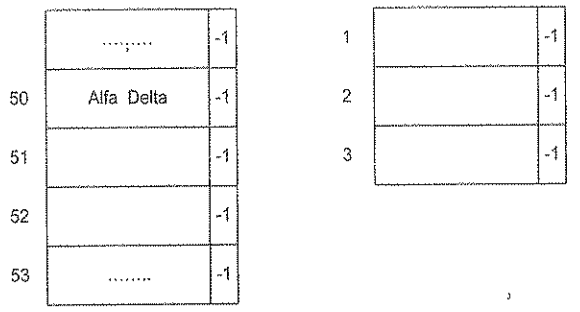


Figura 8.6 a)

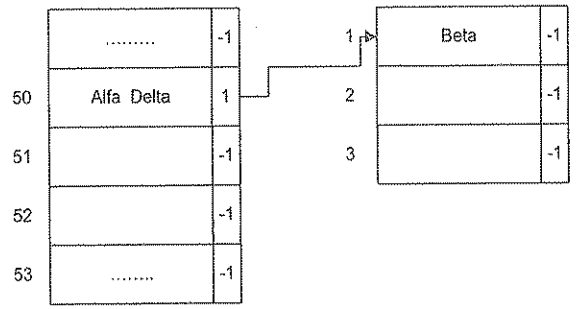


Figura 8.6 b)

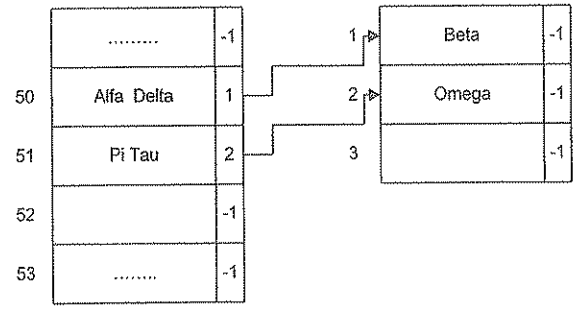


Figura 8.6 c)

continúa >>>

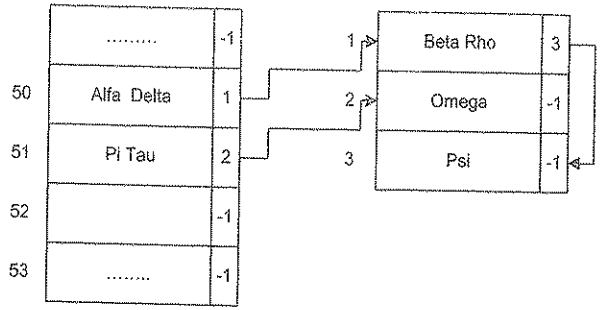


Figura 8.6 d)

La Figura 8.6 a) presenta un ejemplo del problema.

Las llaves Alfa y Delta direccionan la cubeta 50. Suponiendo que esta tiene capacidad para dos registros, se completa. Al dispersar la clave Beta, la dirección obtenida a partir de la función de *hash* es nuevamente la dirección 50. Se produce saturación, y el registro es reubicado en la primera dirección disponible dentro del área de desbordes separada [Figura 8.6 b)]. La dirección base original se encadena con la dirección de reserva.

Luego, como se muestra en la Figura 8.6 c), se dispersan las llaves Pi y Tau, con dirección base 51; si llega Omega al mismo nodo, se produce *overflow* y se utiliza otra dirección del área de desbordes, la 2 en este caso.

Por último, la Figura 8.6 d) muestra lo que ocurriría si llegaran dos claves más que obtuvieran la dirección 50. En este caso, Rho y Psi; la primera de ellas se ubica en la dirección 1 de desbordes; la segunda no cabe: por lo tanto, se redirecciona un nuevo nodo, el tercero del área de desbordes, el cual se enlaza con el primero.

Hash asistido por tabla

El método de *hash* resultó ser el más eficiente en términos de recuperación de información; permite conseguir un acceso para recuperar un dato en más de 99.9% de los casos, cuando la DE es inferior a 75%. Asimismo, las operaciones de altas y bajas se comportan con el mismo nivel de eficiencia para los mismos porcentajes.

Sin embargo, restan aún situaciones en las que puede ser necesario utilizar más de un acceso para recuperar o almacenar un registro. Si bien se discutieron anteriormente alternativas eficientes, es de notar que, a medida que se generan situaciones de saturación, el número de accesos requeridos aumenta.

Existe una alternativa que sigue utilizando espacio de direccionamiento estático y que, aun así, asegura acceder en un solo acceso a un registro de datos. Esta variante para *hash* estático se denomina *hash* asistido por tabla, y para poder ser implementada requiere que una de las propiedades (definidas al principio de este capítulo) no sea cumplida; necesita una estructura adicional.

El método utiliza tres funciones de *hash*: la primera de ellas retorna la dirección física del nodo donde el registro debería residir (F1H); la segunda retorna un desplazamiento, similar al método de doble dispersión, (F2H); y la tercera retorna una secuencia de bits que no pueden ser todos unos (F3H). El método comienza con una tabla con tantas entradas como direcciones de nodos se tengan disponibles. Cada entrada tendrá todos sus bits en uno.

Ante cada inserción, la primera función de *hash* retorna la dirección del nodo donde se debe almacenar el registro. Si el nodo tiene suficiente espacio, es decir, no se produce saturación, el registro es almacenado en ese lugar.

La Tabla 8.1 presenta el resultado de aplicar las tres funciones de *hash* a todas las claves que se utilizarán en el ejemplo. A fines prácticos, F3H retorna, en este caso, solamente 4 bits. Se debe notar que, en dicha tabla, la F1H retorna muchas direcciones similares, para poder presentar la forma que el método tiene para tratar la saturación.

Tabla 8.1

Clave	F1H(Clave)	F2H(Clave)	F3H(Clave)
Alfa	50	3	0001
Beta	51	4	0011
Gamma	52	7	0101
Delta	50	3	0110
Epsilon	52	3	1000
Rho	51	5	0100
Pi	50	2	0010
Tau	53	2	1110
Psi	50	9	1010
Omega	50	4	0000

La Figura 8.7 a) muestra la llegada de las claves Alfa, Beta, Gamma, Delta, Epsilon y Rho; suponga que cada nodo tiene capacidad para dos elementos.

Figura 8.7

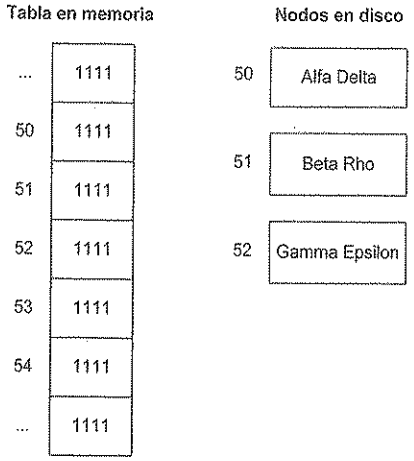


Figura 8.7 a)

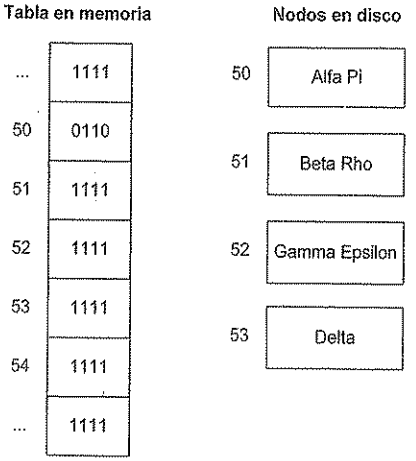


Figura 8.7 b)

continúa >>>

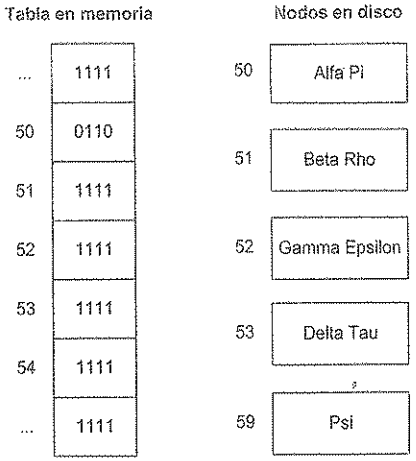


Figura 8.7 c)

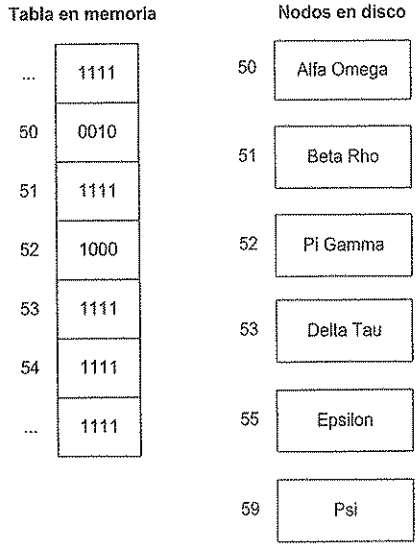


Figura 8.7 d)

Puede observarse que, en cada uno de los casos planteados, no se ha generado saturación. La siguiente clave en la Tabla 8.8 es Pi, que

debería insertarse en la dirección 50. Como esta dirección está completa, el procedimiento es el siguiente:

1. Se obtiene el valor de la F3H para todos los registros en el nodo 50.
2. Se determina la clave que genera el valor mayor. En este caso, Delta tiene el valor mayor.
3. Se escribe el nodo 50 con todos los registros menos el que posee F3H mayor.
4. En la tabla en memoria, para la posición 50 queda el valor de F3H de Delta. La Figura 8.7 b) presenta el estado de dicho nodo.
5. Para la clave seleccionada en el Paso 2, Delta para este ejemplo, se obtiene la F2H.
6. Se suma, a la F1H de Delta, el desplazamiento determinado por F2H.
7. Se intenta insertar Delta en la dirección resultante, en caso de que no haya saturación. Si la hubiera, se procederá nuevamente con los pasos indicados desde 1. Nuevamente, la Figura 8.7 b) presenta el estado final del proceso de inserción con saturación.

Para localizar la llave Delta a partir del archivo generado en la Figura 8.7 b), el proceso debe asegurar un solo acceso a disco, y el procedimiento a realizar es el siguiente:

1. Se genera la F1H para la clave a buscar.
2. Se genera la F3H para la misma clave.
3. Se chequea el contenido de la posición de la tabla indicada por F1H, comparándolo con el resultado de F3H.
   a. Si el resultado de F3H tiene valor menor que el encontrado en la tabla, se busca la clave en cuestión en la dirección del nodo indicada por F1H y el proceso termina.
   b. Si la F3H tiene valor mayor o igual que el encontrado en la tabla, el proceso continúa como se indica en el Paso 4.
4. Se obtiene el valor de F2H y se suma al valor indicado por F1H, obteniéndose una nueva posición.
5. Se comienza nuevamente desde el Paso 3, utilizando ahora como posición el resultado de F1H más el desplazamiento.

De acuerdo con los pasos indicados anteriormente, si se busca Delta, F1H(Delta) = 50, F3H(Delta) = 0110, al comparar este último valor con el almacenado en la dirección 50 de la tabla, el resultado es igual; por lo tanto, se calcula F2H(Delta) = 3 y se suma dicho valor (desplazamiento) a la dirección base previamente obtenida, y el resultado es 53. Se compara el resultado de F3H(Delta) con el contenido de la dirección 53 de la tabla. Como el valor de F3H es menor, la clave Delta debería encontrarse en la dirección 53.

Se accede al nodo 53 y se recupera Delta. Cabe destacar que hubo un solo acceso a memoria secundaria; el resto de las operaciones fueron sobre memoria principal.

Suponga que se desea acceder a la clave Psi, procediendo como en el ejemplo anterior y utilizando los valores de la Tabla 8.1, la dirección de F1H(Psi) = 50, F3H(Psi) = 1010. Como dicho valor (1010) es mayor que la entrada 50 de la tabla, la clave buscada no está en el nodo 50.

Se obtiene la F2H(Psi) = 2, se suma el desplazamiento (50 + 2) y se chequea la dirección 52 de la tabla; en este caso, el resultado de F3H es menor y se accede a la dirección 52 donde Psi no se localiza.

En este caso, se está en condiciones de responder que la clave buscada no se encuentra en el archivo.

Quedan para insertar cuatro claves de la tabla. A continuación, se muestra cómo es el proceso en cada caso. Siguiendo el procedimiento de inserción descripto anteriormente, la clave Tau no produce *overflow*, por lo que se inserta en el nodo 53.

La llave Psi se debería almacenar en el nodo 50, nuevamente produce *overflow*. Se calcula F3H para Alfa, Pi y Psi, y al ser el valor de Psi el mayor, es esta la clave que se quita del nodo 50. En este caso, como el valor de F3H(Psi) es mayor que el valor contenido en la dirección 50 de la tabla, este valor no debe ser cambiado para que la búsqueda pueda realizarse sin problemas.

El desplazamiento (FH2) indicado para Psi es 9, por lo que la dirección utilizada para almacenar Psi es 59. La Figura 8.7 c) muestra cómo queda el archivo luego de insertar Tau y Psi.

Por último, Omega también debe ser almacenada en el nodo 50.

Se produce *overflow* y, entre las tres claves, la que tiene F3H mayor es Pi. En el nodo 50 quedan Alfa y Omega, se actualiza la entrada 50 de la tabla con el valor de F3H(Pi) y se intenta ubicar a Pi en la dirección 52 (50 + el desplazamiento indicado por FH2 para Pi).

Se produce nuevamente *overflow*. Entre las tres claves (Pi, Gamma y Epsilon), la que posee F3H mayor es Epsilon. En el nodo 52 quedarán, entonces, Pi y Gamma, actualizando la entrada 52 de la tabla (con el valor de FH3 para Epsilon).

Epsilon será direccionada al nodo 55 [que se obtiene de sumar F1H(Epsilon) + F2H(Epsilon), como lo muestra la Figura 8.7 d)].

El método asegura encontrar la clave buscada en un solo acceso a disco.

El costo asociado presenta dos aristas. La primera tiene que ver con la necesidad de utilizar espacio extra para la administración de la tabla en memoria principal. La segunda, en tanto, tiene que ver con la complejidad adicional en caso de una inserción que produzca saturación. En este caso, la cantidad de accesos para terminar el proceso está vinculada con la cantidad de *overflows* sucesivos que se produzcan.

La conclusión final de esta variante para *hash* con espacio de direccionamiento estático es que este método pondera la búsqueda sobre las otras operaciones.

El proceso de eliminación

Es necesario, además, hacer mención –aunque de manera somera– del proceso de eliminación que utiliza la variante de *hash* asistido por tabla.

A fines prácticos, se puede establecer el siguiente proceso para dar de baja un registro del archivo:

1. Se localiza el registro de acuerdo con el proceso de búsqueda definido anteriormente.
2. Si el paso anterior encontró la llave buscada, se reescribe el nodo en cuestión sin el elemento a borrar.

Es de notar la simplicidad del proceso de eliminación. Sin embargo, se debe tener en cuenta un caso especial. Suponga que se borra una clave de un nodo que se encontraba completo. Cuando se intente insertar un nuevo elemento en el nodo, habrá lugar.

Si el nodo ya había producido *overflow*, la tabla en memoria contiene un valor correspondiente a la F3H de la llave que produjo saturación. En ese caso, el nuevo elemento a insertar debe cumplir la siguiente propiedad:

F3H(nuevo elemento) < Tabla(dirección del nodo)

es decir, la tercera función de *hash* debe otorgar un valor menor al dato que se encuentra en la tabla en memoria para el nodo en cuestión.

Hash con espacio de direccionamiento dinámico

Hasta el momento, se ha considerado al método de *hash* como una alternativa de almacenamiento que direcciona una clave a un nodo dentro de un conjunto de direcciones disponibles y establecidas de antemano. En todos los ejemplos planteados se determinaron el número de nodos y la capacidad de ellos, sin estudiar ese punto con detalle.

Además, cuando se analizó la probabilidad de saturación, quedó establecido un umbral (de 75%) deseado para la DE; es decir, mientras la DE se encuentre en un valor inferior a ese tope, la probabilidad de *overflow* disminuirá considerablemente y tenderá a cero.

Se debe notar que falta realizar un análisis mayor con respecto a la cantidad de direcciones disponibles, en el momento en que se empieza a trabajar con un archivo utilizando dispersión.

Suponga que se dispone de 100 direcciones de nodos con capacidad para 200 registros cada una de ellas. Entre todas esas direcciones es posible dispersar hasta 20.000 registros. Para un problema particular, cualquiera podría ser la cantidad de registros y, por ende, se podría utilizar una función de *hash* que dispersara las claves entre los 100 nodos disponibles. De ese modo, si la tasa de crecimiento del archivo fuera de 100 nuevos registros por día, luego de 150 días se llegaría a una DE de 75%. A partir de ese punto, la DE sería lo suficientemente elevada como para generar más saturación de la esperada. Además, luego de 200 días, el archivo alcanzaría el tope máximo posible de 20.000 registros.

La pregunta que debería realizarse es: ¿qué hacer en dicho caso? La respuesta es clara. Cuando el archivo se completa, es necesario obtener mayor cantidad de direcciones para nodos. Y también es necesario que la función de *hash* direccione más nodos de disco.

El problema tiene entonces una solución sencilla, se aumentan las direcciones de 200 a 400. Será necesario, además, modificar la función de *hash*. Ahora debe dispersar entre 400 direcciones disponibles, y eso traerá aparejado que todo el archivo deberá ser redispersado, dado que la función original debe cambiar.

El costo de redispersar un archivo es muy alto. El tiempo utilizado es importante, y mientras se realiza esta operatoria no es posible que el usuario final de la información acceda al archivo. Por este motivo, cuando se piensa en la creación de un archivo, es importante:

1. Analizar la tasa de crecimiento posible del archivo.
2. Determinar la cantidad de nodos que optimice el uso de espacio vs. la periodicidad de la redispersión.

Otra alternativa posible es trabajar con archivos que administren el espacio de direccionamiento de manera dinámica. Esto es, no establecer *a priori* la cantidad de nodos disponibles, sino que esta crezca a medida que se insertan nuevos registros.

Surge así la necesidad de utilizar *hash* con espacio de direccionamiento dinámico. Este *hash* dispersa las claves en función de las direcciones disponibles en cada momento, y la cantidad de direcciones puede crecer, *a priori* sin límites, en función de las necesidades de cada archivo particular.

Existen diferentes alternativas de implementación para *hash* con espacio dinámico: *hash* virtual, *hash* dinámico y *hash* extensible, entre otras. En este libro, será considerada solamente la última alternativa, el *hash* extensible.

Hash extensible

El método de *hash* extensible es una alternativa de implementación para *hash* con espacio de direccionamiento dinámico. El principio del método consiste en comenzar a trabajar con un único nodo para almacenar registros e ir aumentando la cantidad de direcciones disponibles a medida que los nodos se completan.

Este método, al igual que el resto de los que trabajan con espacio dinámico, no utiliza el concepto de DE. Esto se debe a que el espacio en disco utilizado aumenta o disminuye en función de la cantidad de registros de que dispone el archivo en cada momento.

El principal problema que se tiene con los métodos dinámicos en general y con el *hash* extensible en particular es que las direcciones de nodos no están prefijadas *a priori*, y por lo tanto la función de *hash* no puede retornar una dirección fija. Entonces, es necesario cambiar la política de trabajo de la función de dispersión.

Para el método extensible, la función de *hash* retorna un *string* de bits. La cantidad de bits que retorna determina la cantidad máxima de direcciones a las que puede acceder el método.

Suponga que la función de *hash* retorna 32 bits. En ese caso es posible direccionar 2^{32} direcciones de nodos diferentes, si fuera necesario. Si se tiene en cuenta que cada dirección podría almacenar 100 registros, por ejemplo, la cantidad de claves a dispersar es importante.

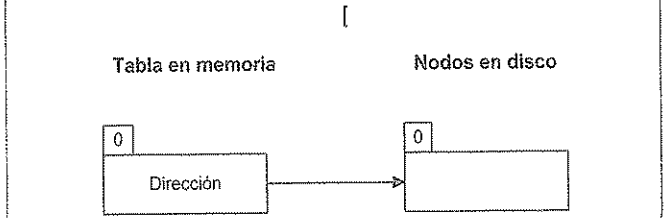
El método necesita, para su implementación, de una estructura auxiliar. Esta estructura es una tabla que se administra en memoria principal. A diferencia del *hash* asistido por tabla, esta estructura contiene la

dirección física de cada nodo. Se debe tener en cuenta que la función de *hash* no retorna una dirección física, sino una secuencia de bits. Dicha secuencia permite obtener de la tabla en memoria la dirección física del nodo para almacenar la llave. Además, la tabla será utilizada posteriormente para recuperar cada registro en un solo acceso a disco.

El tratamiento de dispersión con la política de *hash* extensible comienza con un solo nodo en disco, y una tabla que solamente contiene una dirección, la del único nodo disponible.

La Figura 8.8 presenta el estado inicial del archivo al aplicar el método. El número cero sobre la tabla indica que no es necesario ningún bit de la secuencia obtenida por la función de dispersión, para obtener la dirección física donde almacenar el registro. Considere el lector que hay una sola dirección disponible. Además, inicialmente, el nodo en disco también tiene cero.

Figura 8.8



La Tabla 8.2 muestra una secuencia de llaves a insertar en el archivo generado a partir de la Figura 8.8. Para este problema se establece que la capacidad de cada nodo es de dos registros.

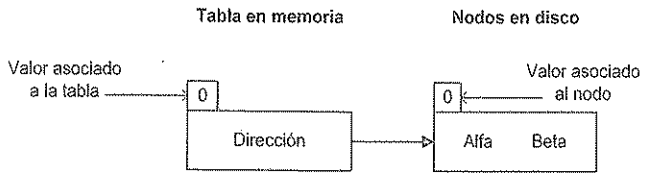
Tabla 8.2

Clave	FH(Clave)
Alfa	00.....1001
Beta	00.....0100
Gamma	00.....0010
Delta	00.....1111
Epsilon	00.....0000
Rho	00.....1011
Pi	00.....0110
Tau	00.....1101
Psi	00.....0001
Omega	00.....0111

Para realizar inserciones, el método trabaja de la siguiente forma. De acuerdo con la Tabla 8.2 se debe dispersar la clave Alfa. Se calcula la función de *hash* y se toman tantos bits como indica el valor asociado a la tabla. En este caso, dicho valor es 0; esto indica que hay una sola dirección del nodo disponible y Alfa debería insertarse en dicho nodo, siempre y cuando no genere saturación; como el nodo está vacío, Alfa se escribe en ese lugar.

Para insertar la clave Beta se procede de la misma forma, dado que nuevamente no se produce *overflow*. La Figura 8.9 muestra cómo quedan el archivo y la tabla en memoria luego de insertar Alfa y Beta.

Figura 8.9



La tercera llave a dispersar es, de acuerdo con la Tabla 8.2, Gamma. Se procede de forma similar a lo descripto anteriormente, pero ahora el único nodo disponible tiene su capacidad colmada. La inserción de Gamma produce *overflow*. La secuencia de acciones, en este punto, es presentada en la Figura 8.10 a); primero se aumenta en uno el valor asociado al nodo saturado. Luego, se genera un nuevo nodo con el mismo valor asociado al nodo saturado.

Cuando se compara el valor del nodo con el valor asociado a la tabla, se puede observar que el primero es mayor que el segundo. Esto significa que la tabla no dispone de entradas suficientes para direccionar al nuevo nodo. De hecho, la tabla tiene una celda única, y como se dispone ahora de dos nodos, hace falta generar más direcciones. En esta situación, la cantidad de celdas de la tabla se duplica, y el valor asociado a la tabla se incrementa en uno. La Figura 8.10 b) muestra cómo queda la tabla.

El valor asociado a la tabla indica la cantidad de bits que es necesario tomar de la función de *hash*. A partir de este momento es necesario tomar uno de ellos, el menos significativo. La primera celda de la tabla direcciona al nodo saturado, y la nueva celda apunta al nuevo nodo generado.

Los elementos de la cubeta saturada (Alfa y Beta) más el elemento que generó la saturación (Gamma) son redispersados entre ambos nodos de acuerdo con el valor que determina el bit menos significativo, resultante de la función de *hash*. Esta situación se presenta en la Figura 8.10 c).

Figura 8.10

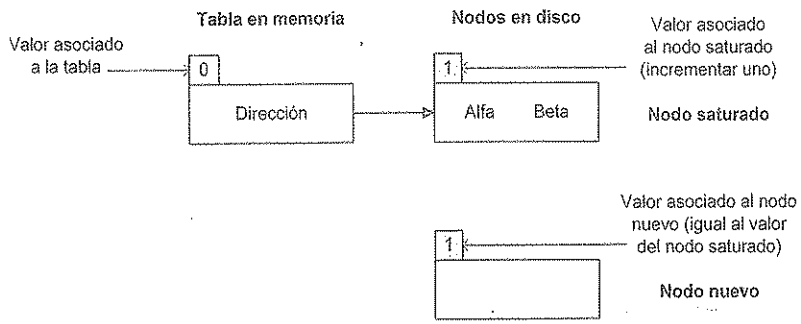


Figura 8.10 a)

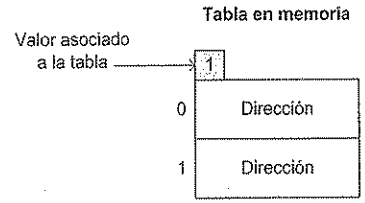


Figura 8.10 b)

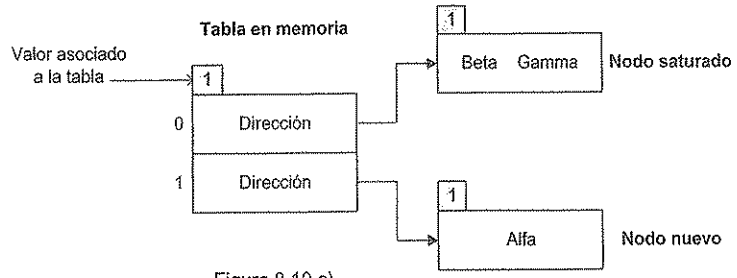


Figura 8.10 c)

continúa >>>

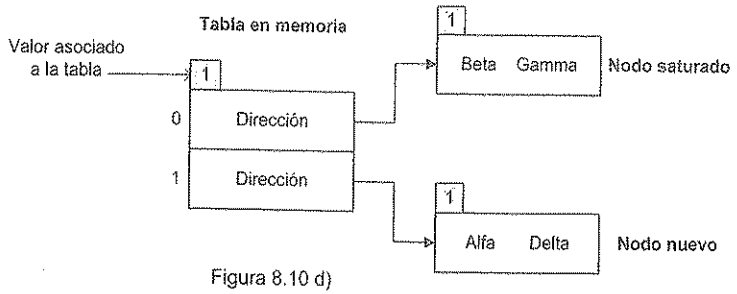


Figura 8.10 d)

La llave Delta tiene un uno en el bit menos significativo. Al direccionar el nodo correspondiente al bit en uno, no se genera *overflow*, por lo que Delta se almacena junto con Alfa [Figura 8.10 d)].

La quinta llave, Epsilon, debe ser almacenada en el nodo asociado a la celda 0 de la tabla. Dicho nodo se encuentra completo, lo que genera un nuevo *overflow*. La Figura 8.11 a) muestra el estado del nodo saturado y el nuevo nodo generado.

Como se explicó en el caso anterior, al no disponer de celdas suficientes en la tabla en memoria principal, se duplica el espacio disponible, que a partir de este momento necesita 2 bits de la función de *hash* para poder direccionar un registro [Figura 8.11 b)].

La celda de referencia 00 contiene la dirección del nodo saturado, en tanto que la celda de referencia 10 contiene la dirección del nuevo nodo.

Los registros Beta, Gamma y Epsilon son reubicados en función de sus 2 bits menos significativos [Figura 8.11 c)]. Se debe notar que el nodo asociado con las celdas de referencia 1, a partir de este momento, es referenciado por dos celdas de la tabla, las correspondientes a 01 y 11.

Figura 8.11

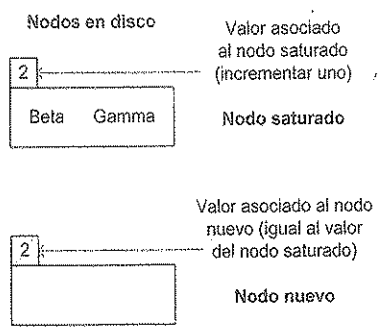


Figura 8.11 a)

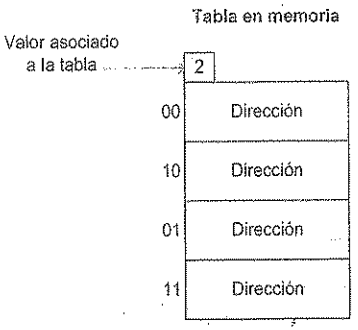


Figura 8.11 b)

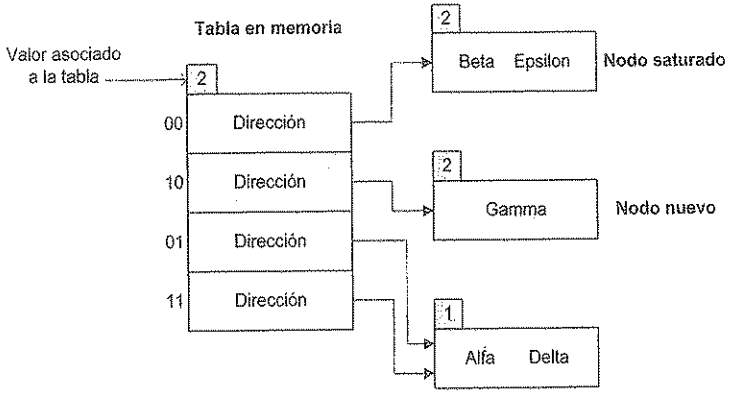
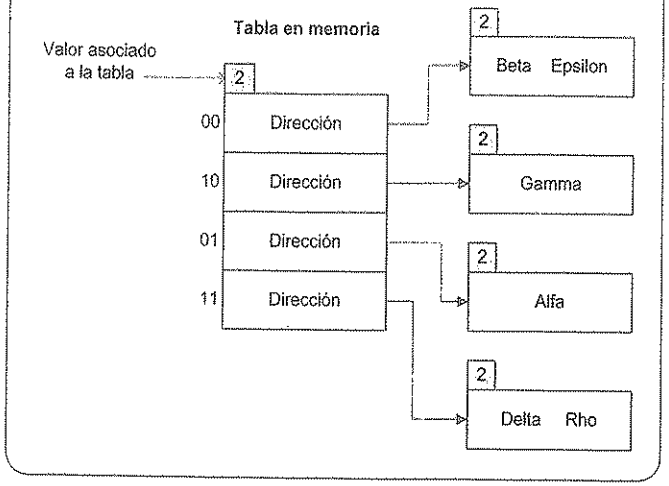


Figura 8.11 c)

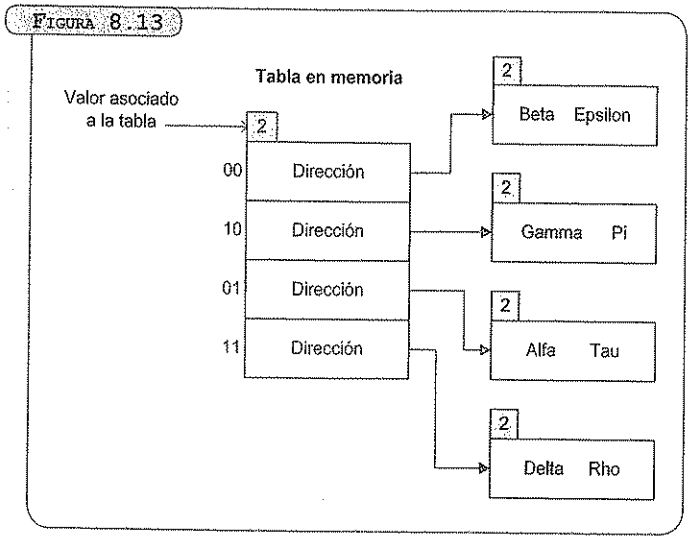
La sexta llave recibida es Rho; su dirección de almacenamiento corresponde al nodo asociado a la celda 11. Dicho nodo está completo, Alfa y Delta lo ocupan. Nuevamente, se genera una situación de saturación y el nodo es dividido. Ahora, el valor asociado a ambos nodos coincide con el valor asociado a la tabla en memoria. Este caso significa que la tabla posee direcciones suficientes para direccionar al nuevo nodo; por lo tanto, la cantidad de celdas no debe ser duplicada. Puede observarse en la Figura 8.11 c) que hay dos direcciones apuntando al mismo nodo. La Figura 8.12 muestra cómo queda el ejemplo luego de insertar Rho.

Figura 8.12



En el ejemplo planteado a partir de las llaves de la Tabla 8.2 quedan, aún, elementos para insertar. La clave siguiente es Pi, la función de *hash* retorna en sus 2 bits menos significativos 10, y el nodo tiene capacidad suficiente para almacenar la clave. Lo mismo ocurre con la llave Tau. En la Figura 8.13 se representa esta situación.

Figura 8.13



La llave Psi se direcciona al nodo correspondiente a la celda 01, la cual produce saturación. La Figura 8.14 muestra el proceso de inserción de la clave en el archivo.

Figura 8.14

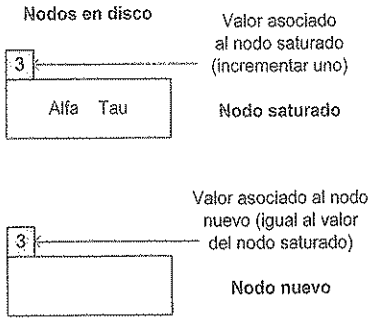


Figura 8.14 a)

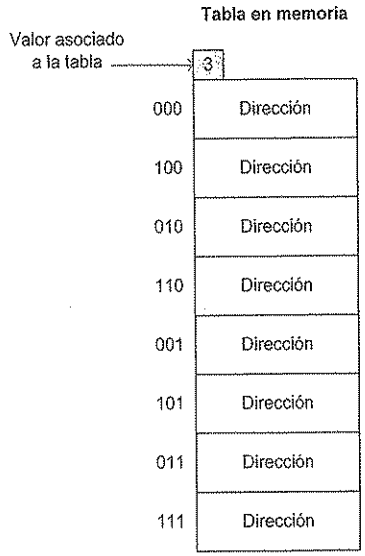


Figura 8.14 b)

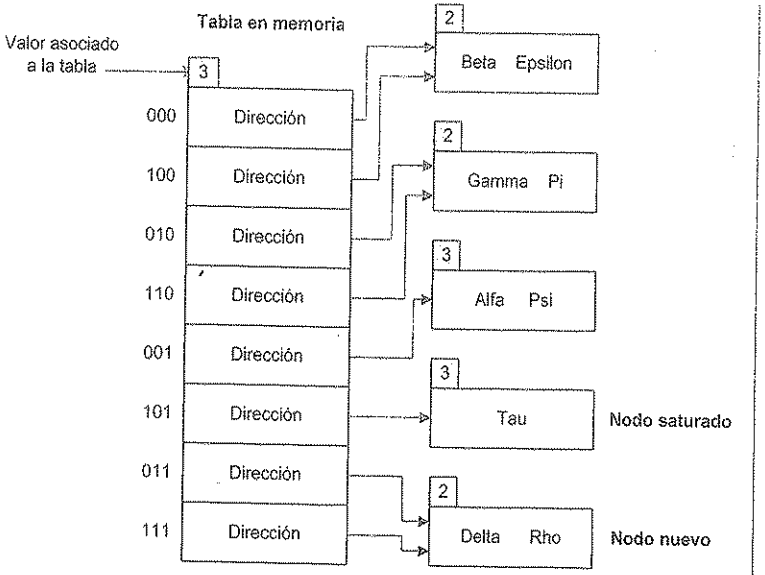
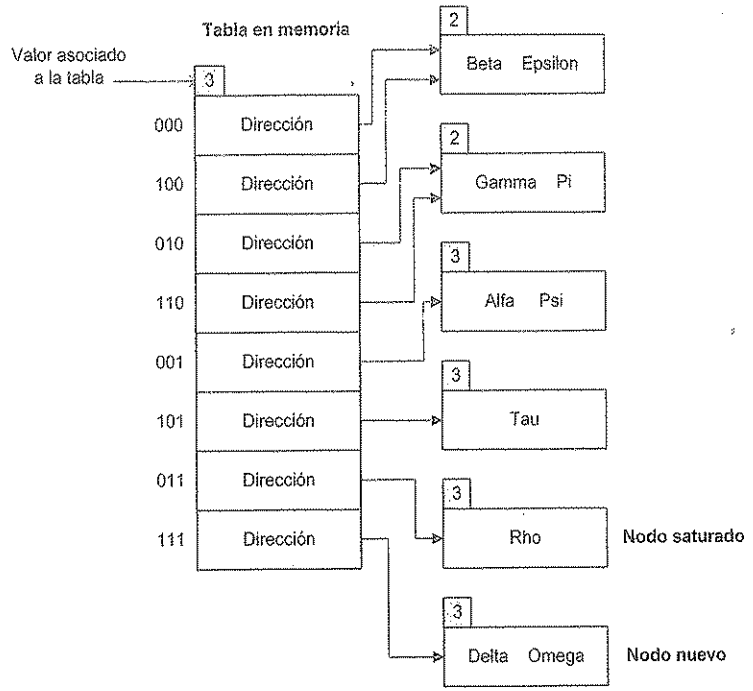


Figura 8.14 c)

La última llave, Omega, se direcciona al nodo correspondiente a la celda 111; nuevamente hay saturación, y la tabla en memoria y los nodos en disco quedan como se presenta en la Figura 8.15.

Figura 8.15



Resumiendo, el método de *hash* extensible trabaja según las siguientes pautas:

- Se utilizan solo los bits necesarios de acuerdo con cada instancia del archivo.
- Los bits tomados forman la dirección del nodo que se utilizará.
- Si se intenta insertar en un nodo lleno, deben reubicarse todos los registros allí contenidos entre el nodo viejo y el nuevo; para ello se toma 1 bit más.
- La tabla tendrá tantas entradas (direcciones de nodos) como 2^i, siendo i el número de bits actuales para el sistema.

El proceso de búsqueda asegura encontrar cada registro en un solo acceso. Se calcula la secuencia de bits para la llave, se toman tantos bits de esa llave como indique el valor asociado a la tabla, y la dirección del nodo contenida en la celda respectiva debería contener el registro buscado. En caso de no encontrar el registro en dicho nodo, el elemento no forma parte del archivo de datos.

Conclusiones

Los métodos de búsqueda de información en un archivo presentan ventajas y desventajas en cada caso. Como se discutió a lo largo de toda esta sección de archivos, la búsqueda secuencial de información tiene una *performance* de búsqueda muy deficiente, pero el archivo generado no necesita mayor análisis en cuanto al proceso de altas y/o bajas.

También se indicó oportunamente que el mayor porcentaje de operaciones sobre los archivos son de consultas; por lo tanto, es necesario en una primera etapa maximizar la eficiencia del proceso de búsqueda. Por este motivo, se analizaron alternativas que contemplan la generación de índices y su implantación mediante árboles.

Los árboles balanceados representaron una solución aceptable en términos de eficiencia, reduciendo el número de accesos a disco. Para árboles balanceados se discutieron tres alternativas: *B*, *B** y *B+*. Las dos primeras plantean el mismo comportamiento, pero se diferencian en que *B** completa más los nodos. La variante *B+* presenta una ventaja adicional, ya que permite no solo la búsqueda eficiente sino también el recorrido secuencial del archivo.

No obstante, para aquellos problemas donde la eficiencia de búsqueda debe ser extrema –y, por ende, se debe realizar un solo acceso para recuperar la información–, se dispone del método de dispersión.

Como se analizó en este capítulo, la dispersión con espacio de direccionamiento estático asegura (en un gran porcentaje de los casos) encontrar los registros buscados con un solo acceso a disco.

Si se desea asegurar siempre un acceso a disco para recuperar la información, la variante de *hash* extensible, que utiliza espacio de direccionamiento dinámico, lo logra. El costo que debe asumirse con esta técnica es mayor procesamiento cuando una inserción genera *overflow*.

La siguiente tabla resume los diferentes métodos de gestión de archivos.

Organización	Acceso a un registro por clave primaria	Acceso a todos los registros por clave primaria
Ninguna	Muy ineficiente	Muy ineficiente
Secuencial	Poco eficiente	Eficiente
Secuencial indizada	Muy eficiente	El más eficiente
Hash	El más eficiente	Muy ineficiente

Cuestionario del capítulo

1. ¿Cuáles son las ventajas de utilizar una política de *hash* para organizar un archivo de datos?
2. ¿Es posible que un archivo se organice mediante *hashing* si se utiliza para dispersar una clave primaria o candidata? ¿Qué sucede si la clave es secundaria?
3. ¿Cuáles son las propiedades básicas que debe cubrir una función de *hash*?
4. ¿Qué parámetros afectan la eficiencia del método de dispersión?
5. Conceptualmente, ¿qué diferencia existe entre colisión y saturación?
6. ¿Qué mide la DE de un archivo? ¿Cómo se calcula?
7. ¿Por qué son necesarios los métodos de tratamiento de saturación? ¿En qué casos son aplicados?
8. ¿Qué aporta el *hash* asistido por tabla?
9. ¿Cuál es la necesidad de disponer políticas de *hash* con espacio de direccionamiento dinámico?
10. ¿Cuál propiedad establecida para los métodos de *hash* no es cumplida por las políticas de *hash* con espacio de direccionamiento dinámico?

Ejercitación

1. Según un resultado matemático sorprendente llamado la paradoja del cumpleaños, si hay más de 23 personas en una habitación, entonces existe más de 50% de posibilidades de que dos de ellas tengan la misma fecha de cumpleaños. ¿En qué forma la paradoja del cumpleaños ilustra un importante problema asociado con la dispersión?
2. Suponga que se tiene un archivo donde el 85% de los accesos son generados por 20% de los registros, y se genera un archivo con una DE de 0.8 y con nodos con capacidad para cinco registros. Cuando se genera el archivo, se dispersa primero 20% de los registros más utilizados. Luego de dispersar a estos registros y antes de ubicar a los demás, ¿cuál es la DE del archivo parcialmente lleno? Con esta DE, calcule el porcentaje de 20% activo que sería registro en saturación. Discuta los resultados.
3. Suponga que debe organizarse un archivo. La siguiente tabla presenta las claves a dispersar y el resultado de aplicar una

determinada función de *hash*, sabiendo que cada nodo tiene capacidad para dos registros. Muestre cómo se genera el archivo suponiendo que la saturación se trata con:

a. Política de saturación progresiva.

b. Política de saturación progresiva encadenada.

Indique en cada caso cuántos accesos son necesarios para recuperar la información.

Clave	Función de *hash*
Roberto	20
María	21
Alberto	22
Sandra	20
Victoria	22
Ignacio	20
Juan	21

4. Para las claves siguientes, realice el proceso de dispersión mediante el método de *hashing* extensible, sabiendo que cada nodo tiene capacidad para dos registros. El número natural indica el orden de llegada de las claves.

1	Nano	0000 1010	2	Mega	0101 0001
3	Micro	1010 1100	4	Giga	0001 1110
5	Mili	0110 0011	6	Tera	1100 0110
7	Kilo	1110 0101	8	Peta	0011 1001

5. Suponga que se está dispersando un archivo de *N* registros, y que se utiliza el método de área de desbordes por separado para la administración de colisiones. Además, se sabe que cada nodo tiene capacidad para dos registros. A continuación, se detalla el conjunto de elementos que llegan al archivo y se indica, además, la dirección del nodo donde debe residir.

   Muestre cómo se completa el archivo, sabiendo que el área de desbordes por separado comienza en el nodo con dirección 105.

Orden	Clave	Cubeta	Orden	Clave	Cubeta
1	Alfa	20	7	Omega	20
2	Beta	21	8	Pi	21
3	Gamma	21	9	Rho	20
4	Delta	22	10	Tita	20
5	Epsilon	20	11	Psi	20
6	Fi	21	12	Tau	21

III

Modelado de datos

CAPÍTULO 9

Introducción al modelado de datos

Objetivo

La parte esencial de un curso de BD es el modelado de BD. Estas representan una de las componentes esenciales de los **Sistemas de Información (SI)**.

Las primeras BD, diseñadas en la década de 1960, fueron generadas utilizando técnicas más artesanales que sistemáticas, que se valían de herramientas muy rudimentarias. Esta situación ha cambiado considerablemente. Los métodos y modelos de diseño de BD han evolucionado paralelamente a la tecnología, que permite generar SI cada vez más completos y complejos.

A medida que la tecnología de BD ha evolucionado, se han desarrollado metodologías y técnicas de diseño.

Actualmente, existe consenso sobre la forma en que el proceso de diseño de datos puede ser dividido en diferentes fases.

Dos de los propósitos fundamentales del modelado de datos son: ayudar a comprender la semántica de los datos y facilitar la comunicación de los requerimientos del SI.

En este capítulo, se describen las características generales de modelado y las abstracciones que se pueden generar para desarrollar modelos de datos.

Se presentan además las definiciones de diferentes tipos de modelos, con énfasis en el modelo entidad relación o entidad interrelación, para luego desarrollar los conceptos generales vinculados con modelos de implementación física, como el modelo relacional.

Modelado y abstracciones

La construcción de un modelo de datos requiere responder un conjunto de preguntas que establecen las necesidades básicas del cliente-usuario respecto del SI que necesita. Al formular las respuestas, los diseñadores del modelo de datos descubren la forma de interpretar los datos de la organización del cliente (semántica), las relaciones entre dichos datos y restricciones de funcionamiento que se plantean.

Un **modelo de datos** es un conjunto de herramientas conceptuales que permiten describir la información que es necesario administrar para un SI, las relaciones existentes entre estos datos, la semántica asociada y las restricciones de consistencia.

Los modelos de datos sirven para hacer más fácil la comprensión de los datos de una organización. Se modela para:

1. Obtener la perspectiva de cada actor asociado al problema (clientes y usuarios).
2. Obtener la naturaleza y necesidad de cada dato.
3. Observar cómo cada actor utiliza cada dato.

Los modelos de datos son medios para describir la realidad. Los diseñadores usan los modelos para construir esquemas, los que sintetizan la realidad del problema.

Para construir un modelo se utilizan tres clases de abstracciones: clasificación, agregación y generalización. Estas abstracciones ayudan al diseñador a interpretar, clasificar y generar modelos de la realidad.

El diseñador debe interpretar el mundo real, y abstraer de este los objetos y las relaciones entre ellos.

Abstracciones

La abstracción es un proceso que permite seleccionar algunas características de un conjunto de objetos del mundo real, dejando de lado aquellos rasgos que no son de interés. Así, dado un elemento cualquiera, una silla por ejemplo, las características de acuerdo con la visión del cliente pueden ser el material con que está fabricada, el tipo de respaldo, si posee o no ruedas, entre otras. Para otro cliente, las características relevantes pueden ser el tipo de pata o la comodidad.

El diseño conceptual de BD utiliza tres tipos de abstracciones: clasificación, agregación y generalización.

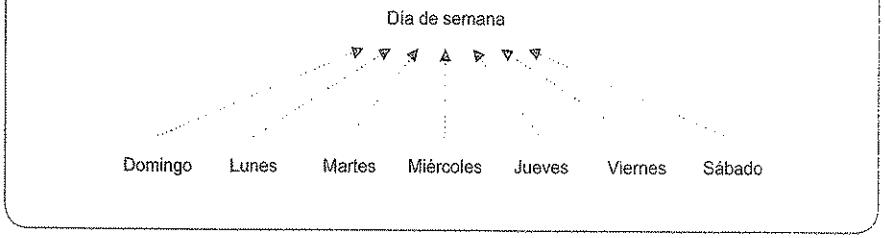
Abstracción de clasificación

La abstracción de clasificación se utiliza para definir una clase. Una clase es una declaración o abstracción de objetos, lo que significa que es la definición de un objeto. Una clase se origina a partir de las características comunes que tienen los objetos que la componen.

Si se define la clase día_de_semana, se piensa en las características propias de cada día; por ejemplo: día laborable o no laborable, hora de comienzo y fin de actividades para cada día, entre otras.

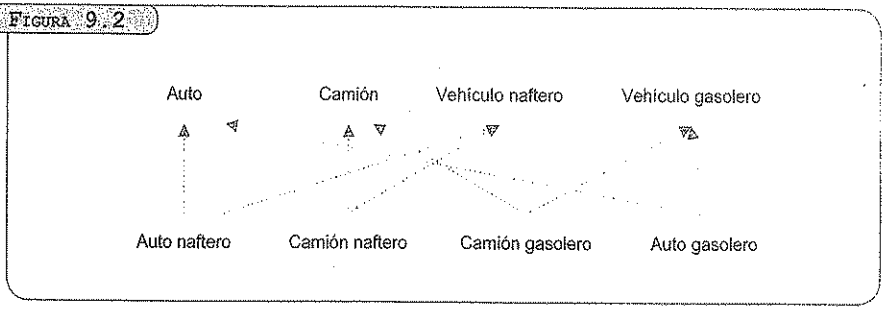
En la Figura 9.1, se representa un ejemplo de abstracción de clasificación. Puede observarse el árbol, que representa la abstracción de clasificación: la clase se define al tope del árbol, y los elementos de la clase, como hojas unidas al tope por líneas punteadas. Cada línea punteada indica que un nodo hoja **es un miembro** de la clase que está definida en la raíz.

Figura 9.1



Un objeto puede clasificarse de varias formas. La Figura 9.2 presenta los objetos auto naftero, auto gasolero, camión naftero, camión gasolero. Es posible observar que el elemento auto naftero corresponde a dos clases posibles, auto y vehículo naftero.

Figura 9.2

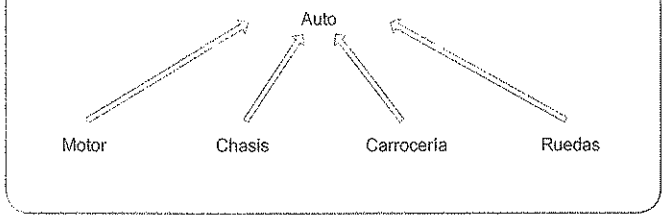


Abstracción de agregación

Una abstracción de agregación define una nueva clase a partir de un conjunto de otras clases que representan sus partes componentes.

Continuando el ejemplo del automóvil, este objeto está compuesto por motor, chasis, carrocería y ruedas, entre otros elementos. La Figura 9.3 muestra la representación gráfica del problema. La raíz del árbol es la clase creada por agregación de las clases que están en las hojas del árbol. En este caso, las clases hoja representan una relación **es_parte_de** de la clase raíz.

FIGURA 9.3



Las abstracciones vistas hasta el momento, clasificación y agregación, representan dos formas de pensamiento muy utilizadas para la construcción de las estructuras de datos de las BD.

En los capítulos siguientes, cuando se profundice la construcción de modelos, se ejemplificará con más detalle el uso de abstracciones.

La clasificación permite identificar los campos o atributos de los elementos individuales de datos. La agregación, en tanto, permite agrupar los campos o atributos formando registros de datos. Deben interpretarse los conceptos de campo y registro de la misma forma que se planteó en los capítulos iniciales de este libro.

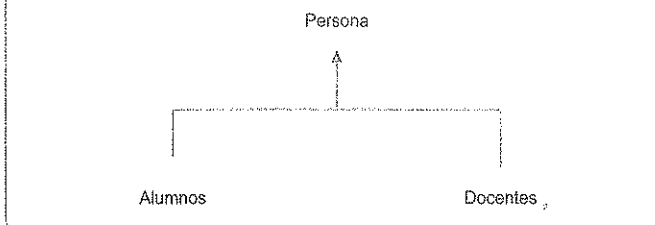
Abstracción de generalización

Una abstracción de generalización define una relación de subconjunto entre los elementos de dos o más clases.

La Figura 9.4 representa una abstracción de generalización: persona representa la clase formada por todos los individuos del problema. Esta generalización puede especializarse con las clases alumno y docente. En este ejemplo, alumno tendrá solamente los rasgos representativos para ellos (por ejemplo, número de alumno y colegio de procedencia, entre otros), y docente contemplará información propia

de estos (por ejemplo, título máximo obtenido, CUIL y jerarquía docente, entre otros). Los rasgos comunes entre alumnos y docentes se definen directamente como rasgos de la clase persona.

FIGURA 9.4



En una generalización, las especialidades (hijos) heredan las características del padre. La Figura 9.4 muestra gráficamente el problema. En este caso, la forma de representación de las abstracciones de generalización se realiza a través de la relación **es_un** entre los hijos (docentes y alumnos) y el padre (persona).

Propiedades de correspondencia entre clases

Las abstracciones de agregación y generalización generan correspondencias entre las clases que las conforman, no así la abstracción de clasificación.

Agregación binaria

La agregación binaria es la correspondencia existente entre dos clases diferentes. Por ejemplo, *nacido_en* es una agregación binaria entre persona y ciudad. *Nacido_en* establece una correspondencia entre una persona particular y la localidad donde esa persona ha nacido.

Otros ejemplos pueden ser: *vive_en* como una agregación nuevamente entre persona y ciudad, o *hecho_de*, agregación entre producto y materia prima.

A partir de esta correspondencia puede definirse un nuevo concepto, la cardinalidad entre las clases.

Se define la cardinalidad como el nivel de correspondencia existente entre los objetos de la clase. Así, es posible definir el nivel mínimo de correspondencia (cardinalidad mínima) y el nivel máximo de correspondencia (cardinalidad máxima) entre los objetos de la clase.

En el ejemplo *nacido_en*:

Card min (persona, nacido_en) = 1 significa que, como mínimo, una persona debe haber nacido en una ciudad.

Card max (persona, nacido_en) = 1 significa que una persona no puede haber nacido en más de una ciudad.

Card min (localidad, nacido_en) = 0 significa que puede existir una ciudad donde no haya nacido persona alguna.

Card max (localidad, nacido_en) = n significa que en una ciudad han nacido varias personas.

Cuando Card min es 0, indica que en la agregación hay una participación opcional, en tanto que la Card min en 1 indica participación obligatoria de la agregación binaria.

Por su parte, la cardinalidad máxima denota cuatro tipos de agregaciones:

- Si la card max(a, b) = 1, card max(c, b) = 1, se dice que la cardinalidad es uno a uno (en el ejemplo anterior, a = persona, b = nacido_en, c = ciudad).
- Si la card max(a, b) = 1, card max(c, b) = n, se dice que la cardinalidad es uno a muchos.
- Si la card max(a, b) = n, card max(c, b) = 1, se dice que la cardinalidad es muchos a uno.
- Si la card max(a, b) = n, card max(c, b) = n, se dice que la cardinalidad es muchos a muchos.

Queda como ejercicio resolver las cardinalidades de las agregaciones vive_en y hecho_de.

Debe notarse que puede ocurrir que determinar el valor de las cardinalidades no sea una tarea simple. Dichos valores deben extraerse del problema sobre el cual se realiza la abstracción. En caso de no disponer de suficiente información, se deberá profundizar en obtener conocimiento del problema.

Existen otros tipos de agregaciones, denominadas *n*-arias, que escapan al contenido de este libro.

Generalización

Así como se define la correspondencia o agregaciones binarias entre clases, es posible definir la cobertura de una generalización.

La propiedad de cobertura define el grado de relación entre padres e hijos. Existen dos coberturas a definir. La primera de ellas determina si la relación entre padres e hijos es total o parcial. Una cobertura es total cuando cada elemento del padre está contenido en alguno de los hijos. La cobertura es parcial cuando pueden existir elementos del padre que no se instancian sobre los hijos.

Tomando por ejemplo la Figura 9.4, puede considerarse que, para ese problema, el universo de las personas está formado por docentes o alumnos; entonces la cobertura es total.

Si se analizara el problema considerando además al personal de apoyo, en este caso la cobertura sería parcial. Debe notarse que la cobertura depende directamente del problema sobre el cual se realiza la abstracción.

Por otra parte, la cobertura puede ser exclusiva o superpuesta. En este caso, lo que se analiza es si un elemento del padre puede o no estar en más de un hijo. Si solo puede estar en un hijo, la cobertura es exclusiva, en tanto que si puede estar en varios, se denomina superpuesta.

En el ejemplo de la Figura 9.4, si para el problema analizado un docente no puede ser alumno, y viceversa, la cobertura es exclusiva. Por otro lado, si en el problema existen alumnos que son docentes (por ejemplo, ayudantes alumnos), la cobertura es superpuesta.

Técnicas de modelado (ER, OO)

Un modelo de datos es una serie de conceptos que puede utilizarse para describir un conjunto de datos y las operaciones para administrarlos. Los modelos de datos se construyen, en general, utilizando mecanismos de abstracción, y se describen mediante representaciones gráficas que tienen una sintaxis y una semántica asociadas.

Genéricamente, los modelos de datos se dividen en tres grandes grupos: modelos basados en objetos, modelos basados en registros y modelos de datos físicos (que no son parte de este libro).

Modelos basados en objetos

Los modelos basados en objetos se utilizan para describir los datos de acuerdo con la visión que cada usuario tiene respecto de la BD. Así, estos modelos permiten generar vistas de las necesidades que posee cada actor respecto de un problema. Posteriormente, a partir de todas y cada una de las vistas producidas, es posible generar

un único modelo final que resuma las necesidades de cada actor y que, por consiguiente, permita generar una única BD que administre la información para el problema.

Existen diferentes modelos basados en objetos; los más difundidos son:

- Modelo entidad relación.
- Modelo orientado a objetos.
- Modelo de datos semántico.
- Modelo de datos funcional.

Cabe acotar que, para propósito de este libro, se definirá con detalle el modelo entidad relación. El modelo orientado a objetos escapa al contenido de este libro.

Modelos basados en registros

Los modelos basados en registros nuevamente permiten describir los datos desde la perspectiva de cada usuario. La diferencia básica con los modelos basados en objetos radica en que los modelos basados en registros permiten especificar la estructura lógica completa de la BD.

Estos modelos llevan su nombre debido a que la BD se estructura como registros de longitud fija, conformados por campos o atributos.

El modelo relacional es actualmente el ejemplo más representativo del modelo basado en registros. Históricamente existieron los modelos de red y jerárquico, hoy en desuso.

Introducción al modelo entidad relación

La técnica de modelado de datos más ampliamente utilizada es el modelo **Entidad Relación (ER)** o **Entidad Interrelación (EI)**, el cual presenta las entidades de datos, sus atributos asociados y las relaciones entre estas entidades. Esta aproximación fue presentada por primera vez por Chen en 1976, modificada y ampliada por Codd en 1979, y por otros autores a lo largo de la década del ochenta, hasta que, en 1988, el ANSI (American National Standards Institute) la seleccionó como el modelo estándar, para la generación de los diccionarios de datos asociados a los SI.

El modelo ER se basa en la concepción del mundo real como un conjunto de objetos llamados entidades y las relaciones existentes entre dichas entidades.

Las entidades representan un elemento de utilidad para el problema, y cada elemento debe ser distinguible del resto. Por ejemplo, el elemento auto o el elemento persona. Se debe considerar que cada auto y cada persona es distinguible del resto de los autos y personas.

Cada entidad está conformada por un conjunto de atributos que la caracterizan. Así, por ejemplo, una persona tiene como características (atributos) su nombre, su edad, su teléfono, etcétera. Para un auto, las características pueden ser su modelo, marca y color, entre otras.

Por último, una relación establece un nexo entre entidades. En el ejemplo planteado, una persona puede ser propietaria de un auto; esto significa que ambas entidades están relacionadas por una característica de "posesión".

En los capítulos siguientes, se describe con detalle al modelo ER como herramienta para la definición del modelo de datos de un problema.

Como ya se indicó, el modelo ER es conocido, además, como modelo EI. La diferencia en la simbología tiene su origen en la forma de concebir la palabra "relación"; algunos autores optan por usar "modelo EI" para evitar confusiones con el modelo basado en registros, denominado modelo relacional.

El modelo ER no llega a tener una implementación física. Esto es, cualquier diagrama obtenido que represente el modelo de datos de un problema tiene precisamente dicha característica, es un diagrama.

Una vez generado el modelo de datos ER definitivo para un problema, deberá utilizarse otra forma de modelado que permita obtener su implementación física.

A lo largo de este apartado, se utilizará el modelo relacional para implementar físicamente un modelo ER.

Modelado e ingeniería de *software*

La **Ingeniería de *Software* (IS)** es una disciplina que abarca todos los aspectos de la producción de *software*, desde sus etapas iniciales (elicitación de requerimientos) hasta la etapa de mantenimiento del *software* generado, conocido como ciclo de vida del desarrollo de *software*. Dentro de este ciclo de vida están incluidas la modelización de datos del problema, la generación de la BD respectiva y su posterior utilización. De esta forma, el modelado y uso de una BD es una parte más del proceso abarcado por la IS.

Sin profundizar en el tema, es posible resumir la IS como una serie de etapas o actividades: comprender un problema, analizarlo, diseñar

una solución para el problema, implementar dicha solución, entregar el producto final obtenido y posteriormente ajustarlo a los cambios del medio donde se haya insertado.

Históricamente, se considera al modelado de datos como una de las actividades propias del diseño del problema. Si bien esta consideración es perfectamente válida, con el transcurrir del tiempo, los analistas no la respetaron. Así, durante fases tempranas, cuando se trata de comprender un problema, el analista comienza a idear, imaginar o esbozar cómo será el modelo de datos. Este premodelo se continúa refinando y mejorando para que, durante la etapa de diseño, quede efectivamente establecido.

La metodología propuesta en este libro consiste en construir un modelo de datos en etapas. La primera etapa permitirá construir lo que se denomina modelo conceptual. Las herramientas disponibles para tal fin harán posible generar un modelo que sirva para mejorar la comunicación entre el analista y el cliente/usuario. Una vez cumplida esta etapa, el modelo será transformado de una percepción conceptual a una percepción lógica. Durante esa transformación, se tendrán en cuenta determinadas reglas, que ayudarán a definir el modelo en términos más cercanos a las computadoras.

La última etapa, en tanto, que se encargará de refinar aun más el modelo, genera una implementación física que permitirá instanciarlo utilizando un SGBD.

Los tres capítulos siguientes muestran rasgos, características y elementos válidos que permitirán la construcción de modelos de datos con la evolución planteada. En estas tres etapas –conceptual, lógica y física–, se parte de las siguientes premisas:

1. **Modelado conceptual:** es desarrollado durante la etapa de adquisición de conocimiento del problema; el analista se independiza del tipo de SGBD a utilizar y, por consiguiente, del producto de mercado. Así, el modelo conceptual se desarrolla independientemente de su implementación final (relacional, de red, jerárquico u OO).
2. **Modelo lógico:** el analista debe determinar el tipo de SGBD, debido a que las decisiones que debe tomar dependen de esa elección.
3. **Modelo físico:** es necesario tomar decisiones específicas. Estas últimas tienen que ver con el producto de mercado a utilizar, es decir, el SGBD específico.

La siguiente tabla resume lo anteriormente expuesto.

Modelo	Tipo de SGBD	SGBD específico
Conceptual	No debe decidirse	No debe decidirse
Lógico	Debe decidirse	No debe decidirse
Físico	Debe decidirse	Debe decidirse

Introducción al modelo relacional

El modelo relacional utiliza un conjunto de tablas para representar las entidades y las relaciones existentes, definidas en el modelo ER. Este modelo es actualmente el más utilizado en aplicaciones que implanten soluciones a SI.

La estructura básica del modelo relacional es la tabla. Cada tabla tiene una estructura conformada por columnas o atributos que representan, básicamente, los mismos atributos definidos para el modelo ER. Cada fila (registro) de la tabla se denomina tupla o relación.

Hay una importante correspondencia entre el concepto de tabla y el concepto de relación en términos matemáticos. De esta correspondencia se deriva el nombre de modelo relacional.

En el Capítulo 12, se definirán con mayor detalle las características del modelo relacional, y cómo obtenerlo a partir de transformaciones del modelo ER.

CAPÍTULO 10

Modelado entidad relación conceptual

Objetivo

Los modelos de datos conceptuales son medios para representar la información de un problema en un alto nivel de abstracción.

A partir de la definición de modelos de datos conceptuales es posible captar las necesidades del cliente/usuario respecto del SI que necesita. Esta descripción de la realidad permite mejorar la interacción usuario-desarrollador, dado que disminuye la brecha existente entre la realidad del problema a resolver y el sistema a desarrollar.

Los modelos conceptuales ER utilizan las bases planteadas por Chen en 1976, para construir la representación básica de las estructuras de datos que darán, posteriormente, soporte a la BD a construir.

En este capítulo, se desarrollan los conceptos necesarios para generar un modelo conceptual ER, con ejemplos de cada una de las situaciones generadas. Para generar los modelos se utilizará una herramienta denominada CASER, la cual fue desarrollada como trabajo de investigación por docentes y alumnos relacionados con la asignatura Introducción a las Bases de Datos, de la Facultad de Informática de la Universidad Nacional de La Plata.

Esta herramienta, de uso libre, permite generar modelos conceptuales de datos, con la metodología, la sintaxis y la semántica definidas en este capítulo.

La creación de la herramienta se debe a que, en la actualidad, no se dispone de ningún *software* que sea de utilidad para la definición de modelos conceptuales en términos estrictamente académicos.

Características del modelo conceptual

Un modelo conceptual debe poseer cuatro características o propiedades básicas. Estas son (por orden alfabético, no de importancia):

- **Expresividad**, es decir, capturar y presentar de la mejor forma posible la semántica de los datos del problema a resolver.
- **Formalidad**, que requiere que cada elemento representado en el modelo sea preciso y bien definido, con una sola interpretación posible. Esta formalidad es comparable a la formalidad matemática.
- **Minimalidad**, característica que establece que cada elemento del modelo conceptual tiene una única forma de representación posible y no puede expresarse mediante otros conceptos.
- **Simplicidad**, que establece que el modelo debe ser fácil de entender por el cliente/usuario y el desarrollador.

Entre las propiedades definidas es posible analizar que algunas de ellas son opuestas. Por ejemplo, un modelo expresivo (rico en conceptos) puede no ser simple; es decir, en pos de expresar mejor los datos de un problema se puede perder simpleza.

Por último, sería deseable incorporar un rasgo adicional: que el modelo sea fácil de leer. Este concepto podría considerarse un corolario de todas las propiedades anteriores. Un modelo expresivo, formal, mínimo y simple debería ser fácil de leer.

Componentes del modelo conceptual

En 1976, Chen planteó tres constructores básicos para generar el modelo de datos ER. Estos tres constructores son: entidades, relaciones y atributos. A lo largo de este apartado, se desarrollarán con detalle estos tres conceptos y sus características asociadas. Además, serán generados una serie de ejemplos que ilustren su uso y comportamiento. Nuevamente, para los ejemplos de este apartado se utilizará CASER, herramienta desarrollada en conjunto por los mismos autores de este libro.

Entidades

Una entidad representa un elemento u objeto del mundo real con identidad, es decir, se diferencia unívocamente de cualquier otro objeto o cosa, incluso siendo del mismo tipo. Ejemplos:

- Un alumno (se diferencia de cualquier otro alumno).
- Un vehículo (aunque sean de la misma marca o del mismo modelo, tendrán atributos diferentes, por ejemplo, el número de motor).
- Una materia de una carrera universitaria.

En los ejemplos anteriores se observa que hay entidades que representan objetos o elementos reales y concretos (autos, alumnos), o de existencia virtual o abstracta (materia).

A continuación, se presentan algunas entidades posibles.

Juan García, Calle 6 N° 1213, La Plata
Andrea Castro, Calle 56 N° 231, Berisso
Inés Tartaila, Montevideo N° 123, Ensenada

Renault Simbol, azul, BXE 123
Peugeot 307, verde, HHI 343
Fiat Palio, rojo, IAA 990

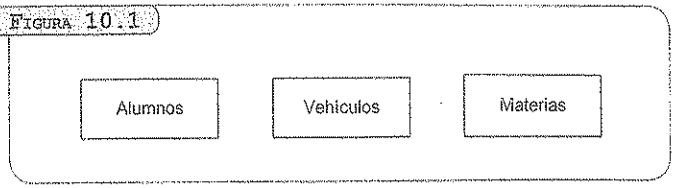
Análisis Matemático I, primer año
Introducción a las Bases de Datos, segundo año
Ingeniería de Software, tercer año

Durante el proceso de modelización de datos, es necesario identificar las entidades, pero resulta muy complejo representar cada una de esas entidades que se distinguen en el problema a resolver. Si se desea modelizar todos los alumnos de un curso numeroso, es poco viable referenciar a cada uno de ellos. Por este motivo, una vez reconocidas las entidades, el diseñador del modelo de datos busca las propiedades comunes de estas, y genera una representación que aglutina a esas entidades. Esta representación se denomina conjunto de entidades.

Un **conjunto de entidades** es una representación que, a partir de las características propias de cada entidad con propiedades comunes, se resume en un núcleo.

La forma de representar un conjunto de entidades es mediante un rectángulo. La Figura 10.1 muestra la representación de tres conjuntos de entidades que resumen a los alumnos, vehículos y materias universitarias antes descriptos.

Figura 10.1



Para reconocer las entidades del problema que permitan luego modelar los conjuntos de entidades, en general se analizan los sustantivos que conforman la especificación del problema.

Relaciones

Las relaciones representan agregaciones entre dos (binaria) o más entidades. Describen las dependencias o asociaciones entre dichas entidades. Por ejemplo, la entidad Dolores García cursa (relación) la materia Introducción a las Bases de Datos (otra entidad).

Siguiendo el análisis expresado anteriormente para las entidades, en el proceso de modelado es complejo distinguir y representar cada una de las relaciones. Aquellas que tienen propiedades comunes se aglutinan en lo que se denomina conjunto de relaciones.

Un **conjunto de relaciones** es una representación que, a partir de las características propias de cada relación existente entre dos entidades, las resume en un núcleo.

La forma de representar un conjunto de relaciones es mediante un rombo. La Figura 10.2 a) muestra la representación del conjunto de relaciones existente entre el conjunto de entidades alumnos y materias. La Figura 10.2 b) de dicho gráfico muestra el conjunto de relaciones propiedad, existente entre los conjuntos de entidades vehículo y propietario.

Figura 10.2

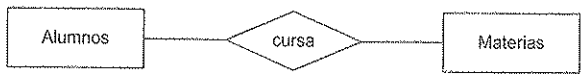


Figura 10.2 a)

Figura 10.2 b)

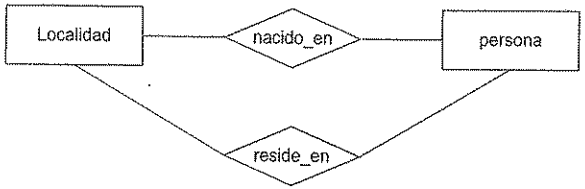


Figura 10.2 c)

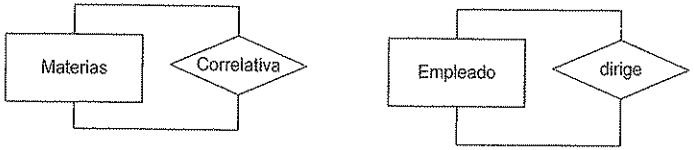


Figura 10.2 d)

Para reconocer las relaciones del problema, que permitan luego modelar los conjuntos de relaciones en general, se analizan los verbos que conforman la especificación del problema.

Pueden existir más de un conjunto de relaciones entre dos conjuntos de entidades. La Figura 10.2 c) presenta un ejemplo, donde los conjuntos de entidades persona y conjuntos de entidades ciudad poseen dos conjuntos de relaciones posibles: nacido_en y reside_en. El primero de ellos relaciona cada persona con su lugar de nacimiento, en tanto que el segundo establece el nexo entre cada persona y su lugar actual de residencia.

En el resto de este libro, por razones prácticas, cada vez que se haga referencia a un conjunto de entidades o a un conjunto de relaciones, se lo denominará entidad o relación directamente.

Por último, pueden existir relaciones recursivas entre entidades. Se denomina relación recursiva a aquella relación que une dos entidades particulares del mismo conjunto. La Figura 10.2 d) presenta dos ejemplos de dicha situación. En el primero de ellos se define la relación correlatividad. Esta relación establece que para cursar una materia se deben tener aprobadas otras materias. El segundo ejemplo presenta la relación dirige o es_jefe, entre dos entidades empleados de una empresa.

En el Capítulo 9, se desarrolló el tema de abstracciones. Una de dichas abstracciones es la agregación. Cuando se definió la agregación, se incorporó una propiedad que se denominó cardinalidad. La cardinalidad define el grado de relación existente en una agregación. Este concepto está presente nuevamente, cuando se definen las relaciones existentes entre entidades. Así, cada relación (como es una agregación hasta ahora binaria) debe tener definida la cardinalidad máxima y mínima. La forma de representar esta cardinalidad se ejemplifica en la Figura 10.3. Esta figura repite los ejemplos de la figura anterior, mostrando en cada caso la cardinalidad máxima y mínima existente.

Figura 10.3

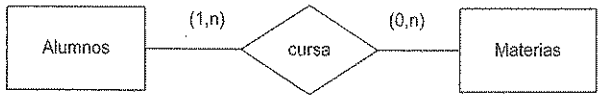


Figura 10.3 a)

Figura 10.3 b)

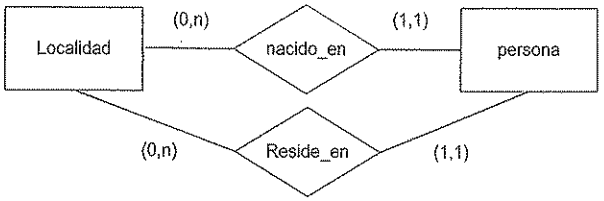


Figura 10.3. c)

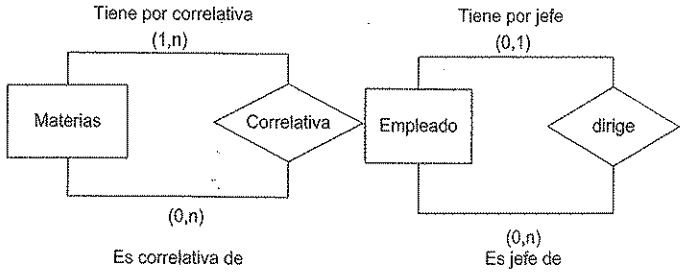


Figura 10.3 d)

La Figura 10.3 a) indica que:

CMin (alumnos, cursa) = 1
CMax (alumnos, cursa) = n

CMin (materia, cursa) = 0
CMax (materia, cursa) = n

Esto muestra que un alumno debe cursar al menos una materia (obligatoriamente), pero puede cursar varias. Además, una materia puede no ser cursada (opcional) por ningún alumno o ser cursada por varios.

Se debe notar que la cardinalidad define el grado de relación existente. Es fundamental, en la construcción de un modelo, definir con precisión la cardinalidad existente. Se podría dudar sobre algún valor de cardinalidad; en ese caso, el analista está obligado a consultar al cliente/usuario que plantea el problema, ya que la decisión tomada no debe ser unilateral.

La Figura 10.3 b) es similar al caso anterior presentado. En la Figura 10.3 c) existen dos relaciones; por lo tanto, se deben analizar más cardinalidades:

CMin (persona, nacido_en) = 1
CMax (persona, nacido_en) = 1

CMin (localidad, nacido_en) = 0
CMax (localidad, nacido_en) = n

CMin (persona, reside _en) = 1
CMax (persona, reside _en) = 1

CMin (localidad, reside _en) = 0
CMax (localidad, reside _en) = n

En este ejemplo, una persona debe haber nacido necesariamente en una sola ciudad y, además, debe residir (en términos legales) en una sola ciudad.

Por último, en la Figura 10.3 d) se debe definir la cardinalidad en una relación recursiva. La forma de proceder es similar a lo visto hasta aquí. Sin embargo, se puede observar en la figura respectiva que, asociado a la relación, además de la cardinalidad, aparece un texto o rótulo que ayuda a la interpretación de la relación. El problema que tiene la relación recursiva es cómo debe ser leída o interpretada posteriormente. En estos ejemplos, dependiendo de la forma en que se lea el resultado, se puede llegar a una interpretación errónea.

Una materia tiene por correlativa al menos a otra, pero puede tener varias; en tanto que una asignatura puede no ser correlativa de ninguna otra o puede serlo de varias. En la relación dirige, un empleado puede

no tener jefe o a lo sumo tener uno; en tanto que un empleado puede no ser jefe o tener varias personas a cargo.

En los ejemplos vinculados con la Figura 10.3, se observan relaciones uno a muchos o muchos a muchos. Quedan sin ejemplificar las relaciones uno a uno; un caso de ellas es presentado en la Figura 10.4 a).

La Figura 10.4 b) muestra un ejemplo de una relación entre tres entidades, denominada ternaria. La expresividad de la cardinalidad, en este caso, se ve afectada. Por ejemplo, suponga que una materia se cursa siempre en un aula, pero por varios alumnos a la vez; la cardinalidad de (materia, cursa) respecto de alumnos indica que varios alumnos la pueden cursar, pero la cardinalidad de (materia, cursa) respecto de aula es uno, es decir, se cursa solo en uno. En este caso, para que el modelo exprese sin errores el problema, se debe optar por la solución (0,*N*).

Para evitar este problema, algunos autores plantean solamente representar relaciones binarias entre entidades. Para ello, aportan otras herramientas que pueden utilizarse cuando se requieren soluciones ternarias. No obstante, en este libro no se hará referencia a estas soluciones. Se debe considerar que cualquier problema puede ser resuelto con agregaciones binarias, si se toman las decisiones de diseño correctas.

Figura 10.4

Figura 10.4 a)

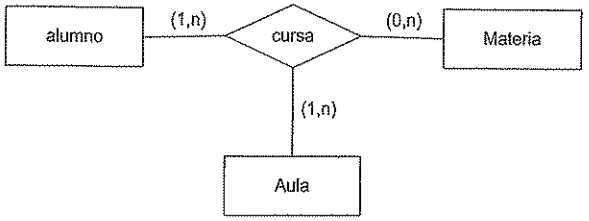


Figura 10.4 b)

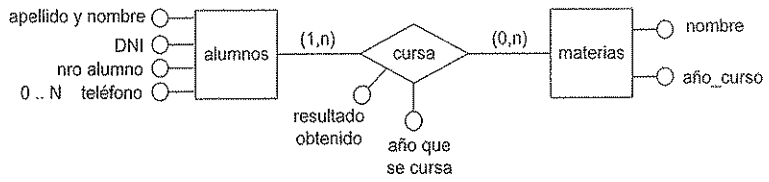
Atributos

Un atributo representa una propiedad básica de una entidad o relación. Es el equivalente a un campo de un registro. Para la entidad alumno, mediante los atributos se representan, por ejemplo, el nombre del alumno, su DNI o su número de alumno.

Las diferencias entre las entidades o relaciones se distinguen fácilmente por los atributos de cada una de ellas. En general, de la misma forma que las entidades, los atributos están indicados por sustantivos que definen el problema.

La Figura 10.5 presenta un ejemplo de atributos. Se presenta, además, la sintaxis gráfica que deberá utilizarse para definir los atributos.

Figura 10.5



Los atributos también tienen asociado el concepto de cardinalidad. Esto es, cuando se define un atributo se debe indicar si es o no obligatorio y si puede tomar más de un valor (polivalente). Las expresiones posibles son:

1..1 monovalente obligatorio; en caso de que un atributo presente esta cardinalidad, no debe ser incluida explícitamente en el modelo. Por ende, un atributo que solamente está definido por su nombre tiene asociada la cardinalidad 1..1. Atención que la cardinalidad existe y está presente; solamente en este caso no se indica en forma explícita. Un caso que ejemplifica esta situación es el DNI de un alumno.

0..1 monovalente no obligatorio.

1..n polivalente obligatorio (podrían existir múltiples valores para este atributo). Por ejemplo, el atributo título de un profesor de una asignatura.

0..n polivalente no obligatorio. Por ejemplo, el e-mail de un alumno.

En la Figura 10.5, el atributo teléfono es un atributo polivalente no obligatorio; el resto de los atributos es monovalente obligatorio.

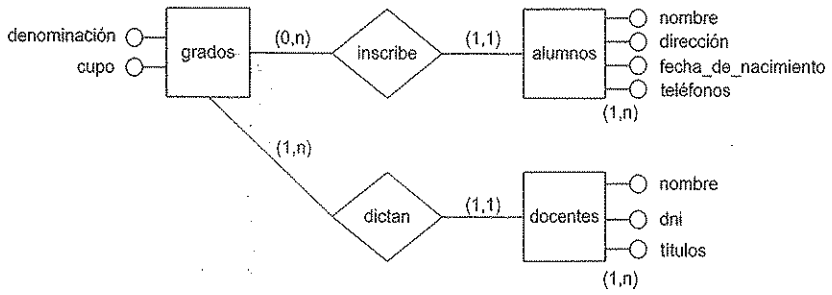
Además, cada atributo debe tener definido un dominio. El dominio de un atributo define el conjunto de valores posibles que el atributo puede tomar. El concepto de dominio está asociado al de tipo de datos. Por cuestiones de practicidad, la definición de dominio no se vuelca al modelo de datos; este concepto debe ser definido como parte de la especificación de requerimientos del problema. Recién en las etapas de modelado físico, se describe explícitamente el dominio de cada atributo.

Construcción del diagrama conceptual. Ejemplo

En este apartado, se describirán someramente dos situaciones reales a partir de las cuales serán definidos los modelos de datos respectivos.

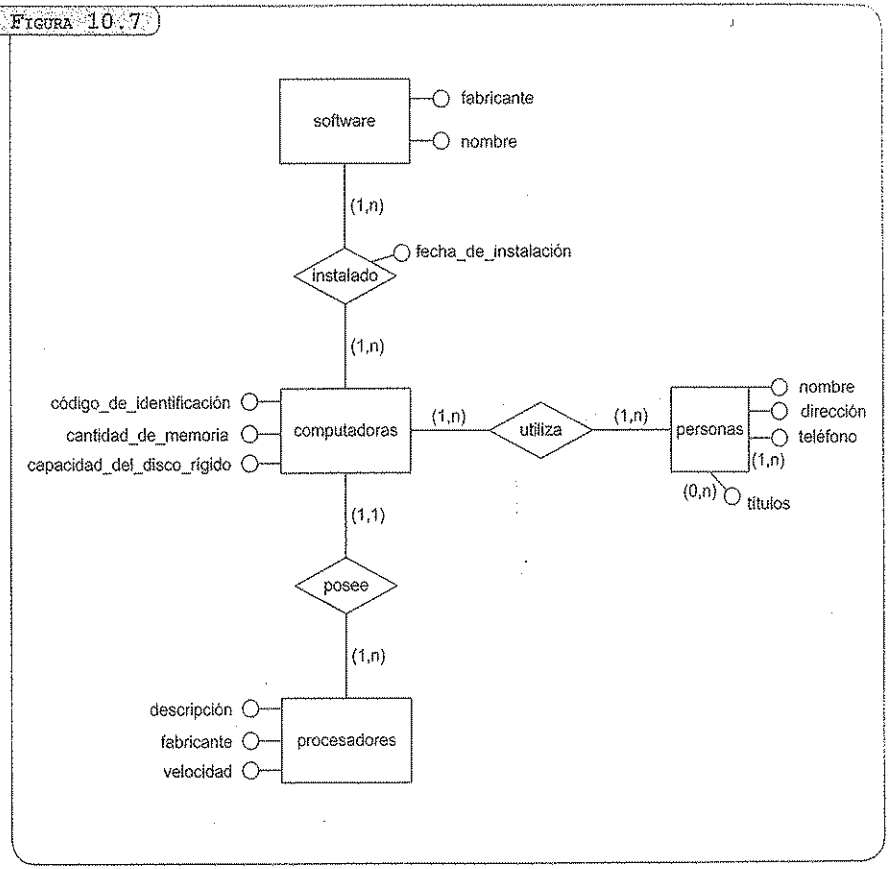
La primera de ellas presenta el caso de un colegio. Es necesario que el modelo represente a los alumnos que concurren al establecimiento. De cada alumno interesan su nombre, fecha de nacimiento, dirección y teléfonos. Los alumnos se inscriben en grados, de acuerdo con su nivel alcanzado. Cada grado tiene una denominación posible y, además, se conoce el cupo máximo para cada grado. A fines prácticos del colegio, asimismo se debe administrar el plantel docente. Los docentes son los responsables de dictar cursos en cada grado. Puede ocurrir que un grado disponga de más de un docente, pero necesariamente debe tener al menos uno asignado. Se desea conocer de cada docente su nombre, su DNI y los títulos que posea. Por último, cada docente es asignado solamente a un grado. La Figura 10.6 presenta el modelo de datos que resuelve el problema y que está representado con la herramienta de modelado CASER.

Figura 10.6



En el segundo ejemplo propuesto, el esquema representa el parque de computadoras de un instituto. De cada computadora se debe guardar un código de identificación, el tipo de procesador que tiene, la cantidad de memoria RAM y la capacidad del disco rígido. Además, cada computadora puede ser utilizada por varios miembros del instituto. Para ello, el problema también debe representar a estas personas (se conocen sus datos personales) y el/los títulos que pueden tener. Asimismo, cada computadora puede tener instalado diferente *software*; es importante tener registro de esta información, y desde qué fecha dicho *software* está instalado. La Figura 10.7 resume el modelo en cuestión.

Figura 10.7



Componentes adicionales del modelo conceptual

A fin de poder dotar al modelado conceptual con las características descriptas al principio de este capítulo, se han incorporado otros elementos que permiten aumentar la expresividad del modelo construido. Estos elementos son cuatro: jerarquías, subconjuntos, atributos compuestos e identificadores, y se presentan en este apartado.

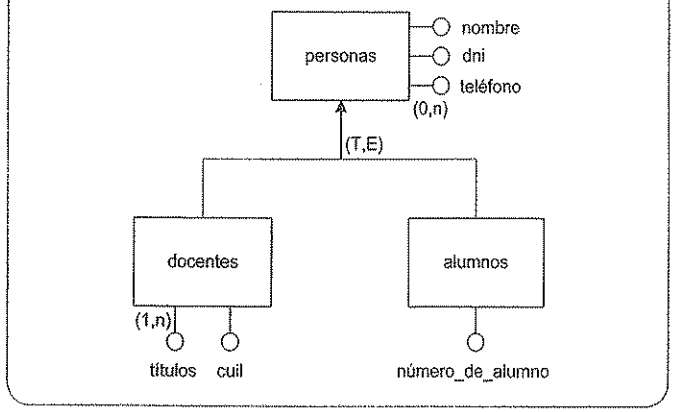
Jerarquías de generalización

Siguiendo el concepto de abstracción de generalización presentado en el Capítulo 9, es posible que un modelo conceptual ER incorpore el concepto de jerarquías de generalización de entidades o de relaciones.

La generalización permite extraer propiedades comunes de varias entidades o relaciones, y generar con ellas una superentidad que las aglutine. Así, las características compartidas son expresadas una única vez en el modelo, y los rasgos específicos de cada entidad quedan definidos en su subentidad.

La Figura 10.8 presenta un ejemplo que ilustra esta situación. Al representar docentes y alumnos de una facultad, es lógico suponer que ambas entidades comparten atributos tales como: nombre, dirección y teléfono. Estos atributos son definidos para la superentidad persona. Además, el alumno tiene rasgos propios, como por ejemplo el número de alumno, y el docente posee CUIL y títulos. Nótese, asimismo, que en la Figura 10.8 se expresa la cobertura (*T, E*) que posee la jerarquía definida, de acuerdo con las pautas ya establecidas.

Figura 10.8



La cobertura, al igual que la cardinalidad, depende directamente del problema. Nuevamente el analista no debe tomar decisiones unilaterales al respecto. Si no tiene suficiente información, debe recurrir a la fuente, el cliente o el usuario.

Se debe recordar la propiedad básica de la abstracción de generalización: el concepto de herencia. Las subentidades o especializaciones (alumnos y docentes en el ejemplo) heredan los atributos de la superentidad o generalización (persona).

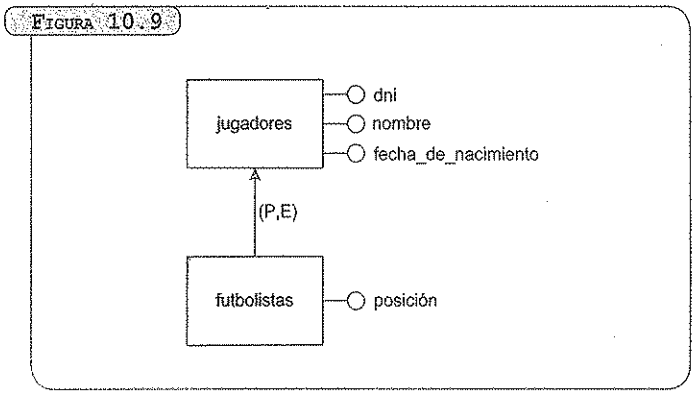
Subconjuntos

Los subconjuntos representan un caso especial de las jerarquías de generalización. Hay problemas donde se tiene una generalización de la que se desprende solamente una especialización. Este es el caso de los subconjuntos. La representación es similar al caso anterior.

Solamente hay que considerar que no es necesario indicar la cobertura para los subconjuntos. Esto se debe a que no puede tratarse de una cobertura total; si no, la especialización y la generalización serían lo mismo, representarían los mismos atributos. Además, no puede ser superpuesta, dado que no hay una segunda especialización con la cual superponerse. Por lo tanto, un subconjunto siempre indica cobertura parcial y exclusiva. Sin embargo, en la herramienta CASER se indica la cobertura aunque no sea necesario.

El ejemplo de la Figura 10.9 representa a los jugadores de todos los deportes de un club cualquiera. Aquí, cada jugador es reconocido por su nombre, DNI y fecha de nacimiento. En particular, para cada uno de los deportistas que juegan al fútbol, se debe definir la posición que le corresponde en la cancha.

Figura 10.9

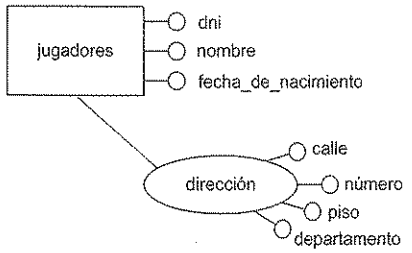


Atributos compuestos

Los atributos compuestos representan a un atributo generado a partir de la combinación de varios atributos simples. Un ejemplo para ilustrar esta situación puede ser la dirección de una persona. Se podría optar por modelar la dirección como un solo atributo simple donde se indican la calle, el número y, eventualmente, piso y departamento. Así, el dominio podría ser una cadena de 50 caracteres. Sin embargo, es interesante poder mantener la calle, el número, piso y departamento por separado. De esta forma, como queda indicado en la Figura 10.10, se establece la dirección como atributo compuesto por los cuatro ítems mencionados.

Un atributo compuesto podría ser polivalente o no obligatorio. Lo mismo podría ocurrir con los atributos simples que lo componen.

Figura 10.10



Identificadores

Un identificador es un atributo o un conjunto de atributos que permite reconocer o distinguir a una entidad de manera unívoca dentro del conjunto de entidades. El concepto de identificador está ligado a los conceptos de claves primarias y candidatas, vertidos en capítulos previos de este libro.

Para el modelado conceptual, se utiliza el concepto de identificador. Los identificadores pueden ser de dos tipos:

- **Simples o compuestos:** de acuerdo con la cantidad de atributos que lo conforman, el identificador es simple si está conformado por solo un atributo y es compuesto en el resto de los casos.

- **Internos o externos:** si todos los atributos que conforman un identificador pertenecen a la entidad que identifica, es interno; en su defecto, es externo.

La Figura 10.11 presenta diversas situaciones que ejemplifican atributos simples y compuestos, que en todos los casos son atributos internos. Se debe notar la representación gráfica que se utiliza para definirlos.

Figura 10.11

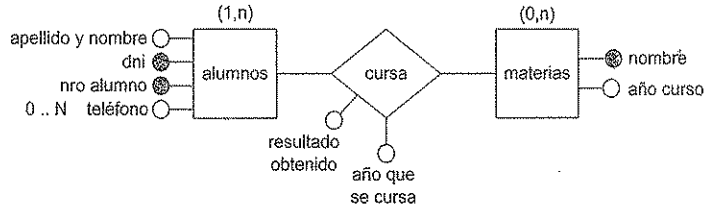


Figura 10.11 a) Identificadores simples

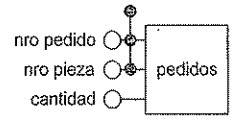


Figura 10.11 b) Identificador Compuesto

Suponga un esquema de una universidad donde los números de alumno se pueden repetir entre diferentes unidades académicas. Así, el número de alumno 1 está presente en Derecho, Informática, Ciencias Exactas, etcétera. Un ejemplo factible de identificador externo es combinar el número de alumno con la unidad académica de la que proviene. Se puede observar en la Figura 10.12 la notación para esta situación.

La entidad Alumnos tiene entonces un identificador simple interno (DNI), y otro compuesto y externo, el que se conforma con un identificador de Unidad Académica más el número de alumno.

Figura 10.12

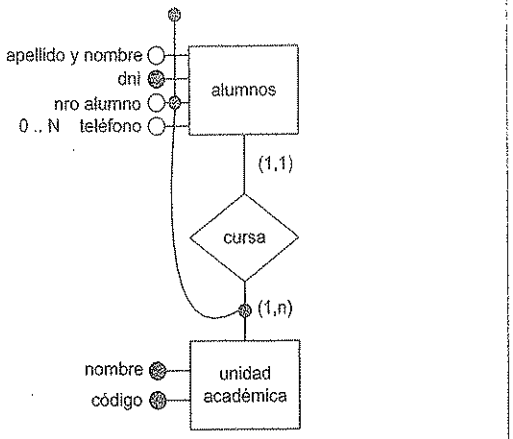
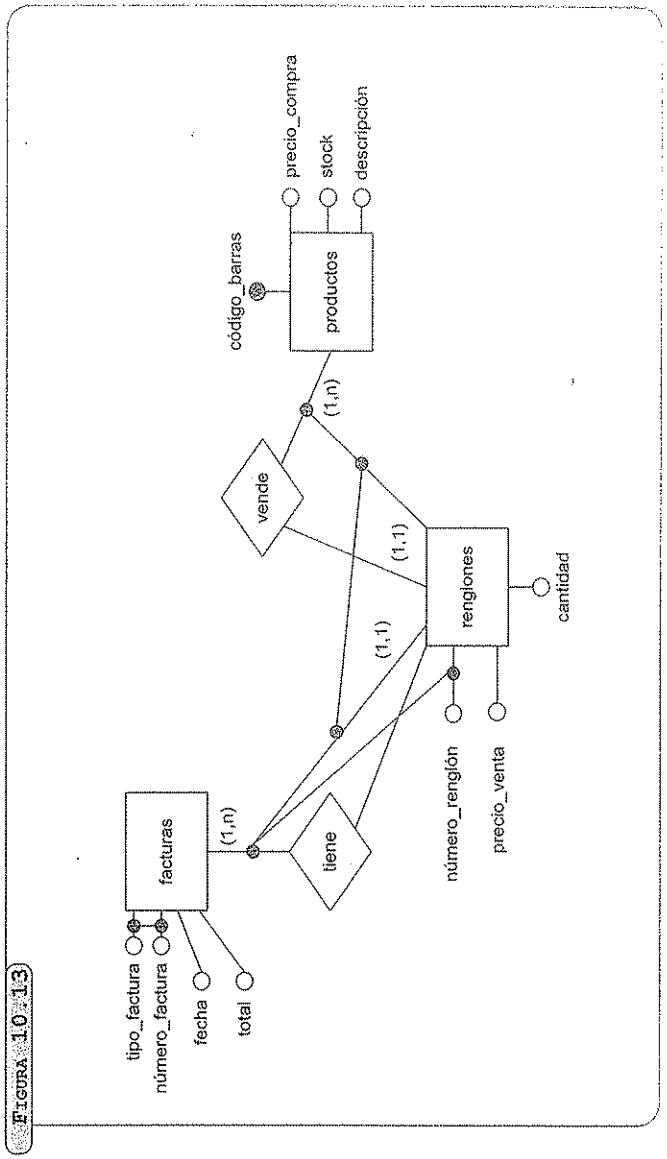


Figura 10.12
Identificador Externo

La Figura 10.13 presenta otro caso para definir identificadores. Se plantea la entidad facturas con los atributos tipo de factura, número, fecha y monto total. El detalle de cada factura está compuesto por renglones; en cada renglón se tiene el precio unitario de venta, el número de renglón y la cantidad vendida. Además, cada renglón representa la venta de un producto. De los productos se conocen el código de barras, la descripción, el *stock* y el precio de compra.

El identificador de facturas es compuesto, formado por tipo y número de factura; en productos, el identificador es el código de barras. La entidad renglones carece de identificador interno, dado que el número de renglón se repite en cada factura. Por lo tanto, combinando el identificador de facturas y el número de renglón se genera un identificador externo para renglones. Además, en una factura no deberían existir dos renglones con el mismo producto; entonces, los identificadores de facturas y productos combinados forman otro identificador externo para la entidad renglones.

Figura 10.13



Los identificadores en una jerarquía también son heredados. Esto es, si una generalización posee identificadores, estos son heredados por las entidades que son sus especializaciones. Además, las especializaciones pueden tener otros identificadores definidos.

Ejemplo integrador

A partir del uso de los elementos definidos hasta el momento para la construcción del modelo conceptual, es posible generar cualquier modelo que exprese la realidad de un problema. A continuación, se enuncia un problema y se plantea una solución posible.

Se debe modelizar el esquema de una facultad. En este esquema, los profesores (interesan los datos personales, más el número de CUIL y los títulos que posee cada docente) se relacionan con los departamentos (cada uno tiene un nombre único) donde cumplen sus actividades. Un docente solamente trabaja en un departamento, pero un departamento puede tener varios docentes. Además, cada docente debe indicar en qué ciudad ha nacido. El esquema debe representar asimismo a los estudiantes (descriptos por sus datos personales más el número de alumno, que es único en la facultad). Los estudiantes deben indicar el lugar de nacimiento y de residencia.

Además, se deben señalar las materias que se dictan en la facultad (de cada materia se sabe el nombre, que no puede repetirse, y el año dentro del plan de estudios). Los alumnos se inscriben en materias, las cuales son dictadas por profesores bajo las siguientes suposiciones: un docente puede dictar más de una materia y una materia puede estar dictada por más de un docente. Los alumnos pueden anotarse en más de una materia y se debe indicar el resultado obtenido. Se debe tener en cuenta que si un alumno no aprueba una materia, puede anotarse nuevamente, por lo que es importante conocer el año en que se anota. Los estudiantes pueden ser graduados. En este caso, tendrán definidos un trabajo de tesis y un asesor, que es un profesor de la casa. Pueden existir, además, profesores visitantes, para los cuales debe indicarse el país de procedencia.

La Figura 10.14 muestra el esquema realizado con CASER que resuelve dicho modelo.

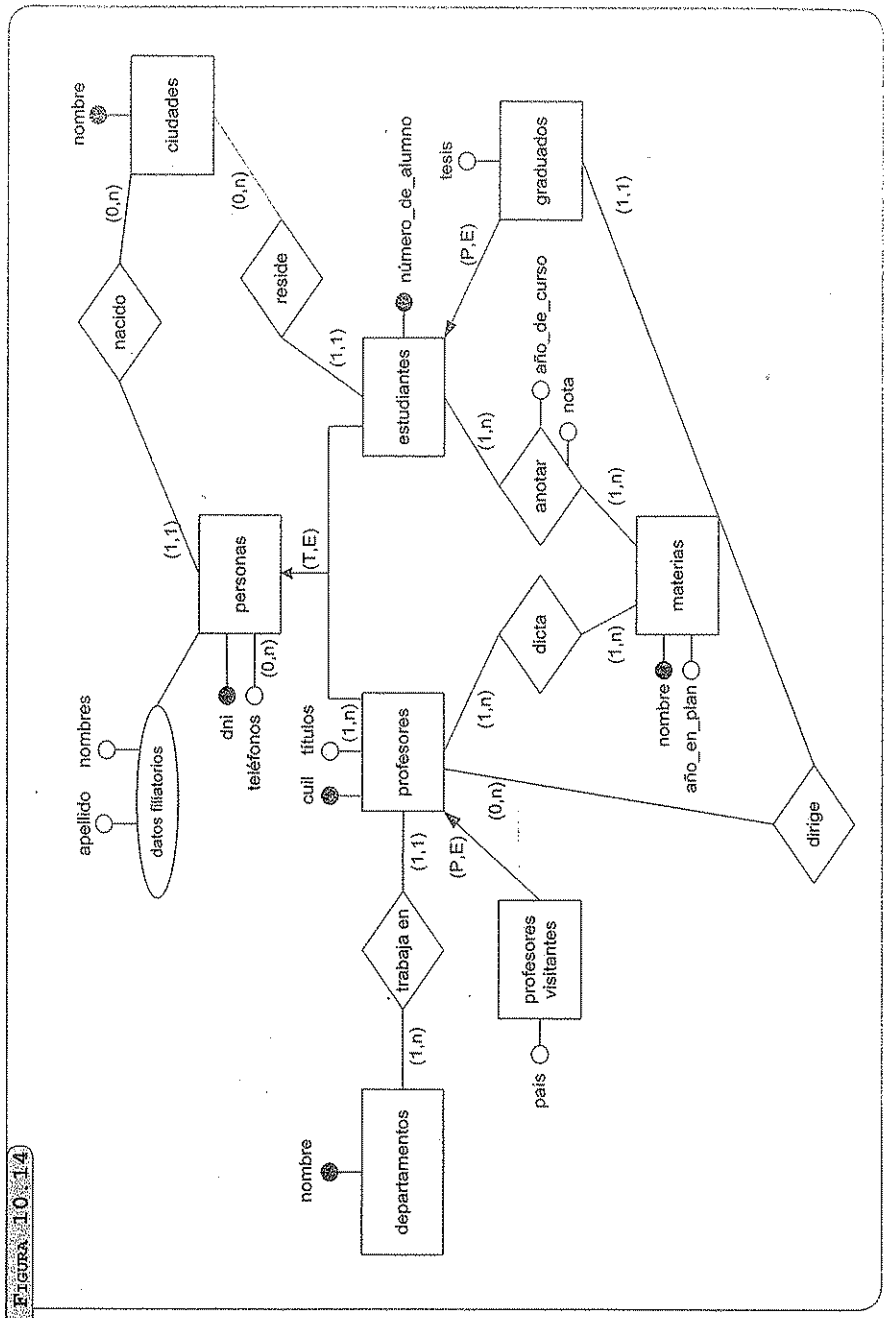


Figura 10.14

Consideraciones del modelo conceptual

Mecanismos de abstracción

Los tres mecanismos de abstracción presentados en el Capítulo 9 están presentes en el modelo conceptual ER.

Los tres conceptos básicos (entidades, relaciones y atributos) se basan en la abstracción de clasificación:

- **Entidades:** son una clase de objetos del mundo real con propiedades comunes.
- **Relaciones:** son un conjunto de objetos que relacionan dos o más entidades.
- **Atributos:** son una clase de valores, determinadas por el dominio de cada uno de ellos, que representan propiedades de entidades y/o relaciones.

La abstracción de agregación está presente en:

- **Entidades:** son agregaciones de atributos.
- **Relaciones:** son agregaciones entre entidades y, además, pueden contener agregaciones de atributos.
- **Atributos compuestos:** son agregaciones de atributos simples.

Por último, la abstracción de generalización solo se ha presentado en entidades, pero es posible, además, utilizar generalización de relaciones (aunque muy poco frecuente).

Ventajas y desventajas del modelo conceptual

El modelo ER ha sido el más utilizado para representar los modelos de datos para los SI. Presenta un conjunto de ventajas que lo hacen aplicable, pero, asimismo, se deben analizar algunas críticas respecto de su construcción.

Las cuatro propiedades indicadas al principio del capítulo –expresividad, formalidad, minimalidad y simplicidad– son ampliamente alcanzadas a partir de los elementos disponibles para construir el modelo.

Lograr que un modelo sea expresivo ayuda a comprender el problema, pero puede atentar contra la simplicidad.

La forma de expresar y determinar la cardinalidad y los identificadores no es sencilla. Se puede comparar el presente libro con otros relacionados con modelado de datos, y comprobar que la forma gráfica y semántica de expresar la cardinalidad varía.

Las relaciones *n*-arias también presentan inconvenientes. Como se discutió anteriormente, la cardinalidad es más compleja de expresar a fin de capturar todas las características del problema.

La minimalidad y la formalidad pueden ser alcanzadas. Obviamente, cada analista puede desarrollar un modelo de manera diferente a partir del mismo problema. Esto se debe a que la forma de representación que cada uno utilice capturará a su mejor entender la naturaleza del problema.

Revisión del modelo conceptual

Decisiones respecto de entidades, relaciones o atributos

Para construir un modelo conceptual, hay una serie de decisiones que el analista debe tomar. La decisiones dependen de tres factores fundamentales: el problema, lo que el cliente/usuario espera obtener desde la BD y, por último, la experiencia práctica del analista.

Para ejemplificar la situación, suponga que se deben modelizar los datos personales de los empleados de cierta compañía. Este problema ya fue analizado anteriormente, pero esta vez su análisis se detiene en dos conceptos: el tipo de documento de la persona y la localidad donde vive. Una opción podría ser que ambos conceptos sean atributos de persona, como muestra la Figura 10.15 a). Sin embargo, hay otra opción, presentada en la Figura 10.15 b); en este caso, tanto la localidad como el tipo de documento se definen en una nueva entidad que se relaciona con la entidad persona. ¿Qué ventajas presenta esta última solución? Una ventaja directa es que si se quiere saber cuántas personas son originarias de la ciudad de La Plata, bastará con contar las relaciones existentes entre empleados y la ciudad de La Plata. Si se hubiese escrito como un atributo, el valor obtenido dependería de la forma de escritura. Así, escribir todo en mayúscula o todo en minúscula podría afectar el resultado. Algo similar podría ocurrir con el tipo de documento.

Por lo tanto, la decisión respecto de cuál es la mejor forma para modelar depende de los tres factores antes descriptos.

Figura 10.15

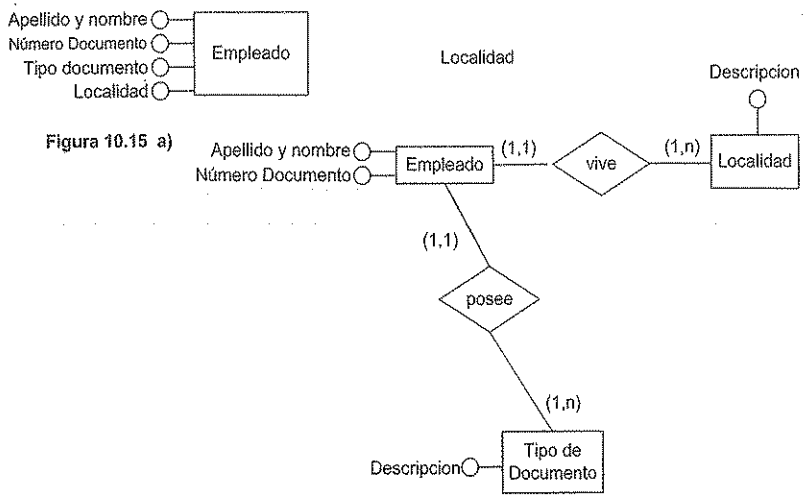


Figura 10.15 b)

La siguiente pregunta a realizar cuando se genera un modelo conceptual tiene que ver con la ventaja o no de definir generalizaciones. Aquí, la decisión debería tomarse considerando si la representación establece una jerarquía o una clasificación. Por ejemplo, si se quiere definir en el modelo de datos solamente el sexo de una persona, la solución se obtiene agregando un atributo a la entidad persona. Sin embargo, si hubiera características propias para los hombres y para las mujeres, sería conveniente generar una jerarquía. Así, la regla a aplicar debería controlar que no queden definidas entidades sin atributos o sin relaciones con otras entidades. Estas entidades (sin atributos y sin relaciones) se denominan entidades colgadas.

La última consideración está ligada, como se explicó anteriormente, a la utilización de atributos compuestos. ¿Conviene un atributo compuesto o directamente se colocan sobre la entidad los atributos simples? Los atributos compuestos se deberían utilizar y luego, cuando se evolucione en el modelo lógico o físico, decidir cómo actuar.

Transformaciones para mejorar el modelo conceptual

Una vez finalizada la construcción del modelo conceptual, este debe ser puesto en consideración y análisis por parte del cliente/usuario. Si bien, este no es un proceso sencillo, con la ayuda del analista es posible que alguien no ligado a la actividad informática pueda analizar el modelo resultante para tratar de determinar faltantes o errores.

Es necesario, entonces, poder lograr un modelo que represente la realidad del problema lo más estrictamente posible, disminuyendo ambigüedades y tomas de decisiones de bajo nivel, y aumentando la expresividad y legibilidad del modelo construido.

Existen una serie de conceptos a revisar:

- **Autoexplicativo:** un modelo se expresa a sí mismo si puede representarse utilizando los elementos definidos, sin necesidad de utilizar aclaraciones en lenguaje natural para expresar características.
- **Completitud:** un modelo está completo cuando todas las características del problema están contempladas en él. La forma de validar la completitud es revisar la especificación de requerimientos asociada al problema.
- **Corrección:** un modelo es correcto si cada elemento en su construcción fue utilizado con propiedad. Algunos aspectos de fácil resolución en cuanto a correctitud son observar que todas las cardinalidades y coberturas se hayan expresado, haber tenido en cuenta el concepto de herencia en jerarquías, haber expresado todos los identificadores, etcétera.
- **Expresividad:** el modelo conceptual resulta expresivo si a partir de su observación es posible darse cuenta de todos los detalles que lo involucran. Existen varias formas de mejorar la expresividad. Si el lector observa la Figura 10.14, se han utilizado jerarquías para personas, profesores, alumnos y graduados, las que permiten realizar relaciones con otras entidades. Si no se hubiesen utilizado esas jerarquías, la expresividad del modelo se vería seriamente afectada.
- **Extensible:** el modelo conceptual resulta extensible si es fácilmente modificable para incorporar nuevos conceptos en él, resultantes de cambios en los requerimientos del problema.
- **Legibilidad:** un esquema es legible si la representación gráfica es adecuada. Este fue uno de los motivos que llevó a la generación de CASER: disponer de una herramienta para la asistencia en la generación del modelo conceptual, basado en los elementos propios del modelo.

- **Minimalidad:** un esquema es mínimo cuando cada concepto se representa una sola vez en el modelo. Aquí hay dos factores posibles que pueden afectar la minimalidad. Estos factores son: atributos derivados y ciclos de relaciones. Más adelante en este apartado, se ejemplifican estos casos y se analiza la importancia de reconocerlos.

El diseñador de la BD, una vez terminada la construcción del modelo conceptual inicial, debe repasar los puntos anteriores y determinar si fueron cubiertos por el modelo de datos. Las transformaciones a realizar se aplican sobre el esquema obtenido, y generan un nuevo esquema con las características anteriormente analizadas sin afectar el contenido original. Ambos esquemas deben ser en todo momento equivalentes con respecto al contenido de la información.

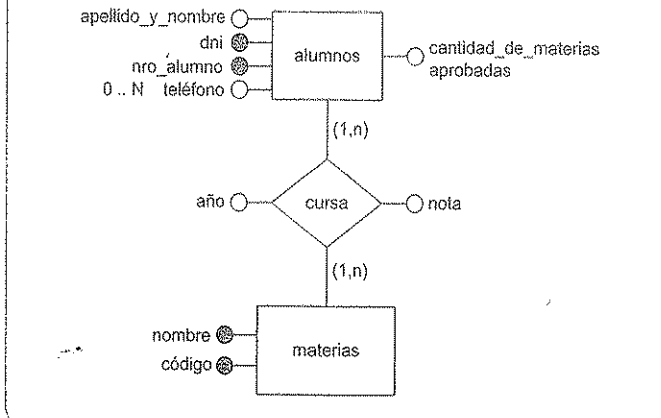
Transformaciones para minimalidad

Un esquema es mínimo cuando una idea se expresa en él una sola vez. La redundancia de información puede ser un efecto positivo si es conocida y controlada, o un efecto negativo si aparece en el esquema por deficiencia en la modelización. En estos últimos casos, la redundancia puede llevar a pérdida de consistencia en la información.

Hay dos causas que llevan a un modelo a perder la minimalidad. En este apartado, se analizan las situaciones. Se debe notar que el resultado esperado en este punto es detectar las condiciones que afectan la minimalidad, pero no necesariamente solucionarlas. Estas decisiones se pueden tomar cuando se genera el modelo lógico del problema.

Los atributos derivados representan la primera causa que afecta la minimalidad. Un atributo derivado es un atributo que aparece en el modelo, pero cuya información, si no existiera almacenada en él, podría igualmente ser obtenida. La Figura 10.16 presenta un atributo derivado, la cantidad de materias que el alumno tiene aprobadas. Si este atributo no existiera, bastaría con contar las relaciones existentes en cursa con nota superior a 4 para obtener dicho resultado. Si el atributo derivado existe, el esquema no es mínimo; no obstante, las decisiones finales sobre los atributos derivados, como se indicó anteriormente, se tomarán sobre el esquema lógico en el Capítulo 11.

Figura 10.16



Los ciclos de relaciones presentan la segunda causa que afecta la minimalidad. Se dice que existe un ciclo cuando una entidad *A* está relacionada con una entidad *B*, la cual está relacionada con una entidad *C*, la que a su vez se relaciona con la entidad *A*.

No todos los ciclos afectan la minimalidad. Un ciclo afecta la minimalidad cuando una de las relaciones existentes puede ser quitada del esquema, y aun en esas condiciones, el modelo sigue representando la misma información.

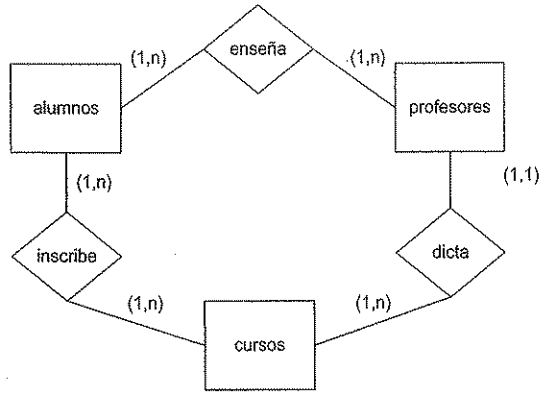
La Figura 10.17 presenta un ejemplo que define las entidades profesores, alumnos y cursos. Los profesores dictan cursos, los cursos son tomados por alumnos y se sabe, además, el dato de los alumnos que tiene cada docente. En este caso, un profesor puede dictar solamente un curso, pero un curso puede ser dictado por varios profesores. Un curso puede ser tomado por varios alumnos, los que a su vez pueden tomar varios cursos. Por último, un alumno puede tener varios docentes y estos, varios alumnos.

Para analizar si se tienen ciclos no deseados, deberían quitarse del modelo una a una las relaciones para determinar si es posible administrar la información deseada. Suponga que se quita la relación dicta; a partir de la relación inscribe es posible determinar los alumnos de cada curso, y de la relación enseña, los profesores que dictan clases al alumno, pero no es posible afirmar si el profesor dicta o no un curso específico. Ejemplo: el curso es Análisis Matemático I; se

obtiene el listado de alumnos inscriptos: suponga que uno de ellos es José García. De este alumno se obtiene el listado de profesores que dictan varias materias, suponga Gomez, Perez y Díaz; ¿cuál de ellos dicta Análisis Matemático I?, es imposible de determinar; por lo tanto, la relación cursa debe ser parte del modelo.

Si se intenta quitar la relación enseña, se debería poder determinar la lista de alumnos de un profesor. Dado un profesor, Perez, se sabe qué materia dicta, la cual puede ser dictada por varios profesores. Por lo tanto, la lista de alumnos de la materia no determina exactamente con qué profesor cursan. Por consiguiente, la relación enseña no puede ser quitada.

Figura 10.17



Queda por analizar la relación inscribe; dado un alumno, se puede determinar qué profesores le imparten clase. Como cada profesor puede dictar solamente una materia, es posible saber en qué materia se inscribió el alumno. A partir de la materia se puede conocer la lista de profesores que la dictan, y con esos profesores, la lista de alumnos a los que se imparte clase. Por lo tanto, nuevamente es posible conocer la nómina de alumnos inscriptos. Esto demuestra que si se quita del modelo la relación inscribe, es posible conocer la misma información respecto del problema. Esto significa que el esquema no es mínimo. Nuevamente, las redundancias que se producen con el ciclo se pueden resolver sobre el modelo lógico.

Queda como tarea tomar el ejemplo de la Figura 10.17, y analizar qué sucedería si un docente pudiera dictar varios cursos.

Transformaciones para expresividad y autoexplicación

Las transformaciones para lograr expresividad y autoexplicación tienen varias causas; se analizan algunas de ellas.

Un modelo es más expresivo si las jerarquías que se representan tienen sentido. Una jerarquía se establece para obtener atributos comunes en una generalización, dejando los atributos específicos para la especialización. El problema surge cuando las especializaciones carecen de dichos atributos. Este ejemplo es bastante común y se observa más cuando el problema determina una clasificación de elementos.

Suponga que los productos que vende cierta compañía se dividen en productos para el hogar, para el auto y para la oficina. Cuando se plantea el esquema conceptual, se establece una generalización de productos (con todos los atributos pertinentes) y especializaciones para productos para el hogar, el auto y la oficina, respectivamente. Pero estas especializaciones no tienen ningún atributo que las conforme. Claramente, se está definiendo en el esquema una clasificación como si fuese una jerarquía, y esto es una solución no deseada.

La Figura 10.18 presenta el mismo problema con dos alternativas; en la segunda se puede observar claramente el significado del atributo tipo que aparece en la relación dirige.

Figura 10.18

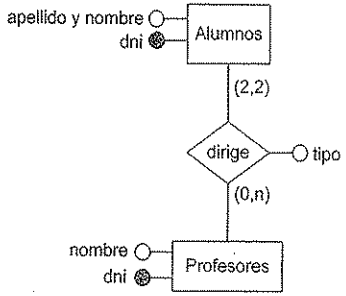


Figura 10.18 a)

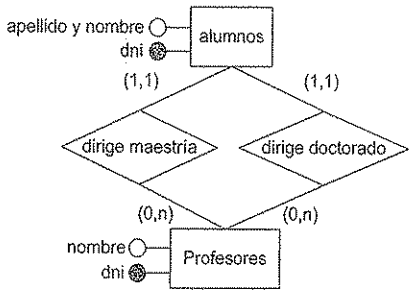


Figura 10.18 b)

Otras causas que mejoran la expresividad y la autoexplicación están relacionadas con utilizar jerarquías o subconjuntos cuando aún no fueron definidas. La Figura 10.19 a) muestra un esquema donde algunos alumnos tienen director de tesis. En la Figura 10.19 b), se define un subconjunto que establece una clase particular de alumnos, alumnos de posgrado, y estos son los que tienen director de tesis.

Figura 10.19

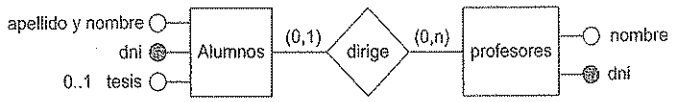


Figura 10.19 a)

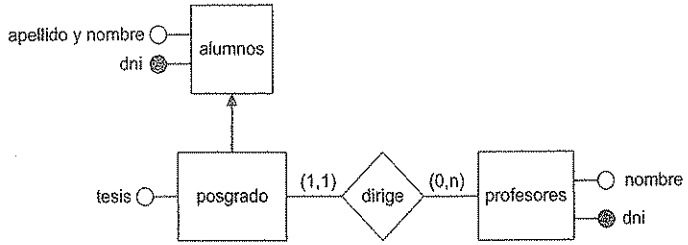


Figura 10.19 b)

Cuestionario del capítulo

1. ¿Cuál es la finalidad de generar un modelo conceptual de datos?
2. ¿Por qué son necesarias, para un modelo conceptual, las cuatro características definidas?
3. ¿Cuáles son los constructores básicos definidos por Chen para el modelo conceptual?
4. ¿Por qué se agregan nuevos constructores?
5. ¿Qué son las transformaciones de esquemas y qué finalidad persiguen?

Ejercitación

1. Resuelva los siguientes problemas simples, indicando las entidades, relaciones y atributos involucrados. Además, indique la cardinalidad de los atributos y de las relaciones.
   a. Se tiene un problema de un banco donde se maneja información de los clientes y las cuentas con las que operan. Esas cuentas pueden ser corrientes, de ahorro o accionarias. De los clientes se conocen su DNI, nombre, dirección y teléfonos; asimismo, el banco les define un código único. Los clientes pueden tener varias cuentas. Las cuentas corrientes tienen un código único, un saldo y un máximo valor en descubierto. Las cajas de ahorro poseen un código único, un saldo, una fecha de vencimiento de la tarjeta de débito asociada, y el monto máximo para operar en cada transacción. Las cuentas accionarias poseen un código y una fecha de apertura, y están relacionadas con todas las acciones que maneja el cliente.
   b. Historias clínicas. Cada paciente tiene una historia clínica y esta tiene un número identificatorio. De cada paciente se conocen sus datos personales. Cada paciente se atiende con médicos, cuyos datos personales también se conocen. Cada médico tiene una especialidad. Cada paciente puede atenderse con varios médicos.

   A partir del modelo anterior, se debería poder responder:

   i. ¿Cuántos médicos hay por cada especialidad?

   ii. ¿Con cuántos médicos se atiende un paciente?

c. Una biblioteca tiene libros. Los libros tienen un ISBN que los identifica, autores, editorial y año de edición. Por cada libro puede haber varias copias. Cada copia tiene un código único dentro de la biblioteca. Los autores tienen nombre, DNI, código de identificación y nacionalidad.

A partir del modelo anterior, se debería poder responder:

i. ¿Cuántos autores de nacionalidad argentina aparecen en libros de la biblioteca?

ii. ¿Cuántos libros de cada editorial hay?

iii. ¿Cuántas copias de cada libro hay?

d. Se desea administrar información de una facultad. Los datos a gestionar son: alumnos, carreras que se cursan, materias que se dictan y plantel docente. El plantel docente está integrado por profesores y auxiliares docentes. De los alumnos se conocen su DNI, código de alumno, nombre, dirección, e-mail y teléfonos. De cada docente interesan su DNI, nombre, dirección, e-mail, teléfonos, CUIL y títulos obtenidos. Cada docente dicta una asignatura en un año particular. Los alumnos pueden anotarse a cursar (o recursar) varias asignaturas.

A partir del modelo anterior, se debería poder responder:

i. ¿Cuántos docentes tienen título de Doctor?

ii. ¿Cuántos alumnos recursan materias?

iii. ¿Qué nacionalidad tiene cada docente y/o alumno?

e. El esquema representa el menú de platos de una cadena de restaurantes. Para cada restaurante se conocen el nombre, la ubicación, la capacidad y los datos de su responsable. La cadena de restaurantes tiene platos a disposición. Cada plato tiene un nombre, un costo y una serie de ingredientes. Dentro de la cadena, cada restaurante puede o no tener el plato, y puede, además, cobrarlo diferente.

2. Indique en cada ejercicio anterior los posibles identificadores a partir de la forma en que lo haya resuelto y de las siguientes consideraciones: las personas pueden reconocerse por DNI-CUIT-CUIL; los alumnos, por su número.

3. Indique, ante cada problema, si es pertinente o no el uso de una jerarquía. En cada caso, indique la cobertura:

a. El esquema representa a clientes y proveedores de la empresa. En el caso de los clientes, se debe indicar si la facturación es de tipo A o B. En el caso de los proveedores, es necesario indicar el tipo de servicio que prestan, y si la empresa tiene o no una cuenta corriente.

b. Un supermercado administra la información de sus productos. De cada producto se conocen el nombre, la presentación del envase, el precio de costo y el precio de venta. Asimismo, es importante conocer quién es el proveedor. Los productos son perecederos o no; además, pueden ser de ferretería, almacén, carnicería, etcétera.

c. Cierta facultad ordena su plan curricular de la siguiente manera: cada materia tiene un nombre, un año de cursada, uno o más profesores que la dictan y, además, pertenece a un área. Por el momento, las áreas disponibles en la facultad son: Programación Básica; Ingeniería de Software y Bases de Datos; Redes y Sistemas Operativos; y Programación Avanzada.

CAPÍTULO 11

Modelado entidad relación lógico

Objetivo

El propósito de la generación de un modelo ER lógico es convertir el esquema conceptual en un modelo más cercano a la representación entendible por el SGBD. El principal objetivo del diseño conceptual consiste en captar y representar, de la forma más clara posible, las necesidades del usuario definidas en el documento de especificación de requerimientos de un problema. Una vez cumplido este paso, el diseño lógico busca representar un esquema equivalente, que sea más eficiente para su utilización.

Al iniciar la definición del modelo lógico, se necesita definir el tipo de SGBD que se utilizará posteriormente para su implantación física. Esto es, la secuencia de pasos de conversión disponibles tiene estrecha relación con el tipo de SGBD. Recuerde que se mencionaron cuatro tipos diferentes: relacional, OO, jerárquico y de red.

En esta obra, se desarrollará la conversión del modelo conceptual en modelo lógico, bajo la suposición de que el SGBD será relacional. Esta elección tiene varios motivos que la fundamentan. El principal motivo es que el modelo relacional es el más ampliamente difundido y utilizado. Los modelos jerárquicos y de red se encuentran en desuso desde hace ya mucho tiempo. El modelo OO aún no se ha impuesto en el mercado y, en general, es menester tratar las BDOO en bibliografía más específica.

Los objetivos concretos del presente capítulo están vinculados con definir una metodología de conversión del modelo conceptual en modelo lógico, estudiar alternativas de decisión de acuerdo con la utilización prevista de la BD y analizar los criterios de carga de datos sobre los elementos del modelo.

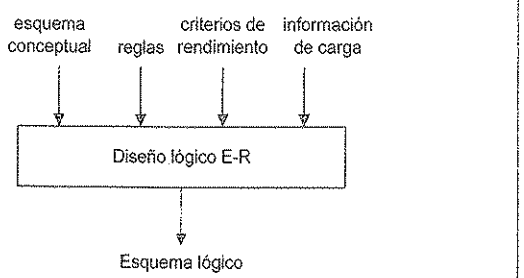
Características del diseño lógico

El diseño lógico del modelo de datos de un problema produce como resultado el esquema lógico de dicho problema, en función de cuatro entradas:

1. **Esquema conceptual:** es el resultado tangible de la etapa inmediata anterior. El esquema conceptual representa la solución, a juicio del analista, respecto del problema original. El esquema lógico a obtener debe representar la misma información disponible en el esquema conceptual.
2. **Descripción del modelo lógico a obtener:** aquí se deben definir las reglas que se aplicarán en el proceso de conversión. Esas reglas están ligadas al tipo de SGBD seleccionado. Para esta obra se repasarán, en el apartado siguiente, aquellas que permiten obtener un modelo lógico compatible con el modelo relacional.
3. **Criterios de rendimiento de la BD:** durante la fase de diseño conceptual, se consideraron los requerimientos del usuario. No obstante, hay otro tipo de necesidades que no se pueden definir sobre el modelo conceptual. Estas necesidades tienen que ver con requerimientos, en general, no funcionales del problema, como por ejemplo *performance* de la BD. Así, una regla puede dar alternativas de solución y el analista deberá optar por aquella que permita alcanzar los estándares de rendimiento definidos para el problema.
4. **Información de carga de la BD:** este concepto aparece, en cierta forma, ligado al concepto anterior. Cuando se genera el esquema lógico, el analista debe observar cada entidad e interrelación definida, y ver la probable evolución de la información contenida en esas estructuras. De este modo, la decisión final sobre el esquema de una relación o entidad dependerá del número probable de elementos que la compondrán, con el propósito de mantener la *performance* bajo control. Como se discutió en la Sección II del libro, cuanto más grande es un archivo de datos, más lento es el proceso de búsqueda, inserción o gestión de la información.

La Figura 11.1 describe de manera gráfica las entradas descriptas.

Figura 11.1



En el presente libro, como se indicó entre los objetivos del capítulo, se definirán las reglas asociadas a la conversión del esquema lógico orientado al modelo relacional. Además, se pondrá énfasis en ejemplificar diferentes situaciones para dichas reglas, considerando también cuestiones de rendimiento o información de carga de datos en la BD. Esto se debe, básicamente, a que la naturaleza del rendimiento y la información de carga están directamente vinculadas con el problema que se encuentra bajo modelado, y con los requerimientos no funcionales de dicho problema. La definición en forma exhaustiva de estos requerimientos supera el objetivo planteado en esta obra.

Se debe tener en cuenta que, en la resolución de un modelo de datos para un problema concreto, el proceso de conversión hacia el esquema lógico se llevará a cabo una vez afianzados los requerimientos del problema original; es decir, una vez que la especificación de requerimientos esté consensuada con el cliente. Esto puede significar, según el caso, un periodo de tiempo considerable, en días o en meses.

Decisiones sobre el diseño lógico

Las decisiones sobre el diseño lógico están vinculadas, básicamente, con cuestiones generales de rendimiento y con un conjunto de reglas que actúan sobre características del esquema conceptual que no están presentes en los SGBD relacionales. Por ejemplo, el concepto de herencia no está presente en el modelo relacional; esto implica que

las jerarquías deberán ser resueltas para adaptarlas a este contexto. Además, el modelo relacional carece de un dominio que permita definir varios atributos; esto quiere decir que no es posible representar atributos compuestos.

En este apartado, se discutirán aquellas cuestiones sobre las cuales se deben tomar decisiones de diseño, que permitirán convertir el esquema conceptual en un esquema lógico.

Atributos derivados

Un atributo es derivado si contiene información que puede obtenerse de otra forma desde el modelo. Es importante detectar dichos atributos, y en el diseño lógico se debe tomar la decisión respecto de dejarlos o no.

La ventaja de un atributo derivado es, básicamente, la disponibilidad de la información. En el Capítulo 10, la Figura 10.16 incluye el atributo derivado *cantidad de materias aprobadas por cada alumno*. Si esa información es muy requerida, estará disponible rápidamente, sin la necesidad de contabilizar cuántas veces aparece un alumno en la relación cursa, con nota igual o superior a 4 (suponiendo que dicha nota representa el valor mínimo para aprobar).

La desventaja de un atributo derivado radica en que necesita ser recalculado cada vez que se modifica la información que contiene. Así, cada vez que un alumno aprueba una materia, es necesario modificar el atributo cantidad de materias aprobadas de la Figura 10.16.

Por lo tanto, la pauta sobre los atributos derivados es dejar en el modelo a todos aquellos que son muy utilizados, y quitar los que necesitan ser recalculados con frecuencia. Se puede observar que, ante un atributo derivado muy utilizado y con mucho recálculo, no es sencillo establecer una pauta. En dichas situaciones, la decisión queda a criterio del analista.

Ciclos de relaciones

En el Capítulo 10, se remarcó que se deben identificar las relaciones que generan repetición innecesaria de información. La Figura 10.17 presenta un ciclo generado entre las entidades alumnos, profesores y cursos, donde se repite información.

Nuevamente, la decisión recae sobre el analista. Para que el modelo no genere información redundante, se debería, en el ejemplo, quitar

la relación inscribe. De esa forma, el modelo quedaría mínimo. Esto significa que, para obtener una lista con los alumnos inscriptos en una materia, se deberá utilizar a la entidad profesores, generando más tiempo de procesamiento.

Por lo tanto, la decisión del analista pasa por tener el modelo mínimo, o por que posteriormente el modelo implique menos tiempo de procesamiento.

Atributos polivalentes

Los SGBD, en general, permiten que sus atributos contengan múltiples valores. Así, es posible que un atributo tenga una dimensión, generando una estructura equivalente a los vectores en memoria. Este tipo de estructura es estática, es decir, la cantidad de lugares previstos para almacenar información está predeterminada.

Un atributo se denomina polivalente cuando puede tomar varios valores diferentes. En el Capítulo 10, se plantearon varios ejemplos: teléfonos o títulos fueron algunos de ellos. En esos casos, se establecía que una persona podría tener varios teléfonos o varios títulos, sin un límite *a priori*.

Ningún SGBD relacional permite que un atributo contenga valores múltiples determinados dinámicamente. Es decir, se puede tener un atributo con múltiples valores, pero la cantidad máxima debe ser previamente determinada. Esto lleva a dos situaciones extremas: para tener suficiente espacio para almacenar los títulos, por ejemplo, se determina que el atributo podrá contener hasta 10 valores diferentes (un número que puede resultar razonable). En general, una persona puede tener dos o tres títulos; con el consiguiente desperdicio de espacio. Pero peor aun, puede ocurrir que una persona tenga 11 títulos y no sea posible registrar a uno de ellos. Ambas situaciones son anómalas, y tienen que ver con definir una estructura estática para almacenar información que puede variar.

En este caso, la solución debe implementarse con otro criterio. El criterio está establecido y en la bibliografía se lo denomina, básicamente, primera forma normal.

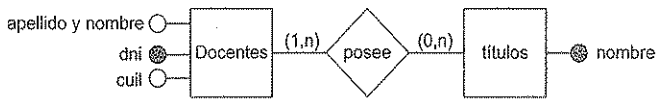
Un modelo está en **Primera Forma Normal (1FN)** si todos los atributos de entidades o relaciones son atributos simples.

Como corolario de la definición anterior, se puede decir que el modelo estará en 1FN si no tiene ningún atributo polivalente.

¿Cuál es la solución propuesta para satisfacer la definición anterior?

Si en el Capítulo 10 se observa la Figura 10.8, la entidad docentes tiene definido el atributo títulos como polivalente obligatorio. A fin de cumplir con la 1FN, la solución en este caso consiste en quitar el atributo polivalente de la entidad docentes, generando una nueva entidad denominada títulos, y estableciendo la relación posee entre docentes y títulos. Esta relación será una relación muchos a muchos, indicando que un docente puede tener varios títulos y que un título puede corresponder a varios docentes, como lo muestra la Figura 11.2.

Figura 11.2



Con esta solución, es posible indicar, sobre la relación, que un docente tiene *n* títulos sin tener que limitar el valor de *n*. El modelo resultante se encuentra en 1FN. Sin embargo, no es la única solución posible.

Suponga que en una entidad se define un atributo cantidad_cobrada_por_mes, y dicho atributo es polivalente con 12 valores posibles, uno para cada mes del año. Nuevamente, la solución puede generarse similar al ejemplo anterior. Se genera una nueva entidad y se establece una relación con la entidad existente. Pero en este caso, la cantidad exacta de valores posible es conocida. El diseñador de la BD podría optar por definir el atributo con una estructura de 12 elementos sin generar una nueva entidad.

La situación anterior es válida, pero se debe tener en cuenta que la forma de resolución planteada por 1FN es mucho más general, y en ningún caso presentará inconvenientes.

Atributos compuestos

Los atributos compuestos son atributos que están conformados por varios atributos simples. En un lenguaje de tercera generación, como por ejemplo Pascal, un atributo compuesto se resuelve generando un registro donde uno de sus campos es, en sí mismo, otro registro.

Los SGBD, en general, no soportan esta variante. Por lo tanto, los atributos compuestos resultan de mucho interés para la generación del modelo conceptual, pues permiten evitar tomar decisiones rápidamente con poca información sobre el dominio del problema. Por ejemplo, el domicilio de una persona se puede definir como la conjunción de calle, número, piso y departamento, o el nombre de una persona, como su apellido y nombres.

Luego, sobre el modelo lógico es necesario decidir qué hacer con un atributo compuesto.

Existen tres respuestas posibles, cada una de las cuales presenta ventajas y desventajas. El diseñador de la BD es el responsable de optar por una de las tres alternativas.

1. La primera solución propuesta consiste en generar un único atributo que se convierta en la concatenación de todos los atributos simples que contiene el atributo compuesto. De esta forma, domicilio se define con un dominio string[50], donde el usuario debería ingresar todos los datos de un domicilio: calle + número + piso + departamento. Esta solución es simple y sencilla de implantar, pero al unir todos los atributos simples que forman el compuesto, se pierde la identidad de cada atributo simple. Conocer cuánta gente vive en un primer piso no se puede resolver directamente.
2. La segunda solución plantea definir todos los atributos simples sin un atributo compuesto que los resuma. Con esta solución, el atributo domicilio desaparece y sobre la entidad se definen los atributos calle, número, piso y departamento. La cantidad de atributos aumenta, pero esta solución permite al usuario definir cada uno de los datos en forma independiente. Esta solución es, en general, la más indicada.
3. La tercera solución presenta una alternativa más radical. Consiste en generar una nueva entidad, la que representa el atributo compuesto, conformada por cada uno de los atributos simples que contiene. Esta nueva entidad debe estar relacionada con la entidad a la cual pertenecía el atributo compuesto. Se debe notar que esta solución capta mejor la esencia del atributo compuesto, pero es una opción más compleja.

La elección dependerá del diseñador de la BD. En general, la segunda alternativa es la más utilizada, pues se adapta mejor a la mayoría de las situaciones de la vida real.

Jerarquías

Las decisiones respecto de las jerarquías constituyen el punto más importante de convertir un modelo conceptual en lógico.

El modelo relacional no soporta el concepto de herencia; por consiguiente, las jerarquías no pueden ser representadas.

El proceso de diseño lógico necesita, entonces, encontrar un mecanismo que represente las jerarquías, captando el dominio del conocimiento de estas, bajo un esquema que no administre herencia.

Básicamente, hay tres opciones para tratar una jerarquía. Estas opciones son las siguientes:

1. Eliminar las especializaciones (subentidades o entidades hijas), dejando solo la generalización (entidad padre), la cual incorpora todos los atributos de sus hijos. Cada uno de estos atributos deberá ser opcional (no obligatorio).
2. Eliminar la entidad generalización (padre), dejando solo las especializaciones. Con esta solución, los atributos del padre deberán incluirse en cada uno de los hijos.
3. Dejar todas las entidades de la jerarquía, convirtiéndola en relaciones uno a uno entre el padre y cada uno de los hijos. Esta solución permite que las entidades que conforman la jerarquía mantengan sus atributos originales, generando la relación explícita ES_UN entre padre e hijos.

Las tres soluciones no son aplicables en todos los casos. A continuación, se describen diferentes situaciones y se analizan las soluciones posibles, ventajas y desventajas. La cobertura de la jerarquía es la que determina la solución viable en cada caso.

Si la cobertura de la jerarquía fuese parcial, la segunda solución planteada anteriormente no resultaría aplicable. Una cobertura parcial significa que algunos elementos contenidos en la entidad padre no están cubiertos por las especializaciones. Esto implica que, si se quita la entidad padre, dichos elementos no tendrán más cabida en el modelo. Esta conversión genera un modelo lógico que no es equivalente al modelo conceptual, ya que se pierde información. Por lo tanto, la segunda alternativa no es aplicable en el caso de tratarse de cobertura parcial.

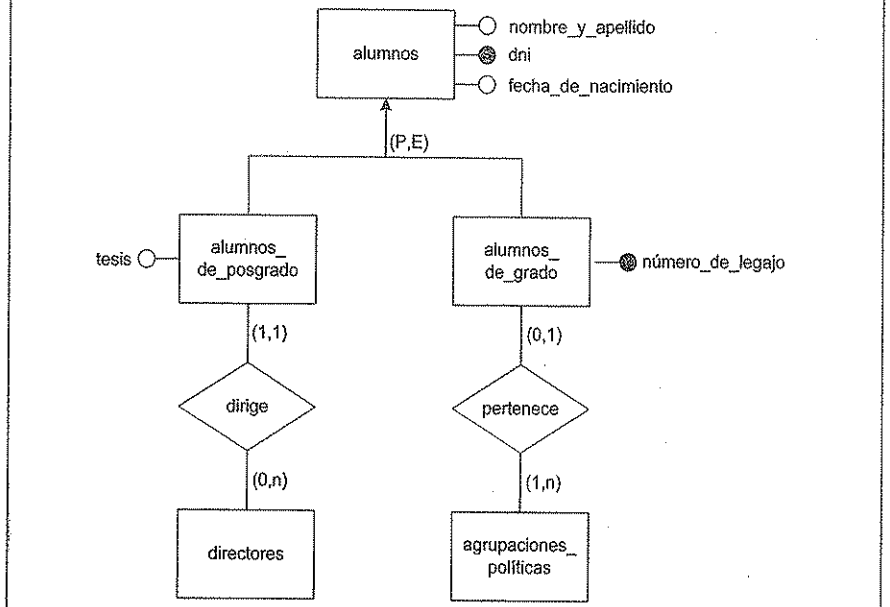
Si se analiza la cobertura superpuesta, la segunda solución, nuevamente, resulta poco práctica. Algunos elementos del padre se repiten en varios hijos; esto significa que se deberá repetir información en las subentidades generadas. Si bien esto no representa un problema en sí mismo, la repetición innecesaria de información puede ocasionar inconvenientes cuando se utilice la BD generada.

A continuación, se presentan dos ejemplos. En el primero de ellos se podrá observar cada una de las tres soluciones propuestas. En el segundo ejemplo, al tratarse de cobertura parcial, la segunda solución queda descartada.

Primer ejemplo de resolución de jerarquías

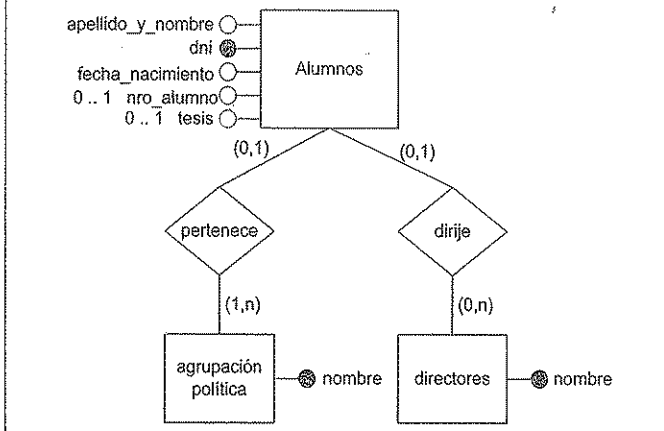
Suponga un entorno universitario que modela los alumnos de una facultad. Los alumnos pueden ser de grado o posgrado. Los alumnos de grado tienen asignado un número de legajo, dato que no tienen los alumnos de posgrado. Estos últimos deben tener definido un trabajo de tesis y su director. Algunos alumnos de grado pueden pertenecer a agrupaciones políticas, pero no en forma obligatoria. Se debe tener en cuenta que existe otro grupo de alumnos, los ingresantes a la universidad, que si bien son considerados alumnos, como aún no tienen asignado número de legajo, no son parte del universo de alumnos de grado. La Figura 11.3 presenta el modelo conceptual generado por la herramienta CASER.

Figura 11.3



Al aplicar la primera alternativa de solución, se quitan las entidades hijas de la jerarquía, dejando solamente la entidad padre. La Figura 11.4 presenta dicha situación, donde se puede observar que el modelo queda más reducido. Sobre la entidad padre quedan los atributos que estaban contenidos en las entidades hijas; estos atributos son ahora opcionales y, además, el atributo número de legajo dejó de ser identificador. La cardinalidad de Alumnos en la relación dirige pasó a ser opcional en lugar de obligatoria, dado que un alumno de grado no tiene director de tesis asociado.

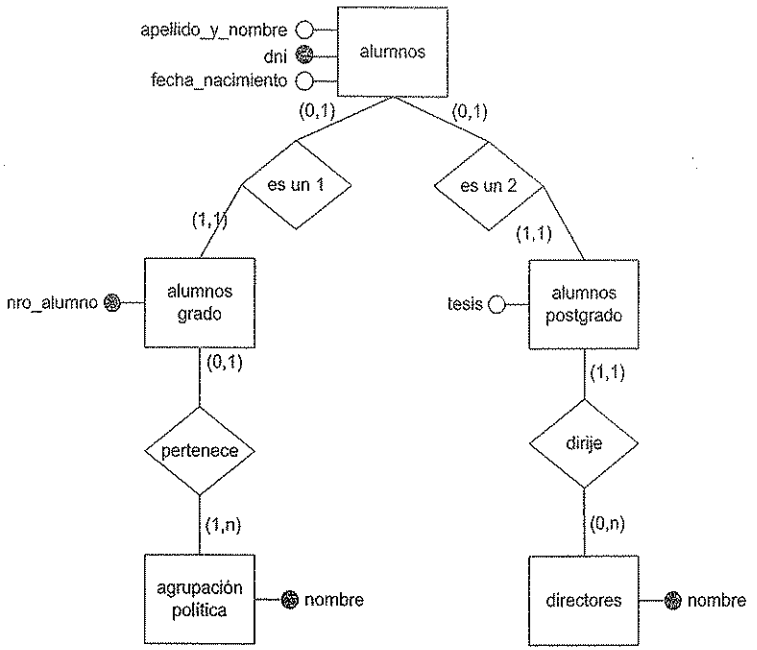
Figura 11.4



La segunda alternativa de solución no es viable. Si se quita la entidad alumnos, aquellos que son ingresantes no tienen cabida ni como alumnos de grado ni de posgrado.

Por último, la Figura 11.5 presenta el problema desde la perspectiva de la tercera alternativa de solución. Aquí, la jerarquía se transforma en una relación explícita Es_Un entre la entidad padre y cada una de las entidades hijas. Si bien esta solución es la más "extensa" (basta con contar el número de entidades y relaciones que se definen), es la que mejor capta los conceptos presentados en el modelo conceptual definido anteriormente.

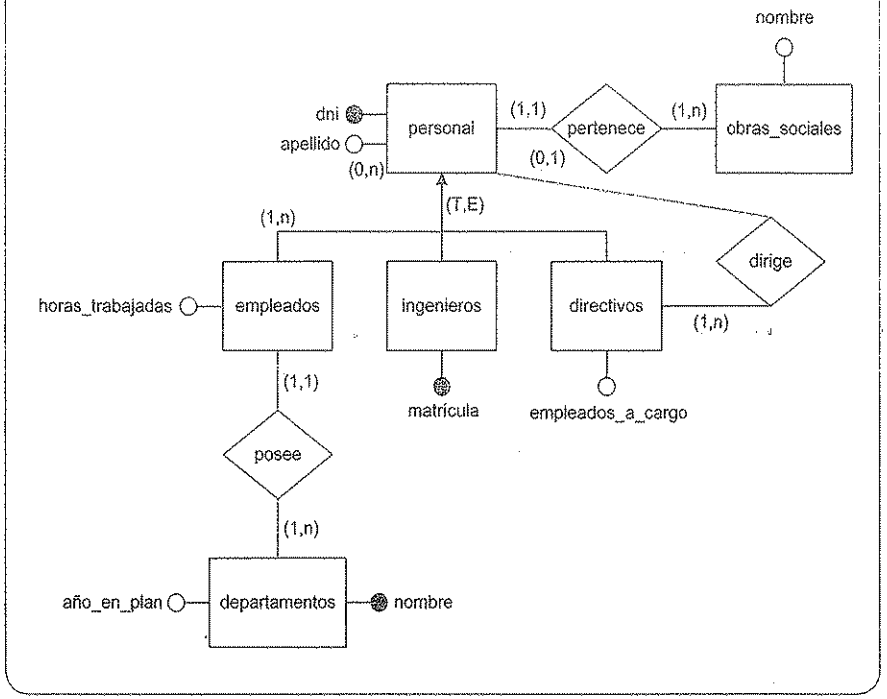
Figura 11.5



Segundo ejemplo de resolución de jerarquías

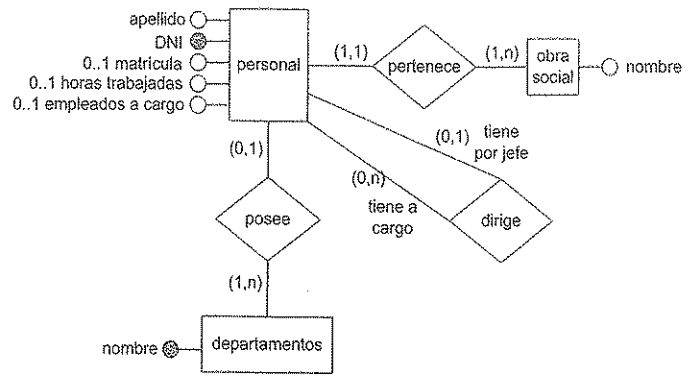
El segundo ejemplo plantea el problema de una empresa de comunicaciones. En dicha empresa, el personal se cataloga en tres grandes grupos: directivos, ingenieros y empleados. Para cada miembro del personal se administran sus datos personales, así como su obra social. Los empleados tienen ligadas las horas trabajadas y el departamento en el cual prestan funciones. De cada ingeniero se conoce su número de matrícula. De los directivos se conoce cuántas personas tienen a cargo y a quién dirige cada uno de ellos. La Figura 11.6 presenta el modelo conceptual del problema.

Figura 11.6



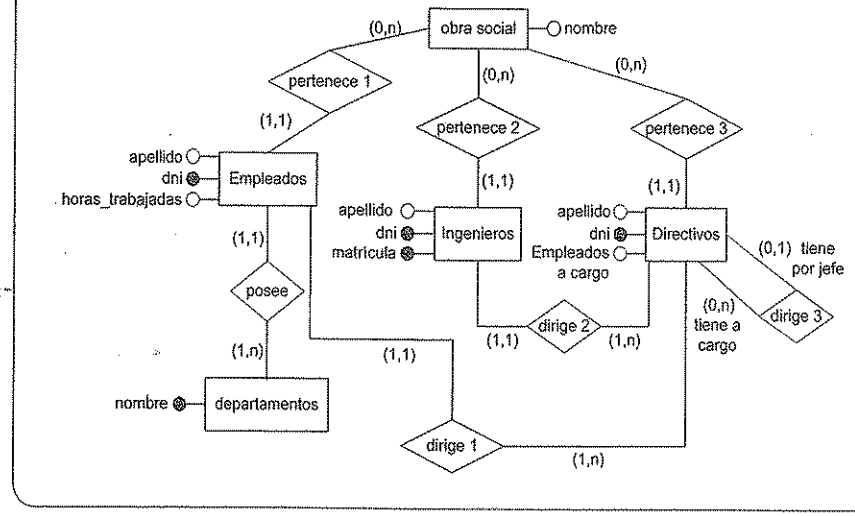
Si se aplica la primera alternativa de solución, se quitan las entidades hijas y solamente se deja la entidad padre (personal). El modelo se reduce. Sin embargo, resulta mucho más compleja su lectura, dado que aparecen varios atributos no obligatorios y algunas de las cardinalidades se ven afectadas, como por ejemplo pertenece (entre personal y departamento). La Figura 11.7 presenta en forma gráfica el resultado obtenido.

Figura 11.7



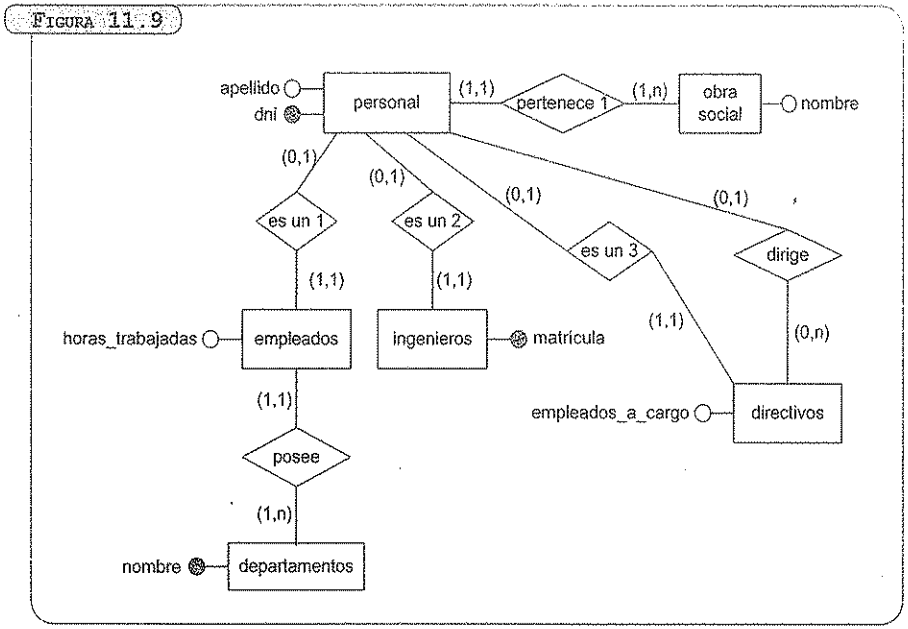
En este caso, por tratarse de una cobertura total, es posible aplicar la segunda solución. Así, se quita la entidad padre (personal) y se dejan solamente las entidades hijas. El modelo resultante aumenta su complejidad. La relación que existía entre personal y obras sociales se convierte, ahora, en tres relaciones, y su cardinalidad se ve afectada. Asimismo, la relación entre personal y directivos también se multiplica por tres, ahora con empleados, ingenieros y con los mismos directivos. La Figura 11.8 muestra el resultado final de la segunda solución.

Figura 11.8



Finalmente, es la tercera solución la que capta mejor la naturaleza del problema. El número de entidades y relaciones finales no se ve afectado sobremanera, y la información planteada en el modelo conceptual es referenciada en el modelo lógico de una forma clara y concisa. La Figura 11.9 presenta este caso.

Figura 11.9



Conclusiones

Se han estudiado tres alternativas para la resolución de las jerarquías. En los modelos donde no se presenten coberturas parciales, las tres soluciones son válidas. Aquellos que presenten una cobertura parcial no podrán eliminar a la entidad padre del problema. Es responsabilidad del diseñador de la BD la elección de la política para eliminar las jerarquías.

Se puede afirmar que la tercera alternativa de solución es la que capta mejor la esencia de la herencia y, por ende, la que resulta más interesante aplicar. Sin embargo, esta solución es la que genera mayor número de entidades y relaciones en el modelo final. Esto podría significar, a futuro, problemas de *performance* en la utilización de la BD.

Los subconjuntos, caso especial de jerarquías, tienen una cobertura parcial exclusiva. En este caso, tanto la primera como la tercera alternativa de solución son aplicables, aunque la segunda queda descartada.

Partición de entidades

El motivo de partir una entidad es reorganizar la distribución de las entidades en un conjunto de entidades (partición horizontal) que la componen, o de los atributos (partición vertical) que conforman cada conjunto de entidades.

Para ambos casos, un objetivo de la partición es mejorar la *performance* de las operaciones futuras sobre la BD. Otra meta es mejorar los niveles de seguridad de la BD, para lo cual se otorgan determinados derechos de acceso a los usuarios de la BD.

Una partición horizontal permite separar las entidades que conforman un conjunto. Suponga que se define en el modelo conceptual un conjunto de entidades persona que agrupa tanto a los clientes como a los proveedores de cierta organización. En este caso, sobre el modelo lógico se puede decidir generar una partición horizontal, definiendo dos conjuntos de entidades: uno para clientes y otro para proveedores.

La partición vertical, en cambio, analiza un conjunto de entidades con múltiples atributos. Suponga que para el conjunto personal de una organización se tienen definidos los siguientes atributos: nombre y apellido, DNI, domicilio, horas trabajadas, salario, obra social, título y habilidades laborales, entre otros. Cuando se manipula a un empleado, se está accediendo a múltiples atributos; sin embargo, si se desea conocer cuáles son los empleados con determinado título, no es necesario disponer del salario u obra social. Así, es posible realizar una partición de manera vertical al conjunto de entidades personal, generando tantos conjuntos como agrupamientos se puedan efectuar de los atributos que los componen. En este ejemplo se podría tener: datos personales, datos laborales, datos salariales como tres conjuntos diferentes.

Se debe notar que, para poder preservar la información tal y como está representada en el modelo conceptual, es necesario que cada uno de los conjuntos formados comparta un atributo que sea el identificador por el cual se puedan relacionar los datos.

Ejemplo

En el Capítulo 10, se definió un ejemplo integrador con el caso de una facultad, sus alumnos de grado y posgrado, en conjunto con los profesores y cursos que se dictan. La Figura 10.14 presentó el modelo conceptual resultante de dicho problema.

En este apartado, se toma como fuente de entrada el esquema conceptual definido para dicho ejemplo, y a partir de las reglas presentadas en este capítulo, se obtiene el esquema lógico respectivo.

Se comienza el análisis desde las jerarquías presentadas. En este ejemplo, existe una jerarquía con cobertura total exclusiva: personas es la entidad padre, y profesores y alumnos, las hijas. Además, se definen dos subconjuntos, uno para profesores visitantes y otro para alumnos graduados.

La jerarquía puede resolverse utilizando cualquiera de las reglas definidas, en tanto que la resolución de los subconjuntos no podrá quitar al padre dejando al hijo.

En este caso, la opción decidida por el diseñador de la BD consiste en dejar todas las entidades resolviendo explícitamente las jerarquías/conjuntos, como se indicó en la tercera alternativa de solución. La Figura 11.10 (pág. 256) presenta el esquema lógico resultante.

Sobre el modelo se han tomado otras decisiones de diseño lógico, las cuales se enumeran a continuación:

- El atributo compuesto datos filiatorios, conformado por apellido y nombre de la persona, fue resuelto definiendo los atributos simples directamente sobre la entidad personas.
- El atributo polivalente teléfono en la entidad personas fue reemplazado por un atributo simple monovalente. En este caso se decide generar un atributo con un dominio string[100], donde el usuario puede definir múltiples teléfonos sin complicar la solución.
- El atributo polivalente títulos fue resuelto generando una nueva entidad, donde se definen los títulos válidos, y una relación entre esta nueva entidad y profesores.
- Los ciclos de entidades no fueron analizados, ya que su permanencia se considera de mayor importancia para la *performance* final de la BD a generar.

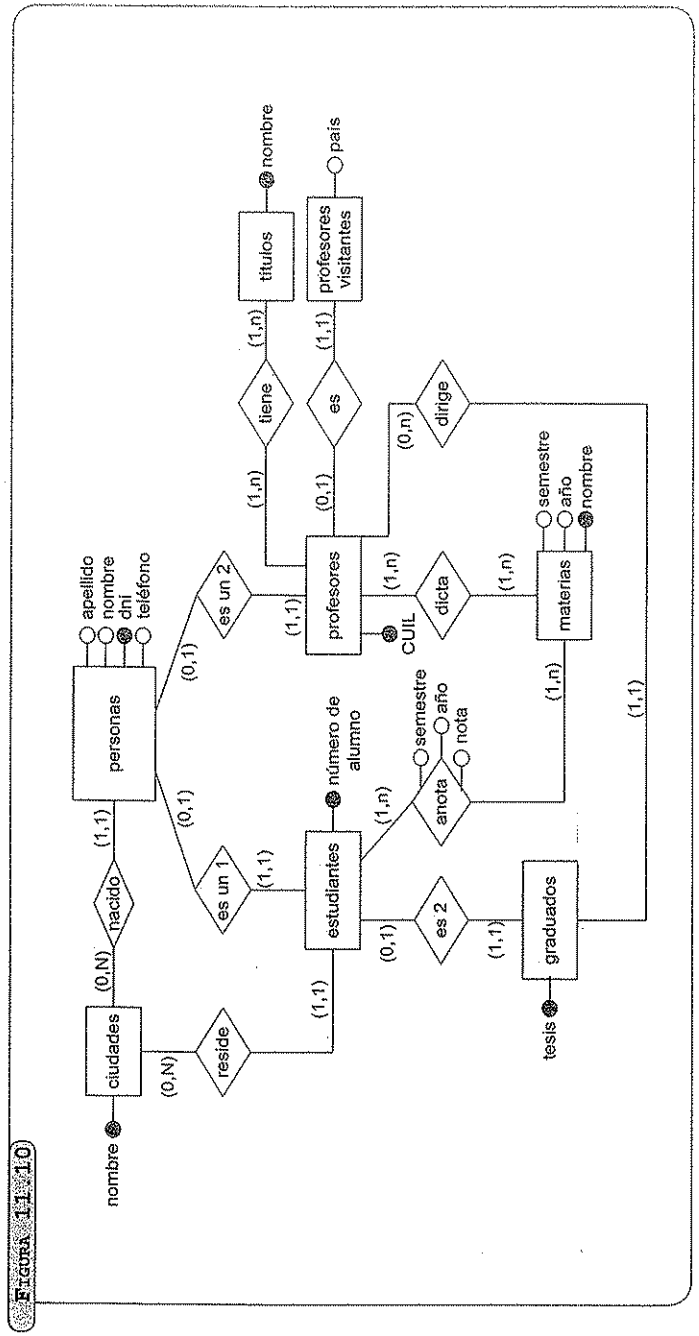


Figura 11.10

Cuestionario del capítulo

1. ¿Cuál es el objetivo de generar un esquema lógico de la BD?
2. ¿Cuántos esquemas lógicos diferentes se podrían generar a partir de un esquema conceptual?
3. ¿En qué se debe basar la decisión que se tome respecto de un atributo derivado?
4. ¿La respuesta anterior es aplicable, además, a los ciclos de relaciones entre entidades? Justifique.
5. ¿Por qué motivo las jerarquías no son representables en el esquema lógico?
6. ¿La respuesta anterior es aplicable sobre cualquier esquema lógico?
7. ¿Cuál solución para las jerarquías resulta más genérica en su aplicación?

Ejercitación

1. Revise los Ejercicios 1 y 2 del Capítulo 10. Sobre cada modelo conceptual de datos generado, aplique las reglas que permitan convertirlos en un esquema lógico.
2. Resuelva las jerarquías planteadas en el Ejercicio 3 del Capítulo 10, aplicando en cada caso la solución que considere más oportuna.

CAPÍTULO 12

Modelado físico (relacional)

Objetivo

El modelo de datos relacional fue introducido en la década del setenta por Codd, con amplia difusión por motivos de simplicidad, y por estar sustentado en fundamentos matemáticos básicos. El modelo relacional utiliza el concepto de relación matemática como elemento constitutivo y se basa en los principios del álgebra, la teoría de conjuntos y la lógica de predicados.

El objetivo de este capítulo es presentar los pasos necesarios que permiten generar un modelo físico de datos (sobre el modelo relacional) a partir del esquema lógico generado en el Capítulo 11. La operatoria descripta para lograr la conversión no contempla el desarrollo sobre un SGBD específico, solamente se presentan los mecanismos necesarios para generar los elementos que componen al modelo relacional de datos, a partir de las entidades, relaciones, atributos e identificadores. Posteriormente, el administrador de la BD deberá seleccionar los dominios de aplicación de acuerdo con el SGBD elegido, pero esta acción está fuera del alcance de este libro.

En la primera parte de este capítulo, se describen los componentes del modelo relacional en forma detallada. Luego, se discuten aspectos vinculados con identificadores y definición de claves.

A continuación, se presenta con ejemplos la conversión de los elementos del esquema lógico al modelo físico. Por último, se discute con detalle el concepto de integridad referencial, y su incidencia sobre el modelo de datos.

Conceptos básicos del modelo relacional

El modelo relacional representa a una BD como una colección de archivos denominados tablas, las cuales se conforman por registros. Cada tabla se denomina relación, y está integrada por filas horizontales y columnas verticales. Cada fila representa un registro del archivo y se denomina **tupla**, mientras que cada columna representa un atributo del registro.

La definición matemática de las relaciones se desarrolla a partir de la noción de dominio. Un dominio representa un conjunto de posibles valores para un atributo. Como un dominio restringe los valores del atributo, puede considerarse como una restricción.

Matemáticamente, otorgar un dominio a un atributo significa que todos los valores de ese atributo deben ser elementos del conjunto especificado. Ejemplos de tipos de dominios son: enteros, cadenas de texto, fecha, entre otros.

A fines prácticos, en el resto del capítulo se referirá con el término de "tabla" a las relaciones obtenidas en el modelo relacional.

Eliminación de identificadores externos

El primer paso en la conversión del esquema lógico hacia el esquema físico consiste en la eliminación de los identificadores externos. Cada una de las entidades que conforman el esquema lógico debe poseer sus identificadores definidos en forma interna. Para lograr esto, se deberán incorporar, dentro de la entidad que contenga identificadores externos, aquellos atributos que permitan la definición del identificador de forma interna a la entidad.

La Figura 12.1 a) presenta un ejemplo donde la entidad Empleados tiene un identificador externo (Número+Nombre de Departamento). Cada empleado se diferencia del resto a partir de su Número más el Departamento donde trabaja; esto significa que un empleado puede tener el mismo Número en diferentes Departamentos.

Por lo tanto, para realizar la conversión es necesario tomar el identificador de Departamento e incorporarlo como atributo de la entidad Empleados. Esta acción se refleja en la Figura 12.1 b).

Figura 12.1

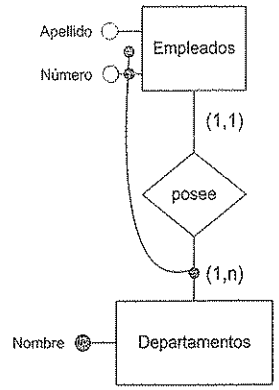


Figura 12.1 a)

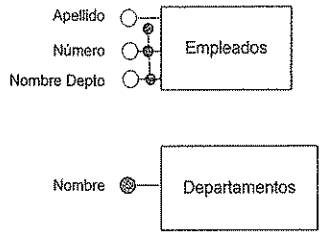


Figura 12.1 b)

La pregunta a realizarse es cuál identificador de la entidad Departamentos se debería seleccionar en caso de que haya más de uno. La respuesta es simple: se deberá elegir el identificador más representativo, es decir, el que posteriormente será definido como clave primaria.

El siguiente apartado aborda el tema de selección de identificadores, clave primaria y clave candidata.

Selección de claves: primaria, candidata y secundaria

En el Capítulo 10, se presentaron los elementos básicos que conforman un modelo conceptual. Allí se establecieron las bases que permitieron definir el concepto de identificador.

Para el usuario de la BD, el concepto de clave primaria y/o candidata no es importante. Para este actor solamente es necesario que estén definidos identificadores que permitan distinguir una entidad del conjunto de entidades. En cambio, el administrador de la BD debe decidir cuáles de los identificadores se convierten en clave primaria o candidata. Este proceso se realiza al construir el modelo físico.

Cuando se genera el esquema físico sobre el modelo relacional, se debe decidir el criterio de definición de clave primaria a partir de los identificadores reconocidos en cada entidad.

Si una entidad solo tiene definido un identificador, ese identificador es clave primaria de la tabla. Si la entidad tuviese definidos varios identificadores, la selección de la **Clave Primaria (CP)** debería realizarse del siguiente modo:

- Entre un identificador simple y uno compuesto, debería tomarse el simple, dado que así es más fácil de tratar y/o usar.
- Entre dos identificadores simples, se debe optar por aquel de menor tamaño físico.
- Entre dos identificadores compuestos, se debería optar por aquel que tenga menor tamaño en bytes. De ese modo, al construir un índice utilizando un árbol *B* como estructura, es posible almacenar mayor cantidad de claves por nodo.

Las consideraciones anteriores definen, en general, el criterio más adecuado para elegir la CP. El resto de los identificadores será definido como **Clave Candidata (CC)**. Sin embargo, los SGBD ofrecen una alternativa que resulta ser la más conveniente. Para presentar esta alternativa se debe recordar la idea que motiva generar una CP.

Un archivo tiene asociada solamente una CP, en tanto que puede tener asociadas varias CC y/o secundarias.

El acceso físico al archivo de datos se logra a partir del uso de la CP. El resto de las claves pueden utilizarse para generar índices secundarios, los cuales no referencian físicamente al archivo de datos, sino al índice primario.

Por lo tanto, la elección de la CP, que será utilizada para construir el índice primario, debe ser muy cuidadosa.

Cualquier CP elegida que sea directamente tratable por el usuario puede sufrir borrados o modificaciones. Estos posibles cambios influirán negativamente en la *performance* final de la BD, dado que impactarán en los índices primarios y/o secundarios.

Los SGBD de la actualidad presentan una alternativa de tratamiento para las CP, a través del uso de un tipo de dominio denominado

Autoincremental. Así, una tabla que tenga un atributo Autoincremental tiene definida una CP que es tratada por el SGBD en forma exclusiva. El usuario solamente tiene permitida la operación de consulta sobre la CP, es decir, no la puede generar, borrar ni modificar.

Esta elección, combinada con el concepto de integridad referencial que se presenta en el último apartado de este capítulo, genera una CP que actúa de la forma más eficiente posible, mejorando la *performance* final de la BD.

Concepto de superclave

Una superclave es un conjunto de uno o más atributos que permiten identificar de forma única una entidad de un conjunto de entidades. Presentada de esta manera, una superclave es equivalente a una CP o una CC. Sin embargo, una superclave puede contener atributos innecesarios. Suponga que se tiene la siguiente tabla:

Personal = (DNI, #Empleado, Nombre, Dirección, Fecha_Nacimiento)

Tanto DNI como # Empleado permiten identificar unívocamente a una persona del conjunto. Entonces, DNI y #Empleado son superclaves. Pero también, DNI junto a Nombre, o #Empleado más Fecha_nacimiento, representan posibles superclaves.

Si un atributo es superclave, entonces también lo es cualquier superconjunto que incluya a dicho atributo. Una CP o CC es una superclave que no admite a un subconjunto de ella como superclave.

El concepto de superclave es muy importante en el proceso de demostración de normalización de una BD, a tratar en el Capítulo 13.

Conversión de entidades

El proceso de conversión para obtener el esquema físico de una BD comienza con el análisis de las entidades definidas en el modelo lógico. En general, cada una de las entidades definidas se convierte en una tabla del modelo.

Si en el Capítulo 11 se observa la Figura 11.2, tiene definidas dos entidades: docentes y títulos; el proceso de conversión genera entonces dos tablas:

DOCENTES = (Apellido_y_Nombre, DNI, CUIL)
TITULOS = (Nombre)

Si se consideran las definiciones respecto de CP dadas en este capítulo, la conversión debería generar las siguientes tablas:

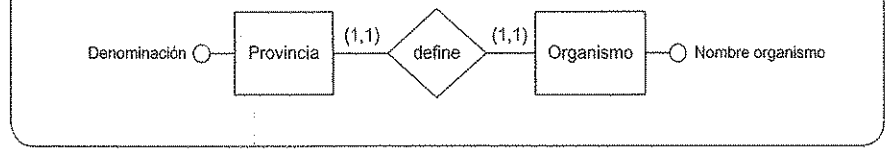
DOCENTES = (idDocente, Apellido_y_Nombre, DNI, CUIL)
TITULOS = (idTitulo, Nombre)

donde los atributos idDocente e idTitulo se agregan en este momento y representan sendos dominios autoincrementales, definiendo la CP de las tablas Docentes y Títulos, respectivamente (por ese motivo aparecen subrayados, indicando que son CP).

En el ejemplo, el DNI o el CUIL de la tabla Docentes podrían ser definidos como CC. Esta representación debe hacerse efectiva sobre un SGBD de mercado; en la conversión al modelo físico, las CC no son indicadas.

El proceso de conversión muestra que todas las entidades deben transformarse en tablas. Sin embargo, puede existir una excepción a esta regla. Suponga el siguiente caso: se debe modelar un problema que en una de sus partes indica que "cada provincia define un organismo encargado del control de los recursos naturales de dicha provincia". Para la solución del problema, el diseñador de la BD genera un modelo, que en alguna de sus partes incluye la solución presentada en la Figura 12.2.

Figura 12.2



Si se aplica el proceso de conversión definido anteriormente, deberían generarse dos tablas, una para provincias y otra para organismos. Sin embargo, se puede notar que, para este problema, la existencia de una provincia determina la existencia de un organismo, y viceversa. La cardinalidad definida entre ambas entidades es uno a uno con participación obligatoria. Por este motivo, la solución más viable en el proceso de conversión al esquema físico consiste en generar una única tabla que aglutine los atributos de ambas:

ORGANISMOS (idOrganismo, nombre_del_organismo, nombre_de_provincia)

Esta solución es la indicada cuando existe una relación uno a uno con cobertura total entre dos entidades.

Conversión de relaciones. Cardinalidad

El proceso de conversión continúa con el análisis de las relaciones. Se deben analizar tres situaciones diferentes. Estos tres casos tienen que ver, básicamente, con la cardinalidad existente entre las relaciones definidas: muchos a muchos, uno a muchos y uno a uno. En este apartado, se discute y ejemplifica cada uno de estos casos con sus variantes.

Cardinalidad muchos a muchos

La resolución de las relaciones con cardinalidad **muchos a muchos** resulta la más sencilla de implementar. La solución propuesta es independiente de la cardinalidad mínima definida, que puede ser en este caso obligatoria u opcional, pero la solución sigue siendo la misma.

Si se continúa con el análisis del ejemplo de la Figura 11.2, en el Capítulo 11, se determinó hasta el momento que las entidades DOCENTES y TITULOS se definían como tablas. En la primera estarán indicados todos los docentes que conforman el problema, en tanto que los títulos disponibles estarán contenidos en la segunda tabla.

Para indicar, ahora, que un docente posee un título, o que un título es poseído por un docente, se deberá definir una nueva tabla, producto de la conversión de la relación posee. El formato de esta tabla será:

POSEE (idDocente, idTitulo)

La relación *N* a *N* se convierte en tabla, conformada por los atributos que definen la CP de cada una de las entidades que relaciona. En este caso, ambos atributos generan la CP de la nueva tabla.

De acuerdo con la política de definición de CP establecida anteriormente en este capítulo, la tabla que se ha generado podría tener la siguiente estructura:

POSEE (idPosee, idDocente, idTitulo)

En este caso, la tabla POSEE define su propia CP: idPosee, con un dominio nuevamente Autoincremental. La ventaja de esta última definición es que se dispone de una CP conformada por un atributo simple.

En el Capítulo 10, la Figura 10.5 presenta otro modelo donde se define una relación *N* a *N*. De acuerdo con el proceso de conversión definido, una solución posible para dicho caso sería:

ALUMNOS = (idAlumno, Apellido_y_Nombre, DNI, NroAlumno, Telefonos)

MATERIAS = (idMateria, Nombre, Año_curso)
CURSA = (idAlumno, idMateria, año_que_cursa, resultado_obtenido)

Se puede notar que la solución dada en este caso para el atributo polivalente Telefono se plantea como si fuese un atributo simple monovalente.

Respecto de la solución de la relación, CURSA es una tabla que contiene las CP de Alumnos y Materias en conjunto con los atributos que estaban definidos en la relación. Se puede notar, además, que la CP de CURSA está integrada por los atributos idAlumno, idMateria y año_que_cursa. Esto se debe a que un alumno puede recursar una materia y, entonces, los atributos idAlumno+idMateria pueden repetirse en dos años diferentes y, por este motivo, no son suficientes para definir la CP.

Una alternativa de solución para CURSA puede ser:

CURSA = (idCursa, idAlumno, idMateria, año_que_cursa, resultado_obtenido)

Otra vez se define un nuevo atributo, idCursa, con dominio Autoincremental, que permite simplificar la elección de la CP.

Se puede notar en la Figura 10.5 que la cardinalidad mínima, en este ejemplo, es con participación parcial para materias; sin embargo, esta situación no afecta a la resolución del problema.

Cardinalidad uno a muchos

La solución para la cardinalidad **uno a muchos** tiene dos alternativas posibles: puede ocurrir que la relación se transforme o no en una tabla. La decisión deberá ser tomada en función de la cardinalidad mínima definida y de las decisiones de diseño que tome el administrador de la BD.

Uno a muchos con participación total

En el Capítulo 11, la Figura 11.7 presenta las entidades Personal y Obra Social. Entre ambas está definida la relación pertenece, con cardinalidad 1 a *N* y participación total de ambos lados. Esto significa que cada empleado tiene siempre una obra social y que cada obra social tiene, al menos, un empleado que la elige. La regla de conversión para entidades produce:

PERSONAL = (idPersonal, Apellido, DNI, Matricula, Horas_trabajadas, Empleados_a cargo)
OBRAS_SOCIALES = (idOS, nombre)

Cada obra social puede tener múltiples empleados que la eligen; sin embargo, cada empleado selecciona una y solo una obra social. Entonces se puede incorporar la obra social como un atributo más del empleado, estableciendo de este modo el vínculo sin necesidad de generar otra tabla:

PERSONAL = (idPersonal, Apellido, DNI, Matrícula, Horas_trabajadas, Empleados_a cargo, idOS)

El atributo idOS, CP en Obras_Sociales, es una clave foránea en Personal. Se denomina **Clave Foránea (CF)** a un atributo o grupo de atributos de una tabla (Personal) referida a un atributo o grupo de atributos que en otra tabla (Obras_Sociales) son CP. Sobre estos atributos se establece un concepto que se denomina integridad referencial, el cual será desarrollado en el último apartado del presente capítulo.

Una CF es, además, clave secundaria en la tabla donde aparece. En la tabla Personal, pueden existir varios empleados que tengan la misma obra social definida; por lo tanto, el atributo idOS puede repetirse para varias tuplas, conformando, por ende, una clave secundaria en Personal.

Uno a muchos con participación parcial del lado de muchos

El siguiente caso a analizar está presente en el Capítulo 10, en la Figura 10.3 c). Aquí aparecen las entidades Personas y Localidades, y entre ambas, la relación nacido_en. La cardinalidad definida establece que una persona tiene definido siempre un lugar de nacimiento, pero puede ocurrir que en la lista de ciudades haya alguna en la que no haya nacido nadie, y otra en la que hayan nacido muchas personas. La cardinalidad definida es uno a muchos, pero con participación opcional del lado de muchos.

Si se intenta resolver el problema de conversión al modelo físico de acuerdo con las pautas definidas, las tablas resultantes serán:

PERSONAS = (idPersona, Nombre_y_Apellido, fecha_nacimiento, idLocalidad)
LOCALIDADES = (idLocalidad, Nombre)

Se asume que los atributos Nombre_y_Apellido y fecha_nacimiento están definidos para la entidad Personas, y que el atributo Nombre está definido para la entidad Localidades. El atributo idLocalidad en la tabla Personas actúa, nuevamente, como CF. Como todas las personas nacieron indefectiblemente en una localidad, la definición de este atributo no presenta ningún inconveniente; en tanto que si en una localidad no ha nacido ninguna persona, su CP no figurará en ninguna

tupla de PERSONAS. La solución para este caso es similar a la primera situación presentada.

Si la conversión al modelo físico se hubiese realizado de la forma

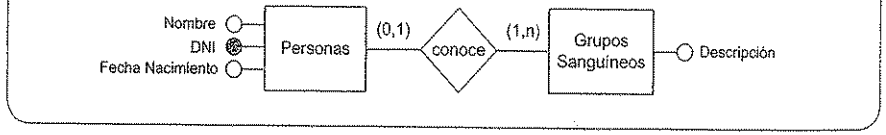
PERSONAS = (idPersona, Nombre_y_Apellido, fecha_nacimiento)
LOCALIDADES = (idLocalidad, Nombre, idPersona)

como en el ejemplo definido existe participación parcial del lado de muchos –es decir, al menos existe una localidad en la que no ha nacido ninguna persona–, el atributo idPersona para la tupla de esa localidad sería nulo. Entonces, no es conveniente incluir la CF en la tabla que posee participación parcial, dado que existirán tuplas con valores nulos para ese atributo. En consecuencia, esta conversión queda descartada, por lo que se opta como solución por la primera variante presentada.

Uno a muchos con participación parcial del lado de uno

La tercera situación que se plantea es una relación uno a muchos con participación total del lado de muchos, pero parcial del lado de uno. La Figura 12.3 presenta un ejemplo de esta situación. Cada persona, si bien tiene un grupo sanguíneo definido, no siempre lo conoce; por lo tanto, para el modelo de datos es importante reflejar esta situación. Una persona a lo sumo puede tener un grupo sanguíneo, pero puede ocurrir que no sepa cuál es. Entonces, la cardinalidad del lado de uno es parcial.

Figura 12.3



La resolución del ejemplo, utilizando el mismo razonamiento que en los casos anteriores, quedaría:

PERSONAS = (idpersona, nombre, dni, fecha_nacimiento, idgruposanguineo)
GRUPOS_SANGUINEOS = (idgruposanguineo, descripcion)

En este caso, la solución planteada define el atributo idgruposanguineo dentro de la tabla PERSONAS, pero este atributo debe permitir valores nulos para aquellas personas que no conozcan su grupo sanguíneo. Igualmente el atributo idgruposanguineo constituye una CF.

Como la utilización de atributos nulos no es recomendada, se debe plantear una alternativa de solución que consiste en mantener las tablas PERSONAS y GRUPOS_SANGUINEOS, pero sin agregar el atributo idgruposanguineo a la tabla PERSONAL. Así se debe definir una nueva tabla que contendrá la CP de PERSONAS y GRUPOS_SANGUINEOS, tal como se resolvió la cardinalidad muchos a muchos. La solución propuesta es:

PERSONAS = (idpersona, nombre, dni, fecha_nacimiento)
GRUPOS_SANGUINEOS = (idgruposanguineo, descripcion)
CONOCE = (idpersona. idgruposanguineo)

Si una persona conoce su grupo sanguíneo, esta información se incorpora como una tupla en la tabla CONOCE, evitando de este modo atributos nulos.

La conclusión que se puede emitir en este punto es que la primera solución analizada genera una tabla menos, la información queda resumida y, por lo tanto, operar con ella es más sencillo. No obstante, se administran atributos con posibles valores nulos, y la situación es no deseada. En consecuencia, se opta por la segunda alternativa.

Uno a muchos con cobertura parcial de ambos lados

El último caso posible de cardinalidad uno a muchos sucede cuando la cardinalidad es parcial de ambos lados. Aquí se puede proceder igual que en el caso anterior. Hay dos soluciones factibles: aquella que genera atributos con valores nulos (no recomendable) y la variante que lo evita.

Cardinalidad uno a uno

Esta situación fue planteada al principio de este capítulo. Cuando la cardinalidad es **uno a uno** con participación total de ambos lados, se debe generar una sola tabla que contenga los atributos de ambas entidades.

Quedan para analizar los otros dos casos posibles: con participación parcial de un lado o de ambos.

En el Capítulo 11, la Figura 11.9 ejemplifica la participación parcial de un lado. En ella están definidas las entidades Personal, Empleados, Ingenieros y Directivos. Cada uno de estos tres últimos es un personal, y cada personal puede ser de uno de estos tres casos. La solución para convertir al esquema físico comienza definiendo la tabla PERSONAL:

PERSONAL (idpersonal, apellido, dni)

Luego se definen tres tablas, pero con una particularidad: el atributo idpersonal, CP de la tabla PERSONAL, también será la CP de cada una de las tres tablas definidas:

EMPLEADOS = (idpersonal, horas_trabajadas)
INGENIEROS = (idpersonal, matricula)
DIRECTIVOS = (idpersonal, empleados_a_cargo)

Como tanto un empleado como un ingeniero o directivo tienen las características de PERSONAL (recuerde que este caso constituía una jerarquía del modelo conceptual), es posible que se repita la misma CP. Se debe destacar que en estas tablas el atributo idpersonal no es autoincremental.

El caso de cardinalidad parcial de ambos lados no es una situación muy común. Sin embargo, la solución aconsejable en este caso es generar una tabla por cada entidad y otra tabla que involucre la relación. Esta última tabla deberá incluir las CP de las entidades que relaciona.

Casos particulares

Si bien con lo planteado hasta el momento la conversión al modelo físico es resuelta sin mayores inconvenientes, resulta interesante comentar dos casos particulares: las relaciones recursivas y las relaciones *n*-arias, particularmente las ternarias.

Se analiza el caso presentado en el Capítulo 10, en la Figura 10.3 d). La entidad Materias muestra una relación recursiva que define las correlativas. Suponga que se genera la tabla MATERIAS, con los siguientes atributos:

MATERIAS = (idmateria, nombre, año, duracion)

La cardinalidad de la relación Correlativas indica que una materia puede tener varias correlativas o ser correlativa de varias, generando un caso de muchos a muchos. Entonces, la solución será generar una nueva tabla:

CORRELATIVAS = (idmateria_original, idmateria_correlativa)

Se debe notar que, como la relación es recursiva, al armar la tabla CORRELATIVAS se toma la única CP disponible y se la replica.

La Figura 10.4 b) presenta una relación ternaria, cursa, entre alumnos, materias y aulas. Como la relación es muchos a muchos, la conversión se realiza:

ALUMNOS = (idalumno, nombre, dni, numero_alumno)
AULAS = (idaula, nombre)

MATERIAS = (idmateria, nombre, año, duración)
CURSA = (idalumno, idaula, idmateria)

La solución depende, nuevamente, de la cardinalidad definida en la relación.

Integridad referencial

La **Integridad Referencial (IR)** es una propiedad deseable de las BD relacionales. Esta propiedad asegura que un valor que aparece para un atributo en una tabla aparezca además en otra tabla para el mismo atributo.

La IR plantea restricciones entre tablas y sirve para mantener la consistencia entre las tuplas de dichas tablas.

Suponga que se generan las tablas:

FACTURAS = (idfactura, fecha, monto, idcliente)
CLIENTES = (idcliente, nombre, direccion)

El atributo idcliente en la tabla FACTURAS es una CF. Esta CF permite establecer IR entre la tabla FACTURAS y la tabla CLIENTES. Se debe notar que el atributo idcliente es CP de la tabla CLIENTES.

Para que exista IR entre dos tablas, necesariamente debe existir un atributo común. En general, el atributo común es CF en una tabla y CP en la otra tabla, aunque pueden presentarse excepciones a esta regla.

No es condición necesaria que entre dos tablas que tengan un atributo común esté definida la IR. Puede ocurrir que el diseñador de la BD no considere oportuna la definición de la IR entre dos tablas del modelo. No obstante, en general, es deseable definir la IR, a fin de permitir que el SGBD controle determinadas situaciones.

En el ejemplo anterior, se debe definir la IR entre las tablas FACTURAS y CLIENTES. La IR puede plantearse para dos casos posibles: intento de borrado o intento de modificación.

Cada SGBD tiene escenarios de definición de IR diferentes, pero en general cuando se la define se puede optar por estas situaciones:

- **Restringir la operación:** es decir, si se intenta borrar o modificar una tupla que tiene IR con otra, la operación se restringe, y no se puede llevar a cabo. En el ejemplo, si se intenta borrar una tupla de la tabla CLIENTES que tiene ligada al menos una tupla en la tabla FACTURAS, la operación no es permitida. Para poder borrar un cliente, este no debe poseer facturas, o las facturas deben ser borradas previamente. Se debería proceder de la misma forma si se intentara modificar la CP sobre la tabla CLIENTES.
- **Realizar la operación "en cascada":** en este caso, si se intenta borrar o modificar una tupla sobre la tabla donde está definida la CP de la IR, la operación se realiza en cadena sobre todas las tuplas de la tabla que tiene definida la CF. Para el ejemplo anterior, eliminar un cliente significaría eliminar en la misma operación todas las facturas de dicho cliente. En caso de modificar la CP de un cliente, se realizaría la modificación de las CF en las tuplas de la tabla FACTURAS.
- **Establecer la CF en nulo:** si se borra o modifica el valor del atributo que es CP, sobre la CF se establece valor nulo. Esta opción no es muy utilizada ni está presente en todos los SGBD.
- **No hacer nada:** en este caso se le indica al SGBD que no es necesario controlar la IR. Esta opción es equivalente a no definir restricciones de IR.

Cuestionario del capítulo

1. Defina el concepto de tabla o relación.
2. Defina el concepto de tupla.
3. ¿Qué diferencia existe entre un identificador y una CP o CC?
4. ¿Cuál es la diferencia entre CP y superclave?
5. ¿Por qué las relaciones muchos a muchos se resuelven de manera diferente que las relaciones uno a muchos?
6. ¿Bajo qué circunstancias una entidad no se transforma en tabla?
7. Explique el concepto de clave foránea.
8. Describa cómo se utiliza el concepto de integridad referencial.

Ejercitación

1. Discuta la solución cuando se pasa al esquema físico de las Figuras 10.12, 10.15 b) y 10.16.
2. Presente cómo quedarían las tablas al generar el esquema físico de la Figura 10.18 b).
3. A partir del modelo lógico generado en la Figura 11.10, determine cómo quedaría el esquema físico de dicho modelo.

Capítulo 13

Conceptos de normalización

Objetivo

El proceso de normalización de una BD consiste en aplicar una serie de reglas a las tablas obtenidas a partir del diseño físico. Una BD debe normalizarse para evitar la redundancia de los datos, dado que cuando se repite innecesariamente la información, aumentan los costos de mantenerla actualizada. Además, la normalización ayuda a proteger la integridad de la información de la BD.

El proceso de normalización fue propuesto por Codd y se aplica desde 1972. En este capítulo, se describen el propósito de la normalización y su incidencia en el proceso de diseño de BD relacionales.

Se definirá y ejemplificará el concepto de dependencia funcional en las tablas del modelo de datos, con todas sus variantes: las dependencias funcionales parciales, transitivas, de Boyce-Codd y, por último, las dependencias multivaluadas.

Se identificarán e ilustrarán, además, los problemas potenciales asociados con la redundancia de datos, en una BD que no se encuentre normalizada. Por último, se presentarán todos los estadios de normalización, que se conocen bajo el nombre de "formas normales".

Concepto de normalización

La **normalización** es un mecanismo que permite que un conjunto de tablas (que integran una BD) cumpla una serie de propiedades deseables. Estas propiedades consisten en evitar:

- Redundancia de datos.
- Anomalías de actualización.
- Pérdida de integridad de datos.

La ventaja de utilizar una BD normalizada consiste en disponer de tablas, cuyos datos serán para el usuario de fácil acceso y sencillo mantenimiento.

Algunos autores proponen normalizar el modelo de datos durante el proceso de diseño (conceptual, lógico y físico) de la BD. En este libro, se decide aplicar el proceso de normalización sobre el esquema físico producto del proceso de diseño, es decir, una vez finalizado el proceso de diseño.

El proceso de normalización es un proceso deseado pero no estrictamente necesario. Durante dicho proceso, se toman decisiones que producen cambios en las tablas, los cuales pueden afectar los tiempos de acceso a los datos almacenados en esas tablas. Por lo tanto, es posible que, en determinadas situaciones, la solución obtenida no sea conveniente para la operatoria sobre la BD, pues el tiempo de respuesta de una consulta resulta muy elevado. Ante esas situaciones, la normalización puede quedar de lado y priorizarse la *performance* final de la BD. Si bien esta afirmación puede ser controvertida desde el punto de vista teórico, en la práctica profesional es habitual.

El proceso de normalización puede aplicarse de forma empírica o puede ser demostrado a partir del uso de teoremas. Este libro se limitará a presentar el proceso de normalización a partir de ejemplos y buenas prácticas.

Redundancia y anomalías de actualización

El objetivo principal del proceso de normalización es diseñar una BD que **minimice** la redundancia de información. Para esto, es necesario reagrupar los atributos de cada tabla del modelo. El proceso de diseño genera redundancia de información y en determinadas situaciones esta redundancia es necesaria. Por ejemplo, cuando se resuelve una

relación muchos a muchos del esquema lógico, se genera una nueva tabla con las CP de las dos entidades que relaciona: esta solución repite información. Sin embargo, esta redundancia es necesaria para representar en el modelo físico la relación. Este tipo de redundancia se denomina redundancia deseada y no afecta al proceso de normalización.

Por el contrario, existen otros casos de redundancia, denominadas no deseadas, que generan anomalías de actualización cuando se opera sobre la BD. Estas anomalías se clasifican en tres grupos: anomalías de inserción, de borrado y de modificación de la BD. En este apartado, se presenta y describe mediante ejemplos cada una de ellas.

Anomalías de inserción

Para presentar el concepto de anomalías de inserción, suponga por ejemplo que se dispone de una tabla con los datos laborales de los empleados de una organización:

EMPLEADOS = (idempleado, nombre, salario, fecha_ingreso, iddepto, nombre_depto)

La tabla indica el nombre del empleado, su salario, la fecha de ingreso del empleado en la organización y en qué departamento se encuentra trabajando actualmente.

Si se agrega un nuevo empleado, se debe indicar toda la información, incluyendo repetir el nombre del departamento donde trabaja, aunque sea un departamento ya existente en la tabla. Se debe tener especial cuidado en describir al departamento de forma similar a lo que se hubiese hecho anteriormente. Por ejemplo, en la Tabla 13.1 se agrega al Empleado Danae López en el departamento de Electrónica y Computación, pero si se insertara como Computación y Electrónica, generaría inconvenientes con el departamento ya cargado.

Tabla 13.1

IdEmpleado	Nombre	Salario	Fecha_Ingreso	IdDepto	Nombre_depto
1	Ariel Sobrado	$2.500	01/04/2005	1	Contabilidad
2	Belen Almazan	$3.000	01/06/2007	2	Electrónica y Computación
3	Augusto Villamonte	$2.800	01/09/2003	1	Contabilidad
4	Karen Poch	$2.900	01/12/2007	3	Ventas
5	Danae Lopez	$3.000	01/06/2009	2	*Computación y Electrónica*

Se debe notar que la información presentada en la Tabla 13.1 es incorrecta. El departamento cuyo código es 2 no puede tener nombres diferentes. Esta situación se produce al generar redundancia de información. Cada vez que se agrega un empleado de un departamento existente, debe ingresarse el nombre del departamento, lo cual es innecesario.

Además, podría producirse sobre el ejemplo anterior otra situación que genere anomalía de inserción. Suponga que se crea el departamento de Marketing. Entonces se debe insertar una tupla en la Tabla 13.1 que indique tal situación, y al no haber empleados asignados para ese departamento, los datos correspondientes a nombre, salario y fecha de ingreso quedan nulos.

Anomalías de borrado

Las anomalías de borrado se producen en situaciones similares al último ejemplo planteado. Suponga que sobre la Tabla 13.1 se elimina a la empleada Karen Poch, que trabaja en el departamento de Ventas. Al borrar esa tupla, en la misma operación se borra información del departamento donde trabaja, y como era la única empleada registrada para ese departamento, se pierde a Ventas como departamento de la organización.

Anomalías de modificación

Continuando con el ejemplo de la Tabla 13.1, suponga que el departamento de Electrónica y Computación se modifica y pasa a llamarse departamento de Tecnología. Este cambio debe producirse necesariamente sobre cada una de las tuplas que corresponden a empleados del viejo departamento de Electrónica y Computación, lo que implica cambiar varios registros de la tabla. Además, si por cualquier inconveniente no se produjeran todos los cambios en cuestión, quedarían tuplas donde el iddepto sería 2 y el nombre del departamento sería Tecnología, en tanto que para otras tuplas con el mismo iddepto, el nombre del departamento sería Electrónica y Computación. Claramente, las situaciones planteadas son anómalas.

Dependencias funcionales

El concepto de dependencia funcional es uno de los más importantes cuando se diseña el esquema físico de una BD.

Una **Dependencia Funcional (DF)** representa una restricción entre atributos de una tabla de la BD. Se dice que un atributo *Y* depende funcionalmente de un atributo *X* (denotado por la expresión $X \rightarrow Y$), cuando para un valor dado de *X* siempre se encuentra el mismo valor para el atributo *Y*. Se debe notar que *X* e *Y* pueden representar, además, un conjunto de atributos.

Más formalmente, dadas dos tuplas cualesquiera de una tabla *t1* y *t2*, si $t1[X] = t2[X]$, entonces $t1[Y] = t2[Y]$.

Generalizando, el atributo *X* determina al atributo *Y*.

En una DF $X \rightarrow Y$, al atributo *X* se lo denomina determinante y al atributo *Y*, consecuente.

Dada la siguiente tabla:

ALUMNOS = (idalumno, nombre, dirección, tel, idcarrera)

se establecen las siguientes DF:

Idalumno → nombre
Idalumno → dirección
Idalumno → tel
Idalumno → idcarrera

El nombre del alumno depende funcionalmente de su id, y lo mismo ocurre con la dirección, el teléfono y el código identificatorio de la única carrera en la cual se encuentra inscripto. Se puede notar que de un atributo que es CP se desprenden DF al resto de los atributos de la tabla. Es decir, cualquier atributo depende funcionalmente de la CP.

La condición anterior resulta muy sencilla de demostrar. Dada la tabla ALUMNOS, no van a existir dos tuplas diferentes con el mismo Idalumno; por lo tanto, a partir de idalumno es posible determinar fehacientemente su nombre, dirección, teléfono e idcarrera.

Algo similar ocurre con cualquier atributo que haya sido definido como CC. La siguiente tabla de una BD representa a los departamentos de cierta compañía. El nombre del departamento es definido como CC, debido a que en la organización no pueden existir dos departamentos con igual denominación.

DEPARTAMENTOS = (iddepto, nombre, ubicación, cant_empleados)

Las DF existentes son:

Iddepto → nombre	iddepto → ubicación	iddepto → cant_empleados
Nombre → iddepto	nombre → ubicación	nombre → cant_empleados

Puede observarse en este caso que el iddepto depende del nombre, y viceversa. Esta condición solo se presenta entre un atributo que es CP y otro que es CC, o cuando ambos son CC.

Cuando se definen las DF de una tabla del modelo de datos, pueden generarse un número importante de casos. Es fundamental definir el conjunto mínimo de DF. Este conjunto tiene las siguientes características:

- El consecuente de la DF debe estar formado por un solo atributo. Si se define la DF $X \rightarrow Y$, Y debe ser un solo atributo.
- Si se define la DF $X \rightarrow Y$, no es posible encontrar otra dependencia $Z \rightarrow Y$ donde Z sea un subconjunto de atributos de X. En este caso se define a la DF como completa.
- No se puede quitar de un conjunto de DF alguna de ellas, y seguir teniendo un conjunto equivalente.

Dependencia funcional parcial

Una DF $X \rightarrow Y$ se denomina **parcial** cuando, además, existe otra dependencia $Z \rightarrow Y$, siendo Z un subconjunto de X.

Esta definición contradice la definición de conjunto mínimo de DF. En este caso, la generación de una DF parcial trae aparejada repetición de información y, consecuentemente, las anomalías de actualización ya discutidas.

Dada la siguiente tabla:

PEDIDOS = (idpedido, idproducto, descripciónproducto, fechapedido, cantidad)

que contiene los pedidos recibidos por un negocio de venta de productos por internet, se pueden observar las siguientes DF a partir de la CP definida:

Idpedido, idproducto → descripcionproducto
Idpedido, idproducto → fechapedido
Idpedido, idproducto → cantidad

Pero además, se puede observar que la descripción del producto vendido es un atributo que depende del código del producto. Asimismo, la fecha del pedido depende funcionalmente del código del pedido. Entonces se pueden definir:

Idproducto → descripcionproducto
Idpedido → fechapedido

Esas dos últimas DF se denominan parciales y determinan que el primer conjunto de DF definido no sea mínimo.

De acuerdo con las anomalías de actualización ya discutidas en este capítulo, a partir de las DF parciales, en la tabla PEDIDO pueden generarse anomalías de inserción, borrado o modificación. Los ejemplos que se pueden generar son equivalentes a los presentados anteriormente.

Se puede advertir que al definir DF con determinante múltiple, es posible generar situaciones de DF parciales. Si el determinante es simple, no es posible que una DF parcial exista.

Dependencia funcional transitiva

Una DF $X \rightarrow Y$ se denomina **transitiva** cuando existe un atributo Z, tal que $X \rightarrow Z$ y $Z \rightarrow Y$.

Esta definición contradice, nuevamente, la definición de conjunto mínimo de DF. En este caso, no se cumple la tercera propiedad: se puede quitar del conjunto de DF alguna de ellas ($X \rightarrow Y$) y seguir teniendo un conjunto equivalente.

Dada la siguiente tabla:

ALUMNO = (idalumno, nombre, direccion, idcarrera, nombre_carrera)

se establecen las siguientes DF a partir de la CP:

Idalumno → nombre
Idalumno → direccion
Idalumno → idcarrera
Idalumno → nombre_carrera

Además, se puede notar que el nombre de una carrera depende del código de la carrera en cuestión:

Idcarrera → nombre_carrera

En este ejemplo, X está representado por el atributo idalumno; Y, por el atributo nombre_carrera, y Z, por el atributo idcarrera. En el conjunto inicial definido, la DF Idalumno → nombre_carrera genera una DF transitiva. Nuevamente y tal como se analizó para DF parciales, las DF transitivas generan repetición innecesaria de información y las anomalías de actualización ya discutidas.

Otro ejemplo de DF no deseadas se plantea a continuación:

VENTAINMUEBLE = (idinmueble, idpropietario, nombre_propietario, valor_inmueble, idcomprador, nombre_comprador, idvendedor, comisión_venta)

Las DF existentes a partir de la CP son:

Idinmueble → idpropietario, nombre_propietario, valor_inmueble, idcomprador, nombre_comprador, idvendedor, comisión_venta

Se puede observar además que existen otras DF:

Idpropietario → nombre_propietario
Idcomprador → nombre_comprador
Idvendedor → comisión_venta

Cada una de estas DF es, además, transitiva.

Dependencia funcional de Boyce-Codd

La DF de Boyce-Codd está basada en DF que tienen en cuenta las CC de una tabla. Su definición formal es la siguiente:

Una DF *X* → *Y* se denomina **Dependencia Funcional de Boyce-Codd (DFBC)** cuando *X* no es una CP o CC, e *Y* es una CP o CC, o parte de ella.

Para analizar con más detalle la DFBC, se debe tener en cuenta que los determinantes de la DF sean siempre CP o CC.

Suponga que se define una tabla donde se registran, para cada alumno, las materias en las que se inscribe y quién es el docente a cargo del dictado de cada una de ellas; se sabe además que un docente solamente dicta una materia. La tabla tendrá la siguiente estructura:

DICTA = (nombre_alumno, nombre_materia, nombre_docente)

Existen dos CC para la tabla: (nombre_alumno, nombre_materia) y (nombre_alumno, nombre_docente). Conociendo al alumno y a la materia se puede determinar al docente, y conociendo al alumno y al docente se puede determinar la materia.

Además, en el problema propuesto, es posible determinar la materia que dicta cada profesor. Resumiendo lo anterior, se tienen las siguientes DF:

Nombre_alumno, nombre_materia → nombre_docente
Nombre_alumno, nombre_docente → nombre_materia

Hasta aquí los dos determinantes son, además, CC.

Pero existe una tercera DF:

Nombre_docente → nombre_materia

dado que un docente solamente puede dictar una materia. Esta última DF genera una DF de BC, el determinante no es CC ni CP, y el consecuente (nombre_materia) es parte de una CC. Esta condición genera repetición innecesaria de información, como se muestra en la siguiente tabla.

Tabla 13.2

Nombre_alumno	Nombre_materia	Nombre_docente
Viviana Castelli	IBD	Arturo Servetto
Aixa Mauriello	IS	Luciano Marrero
Emanuel Nucilli	IBD	Hugo Ramón
Alejandro Burakowsky	IS	Ariel Pasini
German Caseres	IBD	Arturo Servetto
Sebastian P. Escribano	IBD	Arturo Servetto
Paula Rispoli	IS	Luciano Marrero
Noelia Restelli	IBD	Hugo Ramón

Formas normales

La normalización es una técnica que permite analizar el esquema físico de datos buscando la mejor representación de cada una de las tablas, evitando la repetición de información y las anomalías de actualización que puedan surgir.

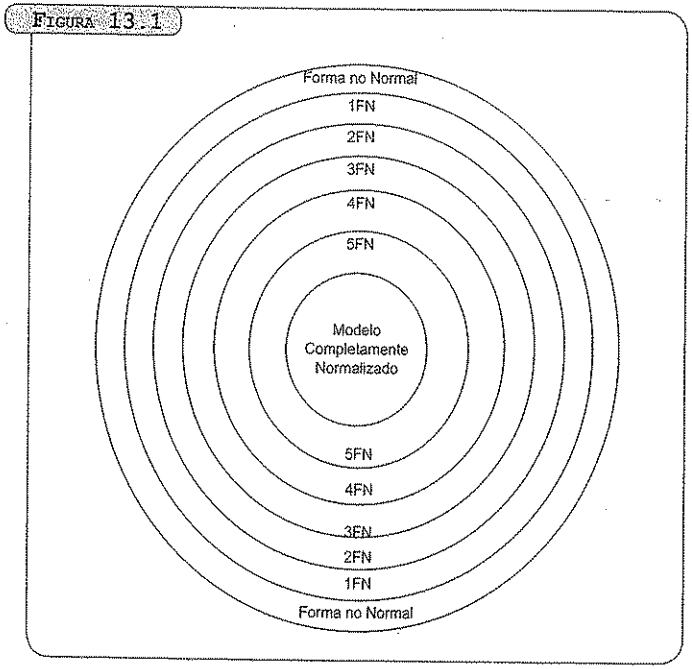
Para realizar el proceso de normalización, se parte de un esquema no normalizado, y se aplica un conjunto de reglas que permiten convertirlo en un esquema normalizado. Estas reglas son acumulativas, y el diseñador de la BD puede decidir hasta qué punto su modelo cumple lo planteado en cada una. Es decir, el proceso de normalización es gradual y no estrictamente obligatorio; de acuerdo con cada problema de la vida real, se pueden aplicar las reglas definidas hasta un determinado nivel.

Los niveles de normalización propuestos inicialmente fueron tres y se denominaron primera forma normal, segunda forma normal y tercera forma normal. La forma normal siguiente fue la planteada por Boyce-Codd. Existen dos niveles más, la cuarta forma normal y la quinta forma normal, que fueron introducidos posteriormente.

La primera forma normal no trata sobre DF, como lo hacen la segunda, la tercera y la de Boyce-Codd. La cuarta forma normal resuelve un problema generado a partir de dependencias multivaluadas, las que serán presentadas en el apartado correspondiente. La quinta forma normal, conocida por algunos autores como forma normal de dominio-clave, representa casos muy específicos, donde se plantean situaciones muy poco comunes. Existen además otros casos de formas normales que escapan al contenido de este libro.

La Figura 13.1 presenta gráficamente los niveles de normalización.

Figura 13.1



Primera forma normal

La primera forma normal presenta una regla que ya fue definida cuando se explicó la conversión del esquema conceptual al esquema lógico. La primera forma normal plantea que todos los atributos que conforman una tabla deben ser monovalentes, es decir, la cardinalidad de los atributos debe ser cero o uno.

Un modelo está en **Primera Forma Normal (1FN)** si todos los atributos que conforman las tablas del modelo son atributos monovalentes.

Los atributos polivalentes generan inconvenientes para la definición de una tabla. Como fue establecido y ejemplificado en el Capítulo 11, la solución de llevar el modelo a 1FN consistió en quitar el atributo polivalente, creando una nueva entidad.

Este proceso se realiza sobre el esquema lógico; luego, al obtener el esquema físico respectivo, todas las tablas del modelo estarán en 1FN.

Segunda forma normal

La segunda forma normal involucra el tratamiento de las DF parciales.

Un modelo está en **Segunda Forma Normal (2FN)** si está en 1FN y, además, no existe en ninguna tabla del modelo una DF parcial.

Como se analizó anteriormente, las DF parciales generan anomalías de actualización. El proceso de normalización que permite llegar a un modelo en 2FN debe tratar las DF parciales, eliminándolas de dicho modelo.

La manera de tratar una DF parcial es quitar de la tabla el atributo o los atributos que generan esta DF, y situarlos en una nueva tabla. Esto significa que la DF planteada debe mantenerse en el modelo, pero en una ubicación donde ya no sea una DF parcial.

Dada la tabla

PEDIDOS = (idpedido, idproducto, descripciónproducto, fechapedido, cantidad)

con DF parciales:

Idproducto → descripciónproducto
Idpedido → fecha pedido

estas dependencias deben ser tratadas. La solución que se debe aplicar consiste en generar tablas, ninguna de las cuales presente DF parciales:

DETALLESPEDIDOS = (idpedido, idproducto, cantidad)
PEDIDOS = (idpedido, fechapedido)
PRODUCTOS = (idproducto, descripciónproducto)

Se debe destacar que, a pesar de la división realizada, la información disponible es la misma, es decir, no hubo pérdida de información. Cada una de las tablas definidas no posee ahora DF parciales.

Tercera forma normal

La tercera forma normal implica el tratamiento de las DF transitivas.

Un modelo está en **Tercera Forma Normal (3FN)** si está en 2FN y, además, no existe en ninguna tabla del modelo una DF transitiva.

Las DF transitivas generan anomalías de actualización. En este caso, para el proceso de normalización se trabaja de manera similar, es decir, se deben quitar las DF transitivas de las tablas que las originan, colocando esos atributos en una nueva tabla. De este modo, se convierte a cada DF transitiva en solamente DF.

Anteriormente se planteó la tabla:

VENTAINMUEBLE = (idinmueble, idpropietario, nombre_propietario, valor_inmueble, idcomprador, nombre_comprador, idvendedor, comisión_venta)

Existen tres DF transitivas:

Idpropietario → nombre_propietario
Idcomprador → nombre_comprador
Idvendedor → comisión_venta

Para eliminar estas DF transitivas, se deberán quitar de la tabla VENTAINMUEBLE aquellos atributos que las generan. Entonces, el modelo resultante sería:

VENTAINMUEBLE = (idinmueble, idpropietario, idcomprador, idvendedor)
VENDEDOR = (idvendedor, comisión_venta)
PROPIETARIO = (idpropietario, nombre_propietario)
COMPRADOR = (idcomprador, nombre_comprador)

El proceso de resolución de 2FN y 3FN es similar. Si bien las DF que lo generan son diferentes, ambas presentan anomalías de actualización.

Como se indicó anteriormente en este capítulo, el proceso de normalización no es estrictamente obligatorio.

La normalización evita redundancia de datos, pero al mismo tiempo segmenta más la información.

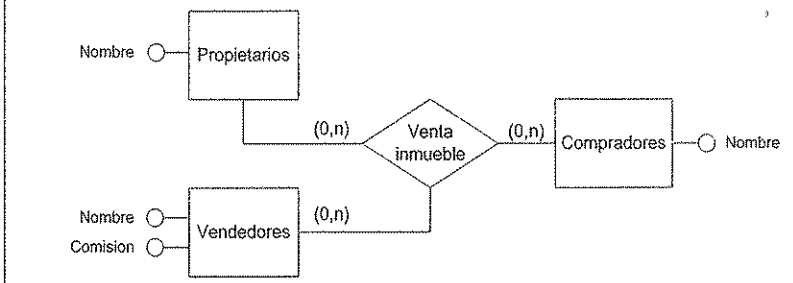
Se puede notar que, para que el modelo esté en 2FN y 3FN, cada tabla que presenta DF parciales o transitivas debe ser dividida. Esta acción significa dividir los datos en múltiples tablas, lo cual conlleva más tiempo de procesamiento para obtener posteriormente la información.

El diseñador de la BD debe, en este punto, considerar qué alternativa es mejor, si evitar redundancia de datos o segmentar la información.

Se puede comprobar que las soluciones planteadas presentan reales mejoras sobre el modelo de datos, dado que las anomalías han sido eliminadas. En general, esta línea de pensamiento es la adecuada. Es conveniente que un modelo de datos esté en 3FN.

Además, en el problema que lleva a definir la venta de inmuebles, como se plantea en el ejemplo anterior, en general, se debe mencionar a vendedores, compradores y propietarios. Más allá de un esquema conceptual que contenga jerarquías, es muy probable que el diseño tenga el formato presentado por la Figura 13.2.

Figura 13.2



La resolución del esquema lógico hacia el esquema físico plantea tablas que respetan la 3FN. El trabajo de generación de tablas para el esquema físico queda como tarea intelectual para el lector.

Ejemplos adicionales

Suponga que se dispone de la siguiente tabla:

OPERARIO_TRABAJO = (idoperario, categoría, valor_hora_categoría)

Siguiendo la definición de DF, se pueden definir:

Idoperario → categoría
Idoperario → valor_hora_categoria

Pero además, dada una categoría, es posible definir el valor de la hora trabajada en ella. Por lo tanto:

Categoría → valor_hora_categoría

Es una nueva DF sobre la tabla OPERARIO_TRABAJO, que, además, es transitiva.

La regla para llevar el modelo a 3FN consiste en quitar la DF transitiva de la tabla OPERARIO_TRABAJO. El modelo resultante quedaría:

OPERARIO_TRABAJO = (idoperario, categoria)
CATEGORIAS = (categoría, valor_hora_categoría)

Suponga que se tiene definida la tabla:

ALUMNOS_INSCRIPCIONES = (idalumno, idmateria, resultado, fecha_cursada, nombre_alumno, correlativa)

El atributo correlativa indica cuál es la materia en que el alumno se anotó. Las DF que se pueden definir son:

Idalumno, idmateria → resultado, fecha_cursada, nombre_alumno, correlativa

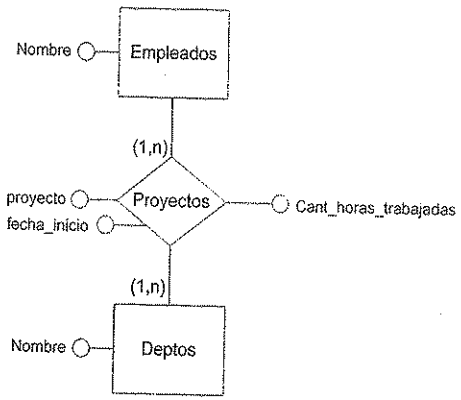
Idalumno → nombre_alumno
Idmateria → correlativa

Las dos últimas DF son parciales. Entonces, para generar un modelo en 2FN, se definen dos nuevas tablas:

ALUMNOS_INSCRIPCIONES = (idalumno, idmateria, resultado, fecha_cursada)
ALUMNOS = (idalumno, nombre_alumno)
MATERIAS = (idmateria, correlativa)

El último ejemplo que se plantea comienza con la presentación de un esquema lógico, como muestra la Figura 13.3.

Figura 13.3



La conversión al esquema físico permite obtener las siguientes tablas:

EMPLEADOS = (idempleado, nombre)
DEPARTAMENTOS = (iddepto, nombre)
EMPLEADO_DEPTO_PROYECTO = (idempleado, iddepto, nombre_proyecto, cantidad_horas_trabajadas, fecha_inicio_proyecto)

Las dos primeras tablas están en 3FN, en tanto la tercera debe ser analizada cuidadosamente. Por definición, se tienen las siguientes DF:

Idempleado, iddepto → nombre_proyecto, cantidad_horas_trabajadas, fecha_inicio_proyecto

Además, la fecha de inicio del proyecto está determinada por el nombre del proyecto, esto es:

Nombre_proyecto → fecha_inicio_proyecto

Esto es una DF transitiva, no deseada.

Suponga que un empleado podría estar trabajando en diferentes proyectos de un mismo departamento. La tabla EMPLEADO_DEPTO_PROYECTO no puede reflejar dicha situación. Al tener una CP formada por los atributos (idempleado, iddepto), no puede ocurrir que el mismo empleado esté en dos proyectos. Entonces, para poder permitir esta situación, la tabla debería tener una CP formada por (idempleado, iddepto, nombre_proyecto).

EMPLEADO_DEPTO_PROYECTO = (idempleado, iddepto, nombre_proyecto, cantidad_horas_trabajadas, fecha_inicio_proyecto)

En este caso, la DF Nombre_proyecto → fecha_inicio_proyecto sigue existiendo, pero se transforma en una DF parcial. La DF sigue siendo no deseada, y la solución consiste en quitar dicha DF. El modelo quedaría:

EMPLEADOS = (idempleado, nombre)
DEPARTAMENTOS = (iddepto, nombre)
PROYECTOS = (nombre_proyecto, fecha_inicio_proyecto)
EMPLEADO_DEPTO_PROYECTO = (idempleado, iddepto, nombre_proyecto, cantidad_horas_trabajadas)

Forma normal de Boyce-Codd

La forma normal de Boyce-Codd fue propuesta con posterioridad a la 3FN como una simplificación de esta, pero en definitiva generó una forma normal más restrictiva.

Un modelo está en **Forma Normal de Boyce-Codd (FNBC**, más conocida como BCNF, por sus siglas en inglés) si está en 3FN y, además, no existe en ninguna tabla del modelo una DF de BC.

Para generar un modelo en FNBC, se debe garantizar que los determinantes de cada DF sean siempre CC.

La regla de normalización plantea que las DF de BC deben ser quitadas de la tabla donde se generan, y llevadas a una nueva tabla. El ejemplo planteado anteriormente comenzaba con la tabla

DICTA = (nombre_alumno, nombre_materia, nombre_docente)

con dos CC (nombre_alumno, nombre_materia) y (nombre_alumno, nombre_docente), y la posibilidad de determinar, dado un profesor, la materia que dicta. O sea, un profesor solamente dicta una materia.

Se tienen, entonces, las siguientes DF:

Nombre_alumno, nombre_materia → nombre_docente
Nombre_alumno, nombre_docente → nombre_materia
Nombre_docente → nombre_materia

En los dos primeros casos, los determinantes son, además, CC. La tercera DF presenta un caso donde nombre_materia (que es parte de una CC) depende de otro atributo, nombre_docente, que también es parte de otra CC. La Tabla 13.2 presentó un ejemplo posible de la BD donde se observa repetición innecesaria de información.

La solución que plantea el proceso de normalización es generar una nueva tabla (CURSA)

DICTA = (nombre_materia, nombre_docente)
CURSA = (nombre_alumno, nombre_materia)

La solución presentada no genera DF de BC, con lo cual el modelo resultante está en FNBC. Sin embargo, se debe notar que esta transformación del modelo no representa lo mismo que el problema original. Suponga que DICTA y CURSA tienen los siguientes datos:

Tabla 13.3

DICTA	
Materia	**Docente**
IBD	Servetto
IS	Marrero
IS	Pasini
IBD	Ramón

CURSA	
Alumno	**Materia**
Castelli	IBD
Mauriello	IS
Nucilli	IBD
Burakowsky	IS
Caseres	IBD
P. Escribano	IBD
Rispoli	IS
Restelli	IBD

En la Tabla 13.3, no es posible conocer el nombre del docente con el que cursan los alumnos. La solución presentada no es adecuada porque se pierde información.

Es necesario agregar en la solución una nueva tabla:

CURSAN_CON = (nombre_alumno, nombre_docente)

la cual está en FNBC y es la solución válida para el problema planteado.

Se presenta a continuación un nuevo ejemplo para FNBC. Dada la tabla

ENTREVISTAS = (#cliente, fechaentrevista, horaentrevista, empleado, lugarentrevista)

las DF existentes son:

#cliente, fechaentrevista → horaentrevista, empleado, lugarentrevista
Empleado, fechaentrevista, horaentrevista → #cliente
Lugarentrevista, fechaentrevista, horaentrevista → empleado, #cliente

Para el ejemplo, el primer grupo de DF se plantea a partir de una CP, y los siguientes dos casos plantean a (empleado, fechaentrevista, horaentrevista) y a (lugarentrevista, fechaentrevista, horaentrevista) como CC. Un empleado en un día a una hora solamente puede concertar una entrevista y, además, en un lugar un día a una hora, solamente puede haber una entrevista. Por consiguiente, el modelo hasta el momento cumple FNBC, debido a que todo determinante es o bien CP o bien CC.

Pero siguiendo con el análisis de DF, es posible encontrar una más:

Empleado, fechaentrevista → lugarentrevista

que indica que para un día en particular, un empleado utiliza el mismo lugar para sus entrevistas. Esta consideración es, *a priori*, bastante normal; suponga que un empleado comienza sus entrevistas en la oficina 8, y por una cuestión de requerimientos es lógico que no sea válido cambiar de oficina.

Esta última DF no es parcial ni transitiva; sin embargo, el determinante no es CC ni CP, y por ende no está en NFBC.

A partir de ENTREVISTAS, la BD puede estar en un momento determinado con este contenido:

#cliente	fechaentrevista	horaentrevista	empleado	lugarentrevista
C123	12/12/2010	12:30	García	Aula 4
C332	12/12/2010	12:30	Perez	Aula 3
C341	15/12/2010	13:00	García	Aula 2
C124	12/12/2010	13:00	Perez	Aula 3

Si el empleado Perez cambia su cita del día 12/12/2010 del aula 3 al aula 20, ¿cuántos renglones hay que cambiar? Entonces, es claramente visible que la información está repetida.

La solución a esta situación puede ser:

ENTREVISTAS = (#cliente, fechaentrevista, horaentrevista, empleado)
LUGARESREUNION = (empleado, fechaentrevista, lugarentrevista)

Aquí las DF quedan:

#cliente, fechaentrevista → hora_entrevista, empleado (CP)
Empleado, fechaentrevista, horaentrevista → #cliente (CC)
Empleado, fechaentrevista → lugarentrevista (CP)

Con estas DF, el modelo está en FNBC, pero se puede notar que la dependencia Lugarentrevista, fechaentrevista, horaentrevista → empleado, #cliente no está más representada en el problema. ¿Cómo debe aplicarse el proceso de normalización a FNBC? La decisión de si es mejor detener el proceso en 3NF o llegar a FNBC depende de:

- La cantidad de redundancia que resulte de la presencia de una DF de BC.
- La posibilidad de perder una CC con la cual se podrían realizar muchos más controles sobre los datos.

Dependencia multivaluada y cuarta forma normal

Hasta el momento, se han analizado el comportamiento de las DF sobre el modelo de datos y su incidencia en la redundancia de información y en anomalías de actualización. Sin embargo, existen otro tipo de restricciones que plantean dependencias que no pueden ser consideradas como DF. Estas nuevas dependencias se denominan multivaluadas.

Una **Dependencia Multivaluada (DM)**, denotada como $X \rightarrow\rightarrow Y$, siendo X e Y conjuntos de atributos en una tabla, indica que para un valor determinado de X es posible determinar múltiples valores para el atributo Y.

Una DM no representa, en sí, una situación anómala para una BD. Cuando un atributo multidetermina a otro ($X \rightarrow\rightarrow Y$), significa que para un valor dado de X se pueden obtener varios valores de Y.

El problema de la multideterminación de datos comienza cuando en una tabla cualquiera tiene tres atributos, X, Y, Z, y se analizan las siguientes DM:

$(X, Y) \rightarrow\rightarrow Z$
$(X, Z) \rightarrow\rightarrow Y$

Pero, en realidad, tanto Y como Z solamente dependen de X en cada una de las DF.

Suponga el siguiente ejemplo:

SUCURSALES = (nombre_sucursal, propietario_sucursal, empleado_sucursal)

En algún momento, la tabla puede contener los siguientes datos:

Nombre_sucursal	Propietario_sucursal	Empleado_sucursal
La Plata	Gomez	Bertagno
La Plata	Perez	Bertagno
La Plata	Gomez	Cappa
La Plata	Perez	Cappa

Se puede notar que en la tabla anterior fue necesario indicar dos veces que el empleado Bertagno trabaja en la sucursal La Plata, y dos veces que el propietario Gomez posee la sucursal La Plata. Esto se debe a que:

(Nombre_sucursal, propietario_sucursal) →→ Empleado_sucursal
(Nombre_sucursal, empleado_sucursal) →→ Propietario_sucursal

Pero tanto el empleado como el propietario dependen de la sucursal.

Observe qué sucede si se agrega un nuevo empleado a la sucursal La Plata.

Nombre_sucursal	Propietario_sucursal	Empleado_sucursal
La Plata	Gomez	Bertagno
La Plata	Perez	Bertagno
La Plata	Gomez	Cappa
La Plata	Perez	Cappa
La Plata	Gomez	Pettorutti
La Plata	Perez	Pettorutti

El nuevo empleado, Pettorutti, debe ser ingresado dos veces debido a que la sucursal tiene dos propietarios. Esto es, con la inserción de un nuevo empleado se tiene que generar una tupla por cada propietario de la sucursal, y además, si la sucursal tuviese un nuevo propietario, debería definirse una nueva tupla por cada empleado de dicha sucursal.

Sin dudas, esta condición de DM genera una enorme repetición de información. La solución, en este caso, es tratar de generar un modelo

donde las DM que se definan no presenten esta anomalía. Esta solución se plantea como la **Cuarta Forma Normal (4FN)**. Así, es posible definir dos tablas con la siguiente estructura:

EMPLEADOS_SUCURSAL = (nombre_sucursal, empleado_sucursal)
PROPIETARIOS_SUCURSAL = (nombre_sucursal, propietario_sucursal)

En ambos casos se mantienen DM, pero bajo la forma:

Nombre_sucursal →→ empleado_sucursal
Nombre_sucursal →→ propietario_sucursal

Un modelo se encuentra en 4FN si está en FNBC y las DM que se puedan encontrar son triviales.

Una DM X →→ Y se considera trivial si Y está multideterminado por el atributo X y no por un subconjunto de X.

Dada la DM (X, Z) →→ Y, esta será trivial si Y está multideterminado únicamente por el par (X, Z). Si Y está multideterminado solamente por X o Z, la dependencia no se considera trivial.

Quinta forma normal

El proceso de normalización no finaliza con la 4FN. Las formas normales posteriores a la 4FN representan situaciones muy particulares y específicas.

En este apartado, se presenta la **Quinta Forma Normal (5FN)** a partir de un ejemplo y con la finalidad de presentar una fuerza normal que no evalúa DF ni DM.

Suponga el siguiente ejemplo: se deben administrar las exportaciones de productos de compañías, que se realizan a diversos países. Por ese motivo, se plantea la siguiente tabla:

EXPORTA = (compañía, país, producto)

En este ejemplo:

Compañía, país →→ producto
Compañía, producto →→ país

A partir de esto, se podría suponer que el modelo no se encuentra en 4FN. Esta suposición resulta natural luego de analizar el ejemplo de DM; sin embargo, en este caso, la DM no presenta una situación anómala. Tanto el producto como el país dependen del par (compañía, país) y (compañía, producto), respectivamente.

Note que un país no necesariamente va a adquirir todos los productos de una compañía. Así, en un momento específico, la BD podría tener las siguientes tuplas para la tabla EXPORTA:

Compañía	País	Producto
Tachon	Bolivia	Acero
Tachon	Paraguay	Acero
Tachon	Bolivia	Tornillos
Tachon	Bolivia	Chapas
Halcol	Chile	Caramelos
Halcol	Bolivia	Pastillas
Halcol	Chile	Pastillas

En la tabla anterior, se puede notar que la compañía Tachon exporta productos a Bolivia y que la información debió indicarse tres veces. Además, Tachon exporta acero y esa información aparece dos veces en la tabla. Sin embargo, como Paraguay no le compra a Tachon el producto chapas, este no se encuentra definido como tupla. Por ende, el modelo se encuentra en 4FN.

La DM

Compañía, país →→ producto

no puede reemplazarse por una DM

Compañía →→ producto

porque se asumiría que Tachon les vende todos sus productos a los países donde exporta.

Sin embargo, la tabla EXPORTA repite innecesariamente información. La solución que plantea 5FN y que no se basa en ningún concepto de dependencia es:

EXPORTA = (Compañía, País)
PRODUCE = (Compañía, Producto)
COMPRA = (País, Producto)

El resto de las fuerzas normales que se encuentran definidas en el área de BD están fuera del alcance de este libro.

Cuestionario del capítulo

1. ¿Qué es una dependencia funcional?
2. ¿Las DF pueden ser incorrectas o solamente pueden ser no deseadas? Justifique.
3. ¿Qué problemas pueden causar las DF no deseadas?
4. ¿Qué problemas puede generar una DF parcial?
5. ¿Qué problemas puede generar una DF transitiva?
6. ¿Qué es la normalización? ¿Para qué sirve?
7. ¿Por qué el proceso de normalización es obligatorio para 1FN y altamente recomendado para 2FN y 3FN?
8. ¿Por qué FNBC es una fuerza normal más restrictiva que 3FN?

Ejercitación

1. Con la siguiente estructura de tabla ESTUDIANTE, indique las DF existentes. Luego, para las dependencias no deseadas, exprese una solución posible para normalizar dicha tabla.

Nombre del atributo	Valor de ejemplo
#alumno	3456/6
Nombre_alumno	Alberto García
Codigo_departamento	A123
Nombre_departamento	Ciencias Básicas
Teléfono_departamento	43235566
Nombre_facultad	Informática
Nombre_decano	Ignacio Mussoppano
Oficina_decano	A335
Edificio_oficina_decano	Decanato
Teléfono_decano	54326789
Horas_clase_alumno	45
Nivel_clase	Pregrado

2. Para no perder de vista los muebles de oficina, computadoras, impresoras, etc., cierta compañía utiliza la siguiente estructura de tabla MUEBLE:

Nombre del atributo	Valor de ejemplo
Item_ID	776-23456-23
Item_descripción	HP LaserJet 3300
Lugar_de_ubicación	Presidencia
Codigo_propietario	345
Nombre_propietario	Juan Perez

Con esta información, indique las DF existentes. Para las dependencias no deseadas, exprese una solución posible para normalizar dicha tabla.

3. Examine el siguiente formulario de medicamentos para pacientes. Luego:
   a. Identifique las DF representadas por los atributos que se muestran en el formulario.
   b. Describa el proceso de normalización de los atributos para producir un conjunto de relaciones o tablas que estén en 3NF.

Hospital TE SANO ENSEGUIDA
Formulario de medicamentos para pacientes

Nombre del paciente: .....
Código de paciente: .....
Número de cama: .....

Número de departamento: .....
Nombre del departamento: .....

# fármaco	Nombre	Descripción	Dosificación	Método admin.	Unidad diaria	Fecha inicio	Fecha fin
1021	Pantomima	Antibiótico	10 mg/ml	Oral	50	10/04	12/04
1023	Fenelgan	Cura todo	20 mg	Oral	30	10/04	15/04
1123	Rinotril	Antidepresivo	40 ml	Intrav.	20	09/04	12/04

4. La tabla siguiente enumera una serie de datos de ejemplo de citas de pacientes con sus dentistas. A los pacientes se les da una cita en una fecha y hora especificadas, con un dentista que trabaja en una clínica concreta. En cada día de citas con pacientes, a cada dentista se le asigna una clínica específica.
   a. La tabla mostrada es propensa a anomalías de actualización. Indique ejemplos de anomalías de inserción, borrado y modificación.
   b. Identifique las DF representadas por los atributos mostrados y luego indique las suposiciones que haga acerca de los datos y de los atributos mostrados en la tabla.
   c. Describa el proceso de normalización de la tabla hasta llegar a 3NF.

Cód. dentista	Nombre	Cód. paciente	Nombre paciente	Cita	Empleado hace cita
D101	García	P10	Abarca	12/07 12:00	E123
D101	García	P12	Gomez	12/07 13:30	E122
D100	Perez	P10	Abarca	16/07 09:00	E123
D105	Fernandez	P13	Baansa	20/07 19:30	E112
D100	Perez	P12	Gomez	31/07 23:30	E117
D106	Hernandez	P15	Pieters	08/08 10:00	E123

5. Una agencia proporciona personal temporario para hoteles. La tabla siguiente muestra una serie de datos de ejemplo, en los que se indica el tiempo utilizado por el personal de la agencia trabajando en diferentes hoteles. El número de DNI identifica a cada empleado.

   a. Proporcione ejemplos de anomalías de inserción, borrado y modificación.

   b. Identifique las DF representadas por los atributos mostrados en la tabla.

   c. Describa el proceso de normalización de la tabla hasta llegar a 4NF.

DNI	Cód. contrato	Horas	Empleado	Cód. hotel	Nombre hotel
1234	C456	45	García	H12	Gran Hotel
3456	C456	23	Perez	H12	Gran Hotel
3345	C123	34	Cartaz	H45	Cristal
1234	C123	23	García	H45	Cristal
3323	C321	23	Hernandez	H33	Argentino
3345	C321	12	Cartaz	H33	Argentino

6. Considere la siguiente relación cuyos atributos incluyen horarios de cursos y secciones en una universidad:

   T1 = (númerocurso, númerosección, deptooferta, horascrédito, nivelcurso, profesor, semestre, año, horas_dia, númeroaula, númeroestudiantes)

   Suponga las siguientes DF:

   Númerocurso → deptooferta, horascrédito, nivelcurso
   Númerocurso, númerosección, semestre, año → horas_día, númeroaula, númeroestudiantes, profesor
   Númeroaula, horas_día, semestre, año → profesor, númerocurso, númerosección

   Intente determinar qué conjuntos de atributos constituyen claves para *T1*.

   Luego, lleve a cabo el proceso de normalización hasta lograr un modelo en FNBC.

7. Considere la siguiente tabla:

   ventaAutos = (#auto, fecha venta, #vendedor, comisión, cant_descuento)

   Suponga que un auto puede ser vendido por múltiples vendedores, y por lo tanto, la CP la definen #auto y #vendedor. Otras dependencias adicionales son:

   fecha_venta → cant_descuento

   #vendedor → comisión

   Basándose en la CP definida, ¿el modelo está normalizado? ¿Cómo lo normalizaría hasta lograr 3NF?

8. Considere la relación para libros publicados:

   Libro = (título, autor, tipo_libro, listaprecios, editor, dirección_autor)

   Si las dependencias existentes en la tabla son:

   Título → editor, tipo_libro
   Tipo_libro → listaprecios
   Autor → dirección_autor

   a. ¿En qué forma normal se encuentra la tabla?

   b. Aplique el máximo proceso de normalización posible.

9. La relación de la siguiente tabla indica los empleados que trabajan en un departamento determinado y los pacientes asignados a dicho departamento. No existe ninguna relación entre los empleados y los pacientes en cada departamento. Tanto el nombre del empleado como el nombre del paciente identifican unívocamente a un empleado o paciente, respectivamente.

Departamento	Empleado	Paciente
Pediatría	García	Gomez
Pediatría	García	Perez
Pediatría	Gonzalez	Gomez
Pediatría	Gonzalez	Perez

   a. Indique por qué la relación no está en 4FN.

   b. Indique las anomalías que puede sufrir dicha tabla.

   c. Llévela a 4FN.

IV

Procesamiento de consultas

Capítulo 14

Lenguajes de procesamiento de datos. Álgebra y cálculos

Objetivo

En 1971, Codd presentó por primera vez los conceptos iniciales del modelo de datos relacional, y de los lenguajes de procesamiento que permitían manipular los datos contenidos en el modelo generado. Si bien el modelo relacional produjo una revolución en la concepción de las BD, también lo fue la presentación de los lenguajes por sus características lógicas.

En ese año, Codd introdujo el álgebra relacional como un lenguaje procedural para la manipulación de las tablas o relaciones de la BD. Posteriormente, el mismo Codd introdujo el concepto de los cálculos como alternativa de lenguaje de consulta no procedural. Además, demostró la equivalencia entre ambos lenguajes, lo que permite establecer una correlación entre una solución de álgebra relacional y una solución de cálculos.

En este capítulo, luego de una introducción sobre los lenguajes de procesamiento de datos, se analizarán los lenguajes teóricos procedurales y no procedurales definidos en la década del setenta por Codd. En primera instancia, se presenta el álgebra relacional y, posteriormente, los cálculos: el cálculo relacional de tuplas y el cálculo relacional de dominios. En todos los casos, se desarrollarán una serie de ejemplos representativos.

Lenguajes de procesamiento de datos

Los lenguajes de procesamiento de datos permiten operar con la información contenida en una BD. Hasta este capítulo, se ha discutido la forma en que se modela una BD para resolver un problema. A partir de los

requerimientos del usuario, el diseñador de la BD construye un modelo conceptual, el cual, a través de sucesivas transformaciones, se convierte en un esquema lógico primeramente, para luego llegar a un esquema físico. Este esquema físico se impacta sobre un SGBD y de esta manera se define el modelo de datos para el problema en cuestión.

Una vez definido el modelo de datos, las operaciones posibles sobre él son básicamente cuatro: agregar, borrar, modificar o consultar información. Para poder realizar estas cuatro operaciones, se debe contar con un lenguaje de trabajo que permita indicar el tipo de operación deseada.

Los lenguajes de procesamiento de datos describen sintáctica y semánticamente todas las operaciones posibles a realizar sobre una BD. Los lenguajes se catalogan en dos grandes grupos:

1. **Procedurales:** cuando se describe una operación, se debe indicar **qué** se quiere realizar y **cómo** se debe proceder para llegar al resultado deseado.
2. **No procedurales:** cuando se describe una operación, se debe indicarse solo **qué** se necesita. El SGBD será el encargado de encontrar la mejor forma para obtener el resultado.

A priori, para realizar consultas, los lenguajes no procedurales parecen más fáciles de utilizar, pero la operatoria se complica rápidamente a medida que se intentan plantear consultas más complejas. Los lenguajes procedurales permiten al analista generar la operación e incidir en cómo resolver las consultas.

En este capítulo, se describen los tres lenguajes de manipulación de datos que representaron la base para el modelo relacional. Esos lenguajes son el álgebra relacional, que es un lenguaje procedural, y los cálculos de tuplas y de dominio, ambos no procedurales. Estos lenguajes teóricos representaron la base para la definición del lenguaje de consulta estructurado y de consulta por ejemplo (que se presentan en el Capítulo 15 como SQL y QBE, respectivamente); ambos son lenguajes cotidianos de trabajo para la mayoría de los SGBD existentes en el mercado.

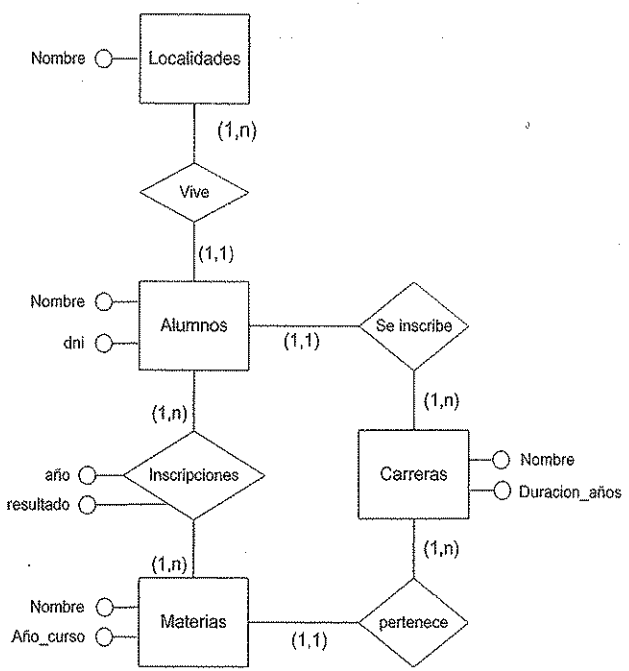
Estudios estadísticos realizados sobre BD indican que aproximadamente 80% de las operaciones sobre una BD son operaciones de consulta. Por este motivo, en primera instancia se detallan las operaciones de consulta para cada uno de los lenguajes. Luego, solamente para álgebra relacional se describen, además, las otras operaciones (altas, bajas y modificaciones).

A fin de desarrollar el capítulo de manera ordenada, se describe a continuación un ejemplo de modelo de datos (conceptual y físico).

Las tablas o relaciones resultantes serán utilizadas para describir todas las operaciones de álgebra y cálculos.

La Figura 14.1 representa el modelo conceptual incompleto (dado que no posee identificadores) de un problema donde se administran las inscripciones de alumnos a carreras y a materias.

Figura 14.1



El modelo conceptual presentado es, además, el modelo lógico. Si se observa con detalle el modelo, se podrá distinguir que no hay jerarquías ni condiciones que afecten la minimalidad, autoexplicación y expresividad de este. Por ende, se pueden asumir las siguientes tablas para el esquema físico:

ALUMNOS = (idalumno, nombre, dni, idlocalidad, idcarrera)
MATERIAS = (idmateria, nombre, año_curso, idcarrera)
CARRERA = (idcarrera, nombre, duración_años)
INSCRIPCIONES = (idinscripcion, idalumno, idmateria, año, resultado)
LOCALIDADES = (idlocalidad, nombre)

La Figura 14.2 muestra el estado de la BD en un momento particular. Esta información será considerada para mostrar el resultado de cada ejemplo planteado.

Figura 14.2

Alumnos

Idalumno	Nombre	Dni	Idlocalidad	Idcarrera
1	García	23456876	1	1
2	Perez	17876234	2	1
3	Gomez	33009876	1	2
4	Pizarro	25678965	1	2
5	Castelli	14239034	2	1
6	Pettorutti	19023487	3	2
7	Montenzanti	23434564	2	3
8	Suarez	30212305	3	1

Materias

Idmateria	Nombre	Año_curso	Idcarrera
1	Programación	1	1
2	Matemática	1	1
3	BD	2	1
4	IS	2	1
5	Conceptos	3	1
6	Concurrente	3	1
7	Fisica	1	2
8	Quimica	1	2
9	Organica	2	2
10	Inorganica	2	2
11	Contabilidad	1	3
12	Empresas	2	3
13	Comercio	2	3

Carreras

Idcarrera	Nombre	Duración_años
1	Informatica	5
2	Quimica	3
3	Contador	4

continúa >>>

Inscripciones

Idinscripcion	Idalumno	Idmateria	Año	Resultado
1	1	1	2007	6
2	1	2	2008	8
3	1	3	2008	9
4	2	3	2008	4
5	2	4	2009	6
6	3	7	2007	8
7	3	9	2009	10
8	4	8	2008	8
9	4	10	2007	9
10	5	4	2009	7
11	6	10	2008	6
12	7	11	2007	9
13	7	12	2006	8
14	8	1	2009	7
15	8	2	2008	8

Localidades

Idlocalidad	Nombre
1	General Pico
2	La Plata
3	Tres Arroyos

Álgebra relacional

La manipulación de datos de una BD implantada sobre el modelo relacional mantiene al álgebra relacional como componente principal. Básicamente, el **Álgebra Relacional (AR)** representa un conjunto de operadores que toman las tablas (relaciones) como operandos, y regresan otra tabla como resultado.

Originalmente, Codd definió ocho operadores como componentes de AR. Sin embargo, de estos ocho operadores, solo seis se consideran básicos o indispensables. Los dos restantes pueden expresarse como combinación de operaciones utilizando los operadores básicos.

A lo largo del tiempo, desde el primer planteo generado por Codd, fueron apareciendo nuevos operadores que se adaptan o resuelven de una forma más sencilla determinadas operaciones. En este apartado, primeramente se presentarán los operadores básicos. Estos operadores se utilizan para realizar consultas sobre la BD.

La representación de los operadores tiene, de acuerdo con cada autor, dos tipos de escritura posible. Mientras Date propone una notación textual similar, en cierta forma, al lenguaje de consulta SQL, otros autores como Silberchatz plantean una notación simbólica. En este libro se define el AR a partir de operandos expresados simbólicamente, por considerar a este tipo de expresiones como más sencillas para la definición inicial de consultas.

Operadores básicos

Los operadores básicos de AR son seis y se dividen en dos tipos:

1. **Unarios:** operan sobre una tabla o relación.
2. **Binarios:** operan sobre dos tablas o relaciones.

Los operadores unarios son tres: selección, proyección y renombre; y los binarios también son tres: producto cartesiano, unión y diferencia.

A continuación, se detalla y ejemplifica cada uno de ellos.

Selección

La selección es una operación que permite filtrar tuplas de una tabla. El formato genérico de una selección es:

$$\sigma_P \text{ (tabla)}$$

donde σ indica la operación de selección, tabla es una relación definida en la BD y *P* es un predicado o condición lógica que deben cumplir las tuplas de la tabla, para ser presentadas como resultado.

Ejemplo 1: Se deben presentar todas las materias de segundo año de la carrera.

$$\sigma_{\text{año\_curso} = 2} \text{ (materias)}$$

Resultado:

Idmaterias	Nombre	Año_curso	Idcarrera
3	BD	2	1
4	IS	2	1
9	Organica	2	2
10	Inorganica	2	2
12	Empresas	2	3
13	Comercio	2	3

Ejemplo 2: Se deben presentar las inscripciones de alumnos que correspondan al año 2008 y que tengan resultado superior a 7.

$$\sigma_{\text{año} = 2008 \text{ AND resultado} > 7} \text{ (inscripciones)}$$

Se puede observar que *P* resulta en un predicado que utiliza tanto operadores relacionales como lógicos; ambos son equivalentes a los utilizados en cualquier lenguaje de programación.

Resultado:

Idinscripcion	Idalumno	Idmateria	Año	Resultado
2	1	2	2008	8
3	1	3	2008	9
8	4	8	2008	8
15	8	2	2008	8

Proyección

La proyección es la operación que permite presentar en el resultado algunos atributos de una tabla. El formato genérico de una selección es:

$$\pi_L \text{ (tabla)}$$

donde π indica la operación de proyección, tabla es una relación definida en la BD y *L* es una lista, separada por comas, de atributos definidos en tabla.

Ejemplo 3: Se debe presentar el nombre de todos los alumnos.

$$\pi_{\text{nombre}} \text{ (alumnos)}$$

Resultado:

Nombre
García
Perez
Gomez
Pizarro
Castelli
Pettorutti
Montenzanti
Suarez

Ejemplo 4: Se debe presentar el nombre de los alumnos que tengan DNI superior a 20 millones.

$$\pi_{nombre}\,(\sigma_{dni > '20000000'}\,(alumnos))$$

Este último ejemplo presenta una combinación de los dos operandos definidos hasta el momento; en primera instancia, se seleccionan las tuplas que satisfacen la condición de DNI superior a 20 millones, y luego sobre esas tuplas se aplica una proyección, dado que en el resultado final solo interesa el nombre del alumno.

El orden de las operaciones es importante; si el ejemplo anterior se hubiera presentado como

$$\sigma_{dni > '20000000'}\,(\pi_{nombre}\,(alumnos))$$

la proyección quitaría el atributo dni, por lo que la selección posterior fallaría.

Resultado de $\pi_{nombre,dni}\,(\sigma_{dni > '20000000'}\,(alumnos))$

Nombre	DNI
García	23456876
Pizarro	25678965
Montenzanti	23434564

Sin el análisis de operandos binarios, no es posible presentar el operando unario renombre. Por lo tanto, este será presentado posteriormente.

Producto cartesiano

El producto cartesiano es equivalente al producto cartesiano entre conjuntos. Se aplica a dos tablas o relaciones de la BD, y vincula cada tupla de una relación con cada una de las tuplas de la otra relación. El formato genérico de un producto cartesiano es:

tabla1 × tabla2

donde × indica la operación de producto cartesiano, de la misma forma que se indica esta operación de producto en matemáticas, y tabla1 y tabla2 son dos relaciones definidas en la BD.

Ejemplo 5: Se deben presentar todos los datos de los alumnos con las localidades donde viven.

Para resolver esta consulta, se debe notar que la información requerida se encuentra definida en dos tablas. Entonces, es necesario agrupar la información de las dos tablas para poder satisfacer el pedido. Para agrupar las tuplas de ambas tablas, se utiliza un producto cartesiano alumnos × localidades.

Como se pide toda la información, podría suponerse que no es necesario realizar ninguna proyección en el resultado. En caso de querer proyectar algún atributo, bastará con indicar dicho operando junto con la lista de atributos a mostrar.

Sin embargo, la solución presentada es incorrecta. Debe notarse el comportamiento del producto cartesiano: vincula cada una de las tuplas de alumnos con cada una de las tuplas de localidades.

El alumno García vive en la localidad de General Pico. Por ende, de las tres tuplas generadas por el producto cartesiano, solo una es correcta. La siguiente tabla muestra parcialmente el resultado de alumnos × localidades:

Alumnos					Localidades	
Idalumno	Nombre	DNI	Idlocalidad	Idcarrera	Idlocalidad	Nombre
1	García	23456876	1	1	1	General Pico
1	García	23456876	1	1	2	La Plata
1	García	23456876	1	1	3	Tres Arroyos
.....	.....	.....	.....	.....	.....	.....

Por lo tanto, es necesario filtrar aquellas tuplas sin sentido obtenidas a partir del producto cartesiano. En este caso, el atributo localidad (CF) de alumnos debe coincidir con la CP de la tabla localidades. La consulta correcta es:

$$\sigma_{alumnos.idlocalidad\,=\,localidades.idlocalidad}\,(alumnos \times localidades)$$

dado que, del resultado presentado anteriormente, se quitan las tuplas donde idlocalidad no coincide.

Se puede observar además que fue necesario indicar el nombre del atributo prefijado con el nombre de la tabla. Esto se debe a que en ambas tablas aparece un atributo denominado de la misma forma. A fin de identificar unívocamente cada nombre del atributo, fue necesario utilizar dicho prefijo. En un modelo de datos no pueden existir dos tablas que se denominen igual, y por ende, el uso de prefijo es solo para la distinción de atributos.

En general, luego de un producto cartesiano, es necesario aplicar el operando de selección, para filtrar las tuplas con sentido lógico.

Ejemplo 6: Se debe presentar el nombre de cada materia junto con la carrera a la que corresponde.

$$\pi_{materias.nombre,carreras.nombre}\,(\sigma_{materias.idcarrera\,=\,carreras.idcarrera}\,(materias \times carreras))$$

Este ejemplo es similar al anterior; luego del producto cartesiano, se seleccionan las tuplas con sentido, y después se proyectan los atributos solicitados. Note el uso de los prefijos en la consulta.

Resultado:

Materias.nombre	Carreras.nombre
Programación	Informática
Matemática	Informática
IBD	Informática
IS	Informática
Conceptos	Informática
Concurrente	Informática
Fisica	Quimica
Química	Quimica
Organica	Quimica
Inorganica	Quimica
Contabilidad	Contador
Empresas	Contador
Comercio	Contador

Renombre

El renombre es la operación que permite utilizar la misma tabla dos veces en la misma consulta. El formato genérico del renombre es:

$$\rho_{nombre\_nuevo}\ (tabla)$$

donde ρ indica la operación de renombre, tabla es una relación definida en la BD y nombre_nuevo es el alias (nuevo nombre) que tiene la tabla.

Ejemplo 7: Se debe presentar el nombre de todos los alumnos que vivan en la misma localidad que el alumno Pettorutti.

Para poder realizar esta consulta, primero se debe conocer la localidad donde vive Pettorutti:

$$\sigma_{nombre = 'Pettorutti'}\ (alumnos)$$

Se supone que el nombre del alumno no se puede repetir, o sea que es CC. La tupla resultante contiene todos los datos del alumno Pettorutti (que incluyen la localidad).

Para controlar la localidad donde vive cada alumno con la localidad donde vive Pettorutti, es necesario realizar un producto cartesiano, seleccionando aquellas tuplas donde él idlocalidad coincida:

$$\sigma_{alumnos.idlocalidad = alumnos.idlocalidad}\ (\sigma_{nombre = 'Pettorutti'}\ (alumnos) \times alumnos)$$

Obviamente, la consulta anterior está mal; al realizar un producto cartesiano sobre la misma tabla, no solo se tiene repetición de atributos sino que el prefijo (nombre de la tabla) también se repite. La única alternativa para este tipo de situaciones es renombrar temporariamente una tabla para poder efectuar el producto; aquí es necesario utilizar el nuevo operador. La consulta anterior queda como:

$$\pi_{alu.nombre}\ (\sigma_{alumnos.idlocalidad = alu.idlocalidad}\ ((\sigma_{nombre = 'Pettorutti'}\ (alumnos)) \times \rho_{alu}\ (alumnos)))$$

Resultado:

Alu.nombre
Pettorutti
Suarez

Unión

El operando unión es equivalente a la unión matemática de conjuntos. Se aplica a dos tablas o relaciones de la BD, generando una nueva tabla cuyo contenido es el contenido de cada una de las tuplas de las tablas originales involucradas. El formato genérico de una unión es:

tabla1 U tabla2

donde U indica la operación de unión, de la misma forma que se indica esta operación en matemáticas, y tabla1 y tabla2 son dos relaciones definidas en la BD.

Ejemplo 8: Se deben presentar todos los nombres de los alumnos que cursen la carrera de Informática o vivan en la localidad de General Pico.

Se plantea primero la consulta que retorna todos los alumnos que cursan Informática:

$$\pi_{alumnos.nombre}\ (\sigma_{alumnos.idcarrera = carreras.idcarrera}\ ((\sigma_{nombre = 'Informática'}\ (carrera)) \times alumnos))$$

Luego, la consulta que retorna los alumnos que viven en General Pico:

$$\pi_{alumnos.nombre}\ (\sigma_{alumnos.idlocalidad = localidades.idlocalidad}\ ((\sigma_{nombre = 'General Pico'}\ (localidades)) \times alumnos))$$

Para resolver la consulta original se debe, ahora, realizar una unión entre las dos consultas planteadas:

$(\pi_{alumnos.nombre}\ (\sigma_{alumnos.idcarrera\ =\ carreras.idcarrera}\ ((\sigma_{nombre\ =\ 'Informática'}\ (carrera))\ \times\ alumnos)))$

U

$(\pi_{alumnos.nombre}(\sigma_{alumnos.idlocalidad=localidades.idlocalidad}((\sigma_{nombre='General\ Pico'}(localidades))\ \times\ alumnos)))$

Resultado:

Nombre
García
Perez
Gomez
Pizarro
Castelli
Suarez

Cuando se utiliza el operando de unión en AR, debe tenerse en cuenta que los dos conjuntos a unir deben ser **unión-compatibles**. Esto es, unir dos conjuntos que tengan la misma cantidad de atributos, y además el *i*-ésimo atributo de la primera tabla y el *i*-ésimo atributo de la segunda tabla deben tener el mismo dominio (*i*: 1..*n*). Caso contrario, el resultado obtenido no representa ninguna información útil.

Diferencia

El operando diferencia es equivalente a la diferencia matemática de conjuntos. Se aplica a dos tablas o relaciones de la BD, generando una nueva tabla cuyo contenido es el contenido de cada una de las tuplas de la primera tabla que no están en la segunda. El formato genérico de una diferencia es:

tabla1 – tabla2

donde – indica la operación de diferencia, de la misma forma que se indica esta operación en matemáticas, y tabla1 y tabla2 son dos relaciones definidas en la BD.

Ejemplo 9: Se deben presentar todos los nombres de los alumnos que cursen la carrera de Informática y no vivan en la localidad de General Pico.

Se plantea primero la consulta que retorna todos los alumnos que cursan Informática:

$\pi_{alumnos.nombre}\ (\sigma_{alumnos.idcarrera\ =\ carreras.idcarrera}\ ((\sigma_{nombre\ =\ 'Informática'}\ (carreras))\ \times\ alumnos))$

Luego, la consulta que retorna los alumnos que viven en General Pico:

$\pi_{alumnos.nombre}(\sigma_{alumnos.idlocalidad=localidades.idlocalidad}((\sigma_{nombre='General\ Pico'}(localidades))\ \times\ alumnos))$

Para resolver la consulta original se debe, ahora, realizar una diferencia entre las dos consultas planteadas:

$(\pi_{alumnos.nombre}\ (\sigma_{alumnos.idcarrera\ =\ carreras.idcarrera}\ ((\sigma_{nombre\ =\ 'Informática'}\ (carrera))\ \times\ alumnos)))$

–

$(\pi_{alumnos.nombre}(\sigma_{alumnos.idlocalidad=localidades.idlocalidad}((\sigma_{nombre='General\ Pico'}(localidades))\ \times\ alumnos)))$

Resultado:

Nombre
Perez
Castelli
Suárez

Cuando se utiliza el operando de diferencia en AR, debe tenerse en cuenta que las dos tablas deben ser unión-compatibles.

Operadores adicionales

Además de los seis operadores básicos definidos, existen otros operadores que otorgan mayor expresividad al AR, facilitando en muchos casos la construcción de las operaciones de consulta.

En este apartado, se describen cuatro operaciones adicionales: producto natural, intersección, asignación y división.

Producto natural

La definición del operando producto natural surge a partir de la necesidad de realizar una operación de selección posterior a un producto cartesiano de dos tablas, para dar como resultado las tuplas con sentido. Además y por definición, el producto cartesiano relaciona

cada tupla de la primera tabla con cada una de las tuplas de la segunda tabla. Así, si la primera tabla tiene 1.000 registros y la segunda, 100, el producto cartesiano genera una tabla intermedia de 100.000 elementos. Luego, la selección deja solamente las tuplas deseadas.

El producto natural directamente reúne las tuplas de la primera tabla que se relacionan con la segunda tabla, descartando las tuplas no relacionadas. Así, si se analiza el Ejemplo 5, el producto natural solamente reúne las tuplas de la tabla alumnos con aquellas tuplas de la tabla localidades en las cuales hay coincidencia en la idlocalidad. Esto conlleva una gran ventaja en cuanto a *performance*, dado que no se generan tuplas que luego se descartan, sino que solo se generan las tuplas necesarias.

El operador que denota producto natural es |×|, y la operación que resuelve al Ejemplo 5 queda indicada como

Alumnos |×| localidades

Se debe notar que entre las tablas sobre las que se realiza el producto natural debe existir un atributo común, o sea, una de las tablas debe tener definida una CF de la otra. En su defecto, de no existir un atributo común, el producto natural y el producto cartesiano actúan de la misma forma.

Intersección

El operando de intersección es equivalente a la intersección matemática de conjuntos. Se aplica a dos tablas o relaciones de la BD, generando una nueva tabla con las tuplas comunes a ambas tablas. El formato genérico de una intersección es:

tabla1 ∩ tabla2

donde ∩ indica la operación de intersección, de la misma forma que se indica esta operación en matemáticas, y tabla1 y tabla2 son dos relaciones definidas en la BD.

Ejemplo 10: Se deben presentar todos los nombres de los alumnos que cursen la carrera de Informática y vivan en la localidad de General Pico.

Se plantea primero la consulta que retorna todos los alumnos que cursan Informática:

$\pi_{alumnos.nombre}$ ($\sigma_{alumnos.idcarrera = carreras.idcarrera}$ (($\sigma_{nombre = 'Informática'}$ (carrera)) × alumnos))

Luego, la consulta que retorna los alumnos que viven en General Pico:

$\pi_{alumnos.nombre}$ ($\sigma_{alumnos.idlocalidad = localidades.idlocalidad}$ (($\sigma_{nombre = 'General Pico'}$ (localidades)) × alumnos))

Para resolver la consulta original se debe, ahora, realizar la intersección entre las dos consultas planteadas:

($\pi_{alumnos.nombre}$ ($\sigma_{alumnos.idcarrera = carreras.idcarrera}$ (($\sigma_{nombre = 'Informática'}$ (carrera)) × alumnos)))

∩

($\pi_{alumnos.nombre}$ ($\sigma_{alumnos.idlocalidad=localidades.idlocalidad}$ (($\sigma_{nombre='General Pico'}$ (localidades)) × alumnos)))

Resultado:

Nombre
García

Nuevamente, cuando se utiliza el operando de intersección en AR, debe tenerse en cuenta que las dos tablas deben ser unión-compatibles; en su defecto, el resultado será una tabla (relación) vacía.

El operador de intersección se define como un operador adicional debido a que es posible obtener la intersección de conjuntos a partir de su diferencia, o de combinar la diferencia con la unión:

$$A \cap B = A - (A - B)$$

$$A \cap B = A \cup B - ((A - B) \cup (B - A))$$

Asignación

El operador de asignación no aporta una funcionalidad extra al AR, sino que solamente tiene como objetivo otorgar mayor legibilidad a las consultas. Con el operador de asignación se puede generar una subconsulta y volcar su resultado en una variable temporal de tabla, la cual puede ser utilizada posteriormente. El operador de asignación es ⇐.

Ejemplo 11: Se deben asignar a una variable de tabla todas las materias de segundo año.

Materias_Segundo_Año ⇐ $\pi_{materias.nombre}$ ($\sigma_{materias.año\_curso = 2}$ (materias))

División

Dadas dos tablas *R1* y *R2*, *R1* ÷ *R2* tiene como resultado los valores de atributos de *R1*, que se relacionan con todas las tuplas de *R2*.

R1 % *R2* si, y solo si, el esquema de *R2* está incluido en el esquema de *R1*, es decir que todos los atributos de *R2* son atributos de *R1*. El esquema o "estructura" de la tabla resultante es el esquema de *R1* – el esquema de *R2*.

El siguiente ejemplo presenta genéricamente dos posibles situaciones.

Ejemplo 12:

R1

X	Y
X1	Y1
X1	Y2
X2	Y1
X3	Y2

R2

Y
Y1
Y2

R1 ÷ *R2*

X
X1

R1

X	Y	Z
X1	Y1	Z1
X1	Y1	Z2
X2	Y1	Z1
X2	Y1	Z3
X2	Y2	Z1
X2	Y2	Z2

R2

Z
Z1
Z2

R1 ÷ *R2*

X	Y
X1	Y1
X2	Y2

Ejemplo 13: Dadas las tablas

Hacen_Cursos = (alumno, nombre_curso)
Cursos = (nombre_curso)

se deben obtener los alumnos que realizaron todos los cursos que existen.

Hacen_Cursos ÷ Cursos

Actualizaciones utilizando AR

El lenguaje original de AR permite realizar, además, las operaciones básicas de actualización: alta, baja y modificación de datos contenidos en la BD.

Altas

La operación de alta permite agregar una nueva tupla a una tabla existente. Para ello, se debe realizar una operación de unión entre la tabla y una tupla escrita de manera literal.

Alumnos ⇐ Alumnos U {"Delia", "30777987", 2, 1}

Bajas

La operación de baja permite quitar una tupla de una tabla. Para ello, se debe realizar una operación de diferencia entre la tabla y una tupla escrita de manera literal.

Ejemplo 14: Se debe quitar de la tabla alumnos al alumno López.

$$\text{Alumnos} \Leftarrow \text{Alumnos} - \sigma_{\text{nombre = 'Lopez'}}(\text{alumnos})$$

Modificaciones

A diferencia de las dos operaciones de actualización anteriores, que se resolvían a partir de la utilización de operadores básicos, la operación de modificación necesita representar el cambio con un operador, δ. El formato genérico de la operación es:

$$\delta_{\text{atributo = valor\_nuevo}}(\text{tabla})$$

donde δ es el operador que indica modificación, tabla es la tabla que se modifica, atributo es el atributo de la tabla a modificar y valor_nuevo se asigna como nuevo contenido del atributo.

Ejemplo 15: Se debe modificar el número de DNI de Pettorutti por 34567231.

$$\delta_{\text{dni = "34567231"}}(\sigma_{\text{nombre = 'Pettorutti'}}(\text{alumnos}))$$

Nótese que, previo a realizar la modificación, se debe seleccionar la tupla sobre la que se desea hacer el cambio. Si no se hiciera de esa forma, todos los DNI de todos los alumnos de la tabla se modificarían.

Ejemplos adicionales

En los ejemplos de este apartado, se plantean y resuelven varios ejercicios en AR.

Ejemplo 16: Se debe presentar el nombre de los alumnos que tengan aprobada alguna materia de segundo año de cualquier carrera.

La siguiente subconsulta retorna todas las tuplas de materias, correspondientes a alumnos aprobados.

$$\text{Materias\_aprobadas} \Leftarrow (\sigma_{\text{resultado} > 3} (\text{inscripciones}))$$

De las materias aprobadas, se constata cuáles son de segundo año, y el producto natural se realiza por el atributo idmateria:

$$\text{Materias\_aprobadas\_segundo} \Leftarrow ((\sigma_{\text{año\_curso} = 2} (\text{materias}) \mid\times\mid \text{Materias\_aprobadas}))$$

Luego, se realiza el producto natural con alumnos (el atributo común es idAlumno) y, finalmente, se proyecta el nombre:

$$\pi_{\text{alumnos.nombre}} (\text{Materias\_aprobadas\_segundo} \mid\times\mid \text{alumnos})$$

Ejemplo 17: Se deben obtener los nombres de los alumnos que cursen la misma carrera que el alumno Suárez.

Se obtiene la carrera que cursa el alumno Suárez.

$$\text{Carrera\_Suarez} \Leftarrow \pi_{\text{idcarrera}} (\sigma_{\text{nombre} = \text{"Suarez"}} (\text{alumnos})$$

Se compara la carrera del alumno Suárez con las carreras de los otros alumnos, obteniendo los nombres de los alumnos que estudian la misma carrera que Suárez.

$$\pi_{\text{nombre}} (\sigma_{\text{Carrera\_Suarez.idcarrera} = \text{alumnos.idcarrera}} (\text{Carrera\_Suarez} \times \text{alumnos}))$$

Definición formal

El AR puede ser definido formalmente de la siguiente manera. Una expresión básica en AR consta de:

- Una relación o tabla de una BD.
- Una relación o tabla constante (ejemplo de la inserción, un conjunto de tuplas expresada literalmente cada una de ellas).

Una expresión (consulta) general E en AR se construye a partir de subexpresiones (subconsultas) $E1, E2, \ldots, En$.

Las expresiones posibles son:

- $\sigma_p (E1)$ P Predicado con atributos en $E1$.
- $\pi_s (E1)$ S Lista de atributos de $E1$.
- $\rho_z (E1)$ Z Nuevo nombre de $E1$.
- $E1 \times E2$.
- $E1 \cup E2$.
- $E1 - E2$.

Cálculo relacional de tuplas

El **Cálculo Relacional de Tuplas (CRT)** surge como un lenguaje de consulta de BD no procedural. Esto significa que en una consulta CRT se especifica el resultado al que se desea arribar sin definir el proceso específico para obtenerlo. Esta es la principal diferencia entre CRT y AR.

Para lograr el objetivo, las consultas tienen una sintaxis muy diferente de lo expresado anteriormente con AR. La estructura básica de CRT es la definición de una tupla y las propiedades que esta debe cumplir para ser presentada en el resultado final de la consulta. Así, una expresión en CRT tiene el siguiente formato:

$$\{t \mid P(t)\}$$

que indica que se van a presentar aquellas tuplas t tal que para t el predicado P sea verdadero.

Siguiendo con las tablas presentadas al principio del capítulo, se plantea el siguiente caso.

Ejemplo 18: Se deben presentar todas las tuplas de la relación alumnos.

Este ejercicio resuelto en AR solamente se resuelve indicando el nombre de la tabla. Nótese que no se desea realizar operación alguna con las tuplas de Alumnos.

En CRT, la expresión que resuelve esta consulta tiene el siguiente formato:

$$\{t \mid t \in \text{Alumnos}\}$$

que indica que se presentan todas las tuplas que pertenecen a la tabla (relación) alumnos.

Ejemplo 19: Se deben presentar todas las materias de primer año de cualquier carrera.

Queda como ejercicio intelectual para el lector resolver la consulta en AR. En CRT, quedará:

$$\{t \mid ((t \in \text{Materias}) \wedge t[\text{año\_curso}] = 1)\}$$

A diferencia del ejemplo anterior, se agrega ahora una condición que debe cumplir la variable de tupla para ser presentada en el resultado; en este caso, el atributo año_curso debe ser 1.

Hasta el momento, se presentó cómo realizar una consulta sin selección y otra con selección. Siguiendo la línea de operadores definidos en AR, continúa el operador de proyección.

Ejemplo 20: Se debe presentar el nombre de todas las carreras disponibles.

Esta consulta es sencilla de resolver en AR, aunque, sin embargo, en CRT tiene mayor complejidad.

Antes de resolver el ejemplo, se deben explicar los conceptos de variable de tupla libre y variable de tupla límite. Cuando una variable de tupla está ligada a una tabla, se dice que la variable de tupla es límite, es decir, limitada a los atributos de la tabla. Para limitar una variable de tupla, basta con indicar que la variable de tupla pertenece a una tabla:

$$t \in \text{Alumnos}$$

t está limitada a Alumnos y, por lo tanto, toma todos los atributos de esta. Para que una variable de tupla no esté limitada, la variable de tupla debe ser libre. Una variable se considera libre cuando no se liga a una tabla y, en su lugar, recibe los atributos que se desea presentar en el resultado.

Retomando el ejemplo, se debe utilizar una variable de tupla libre, y la expresión queda:

$$\{t \,/\, ((\exists\, s \,/\, s \in \text{Carreras}) \wedge (t\,[\text{nombre}] = s\,[\text{nombre}]))\}$$

En este caso, la variable de tupla *t* es libre. Es necesario definir una nueva variable de tupla, *s*, que esté limitada a la tabla carreras. La segunda parte de la operación de consulta debe considerarse el equivalente a una asignación. Es decir, la variable de tupla libre *t* recibe el valor del atributo nombre de la variable de tupla límite *s*.

Ejemplo 21: Se debe mostrar el nombre de todas las carreras que duren más de tres años.

$\{t \,/\, ((\exists\, s \,/\, s \in \text{carreras}) \wedge (s\,[\text{duración\_años}] > 3) \wedge (t\,[\text{nombre}] = s\,[\text{nombre}]))\}$

El siguiente ejemplo presenta la equivalencia a los productos (natural o cartesiano) de AR.

Ejemplo 22: Se deben presentar el nombre de la materia y el de la carrera a la que corresponde.

Nótese que esta información está contenida en dos tablas de la BD, Materias y Carreras. Además, el resultado que se desea presentar corresponde a un atributo de cada tabla. Nuevamente, será necesario utilizar una variable de tupla libre para la solución de este ejercicio.

$\{t \,/\, ((\exists\, s \,/\, s \in \text{materias}) \wedge (t\,[\text{nombrematería}] = s\,[\text{nombre}])) \wedge$

$((\exists\, u \,/\, u \in \text{carreras}) \wedge (u\,[\text{idcarrera}] = s\,[\text{idcarrera}]) \wedge (t\,[\text{nombrecarrera}] = u\,[\text{nombre}]))\}$

Son varias las consideraciones que se pueden realizar sobre el ejemplo anterior:

1. Fueron necesarias dos variables de tupla límite, una para cada tabla involucrada.
2. A la variable de tupla *t* se le asignan dos valores, obtenidos de dos tablas diferentes. Puede notarse que sobre la variable de tupla *t* fueron definidos atributos con nombre diferente del de las tablas originales. Esto no es una restricción. El nombre del atributo de una variable libre puede ser definido de acuerdo con lo que el operador de la consulta decida.
3. Aparece en la consulta (*u* [idcarrera] = *s* [idcarrera]); esta condición resuelve el producto natural entre las dos tablas. Nótese que para el CRT no existe diferencia entre el producto natural o cartesiano, debido a que se indica qué se desea obtener en el resultado y no la forma en que el SGBD debe operar.

Existen ocasiones donde se debe utilizar el cuantificador universal (∀) en lugar del existencial (∃).

Ejemplo 22: Se debe realizar una consulta que devuelva el nombre de los alumnos que hayan rendido alguna vez todas las materias de Informática.

$\{t \,/\, ((\exists\, u \,/\, u \in \text{carreras}) \wedge (u\,[\text{nombre}] = \text{“informática”})) \wedge$

$((\forall\, s \,/\, s \in \text{materias}) \wedge (s\,[\text{idcarrera}] = u\,[\text{idcarrera}])) \wedge$

$((\exists\, w \,/\, w \in \text{inscripciones}) \wedge (w\,[\text{idmateria}] = s\,[\text{idmateria}])) \wedge$

$((\exists\, x \,/\, x \in \text{alumnos}) \wedge (x\,[\text{idalumno}] = w\,[\text{idalumno}]) \wedge (t\,[\text{nombre}] = x\,[\text{nombre}]))\}$

Definición formal

Una expresión en CRT, $\{t / P(t)\}$, se define como:

- ***P***: fórmula donde aparecen variables de tupla.
- ***t***: puede ser:
  - **Límite:** $t \in$ tabla.
  - **Libre:** $\{t / \exists\ s / s \in$ tabla$\}$.

Las fórmulas se definen a partir de componentes o átomos. Una componente de fórmula es:

- $s \in r$; donde s es una variable de tupla y r es una tabla o relación.
- $s[x]\ \Theta\ u[y]$, s, u representan dos variables de tupla; x, y representan atributos definidos sobre s y u, respectivamente; Θ es un operador relacional cualquiera (>, <, =, >=, <>, etc.).
- $s[x]\ \Theta\ c$, donde s, x y Θ son equivalentes a la definición anterior, y c es una constante.

Las fórmulas se construyen a partir de la combinación de componentes, y para ello se utilizan operadores lógicos. Los operadores lógicos válidos son la conjunción (^), la disyunción (v) y el entonces ($\Rightarrow$). Se siguen estas reglas:

- Un átomo es una fórmula.
- Si $P1$ es una fórmula, ~ $P1$ también es fórmula.
- $P1$, $P2$ son fórmulas, entonces también las siguientes expresiones son fórmulas válidas:
  - $P1$ v $P2$.
  - $P1$ ^ $P2$.
  - $P1 \Rightarrow P2$.
- $P1(s)$ es una fórmula que contiene variables de tupla libre s. Así, $\forall\ s \in r(P1(s))$ es una fórmula y $\exists\ s \in r(P1(s))$ también es una fórmula.

Seguridad en las expresiones

Las expresiones en CRT deben cumplir un requisito indispensable, que se denomina seguridad en las expresiones. Para que una expresión sea válida, debe ser segura.

Una expresión del tipo $\{t / \sim (t \in$ alumnos$)\}$ se considera una expresión no segura. Se está pidiendo que retorne todas las tuplas de alumnos que podrían estar definidas en la tabla pero no lo están; *a priori*, un número infinito de tuplas.

Cálculo relacional de dominios

El **Cálculo Relacional de Dominios (CRD)** representa el segundo lenguaje de consulta de BD no procedural presentado en este libro. Nuevamente, una consulta en CRD especifica el resultado al que se desea arribar sin definir el proceso específico para obtenerlo. En líneas generales, opera de manera similar al CRT.

La sintaxis definida en CRD es equivalente al CRT. Se trabaja sobre cada dominio que se desea presentar en el resultado, a diferencia del CRT, que trabaja sobre la tupla. Una expresión en CRD tiene el siguiente formato:

$$\{(atr_1, \ldots, atr_n) / P(atr_1, \ldots, atr_n)\}$$

que indica que se presentarán aquellos dominios (atr_i) tal que el predicado P sea verdadero. A partir de las tablas presentadas al principio del capítulo y los ejemplos de CRT, se plantea la solución de los mismos ejercicios en CRD.

Ejemplo 23: Se deben presentar todas las tuplas de la relación alumnos.

La expresión en CRD sigue la misma línea que en CRT; como la tabla alumnos tiene cinco dominios definidos, son necesarias cinco variables de dominio; entonces:

$$\{(a, b, c, d, e) / (a, b, c, d, e) \in \text{Alumnos}\}$$

que indica que se presentan todos los dominios de alumnos. Las variables de dominio pueden tener el nombre que se desee, y la relación entre una variable de dominio y el atributo de la tabla es por posición. Así, la variable de dominio *a* corresponde a idalumno; *b*, a nombre, y así sucesivamente.

Ejemplo 24: Se deben presentar todas las materias de primer año de cualquier carrera.

$$\{(im, no, ac, ic) / (((im, no, ac, ic) \in \text{Alumnos}) \wedge ac = 1)\}$$

A diferencia del ejemplo anterior, se agrega ahora una condición que debe cumplir la variable de dominio *ac*, que se corresponde con el atributo año_curso de la tabla Materias.

Ahora, la proyección resulta más directa. Basta con poner, en la lista de variables de dominio a presentar, aquellos atributos que estarán en la respuesta. El resto se omite.

Ejemplo 25: Se debe presentar el nombre de todas las carreras disponibles.

$$\{nc \,/\, (\exists\, (ic, da) \,/\, (ic, nc, da) \in \text{Carreras})\}$$

El ejercicio resulta más simple que en CRT. Primeramente, para solicitar la pertenencia a Carreras, la lista de variables de dominio debe coincidir con la cantidad de atributos de la tabla; en este caso, son necesarios tres dominios. El primero, *nc*, que corresponde al nombre de la carrera, se presenta en el resultado; pero restan dos dominios más, que como no se presentan deben ser solicitados posteriormente. Nótese que, a diferencia del CRT, no es necesario asignar nada al resultado, dado que ahora este es una variable de dominio.

Ejemplo 26: Se debe mostrar el nombre de todas las carreras que duren más de tres años.

$$\{nc \,/\, ((\exists\, (id, da) \,/\, (id, nc, da) \in \text{carreras}) \wedge (da > 3))\}$$

Ejemplo 27: Se deben presentar el nombre de la materia y el de la carrera a la que corresponde.

Nótese que esta información está contenida en dos tablas de la BD, Materias y Carreras.

$$\{(nm, nc) \,/\, ((\exists\, (im, ac, id) \,/\, (im, nm, ac, id) \in \text{materias}) \wedge (\exists\, da \,/\, (id, nc, da) \in \text{carreras}))\}$$

Se puede suponer que la consulta anterior es errónea o que faltan comparaciones. Sin embargo, no es así. Se desea presentar dos dominios: nombre de materia y nombre de carrera; las variables *nm* y *nc* se utilizan para ese fin. Luego, para trabajar con materias son necesarios tres dominios adicionales: el identificador, el año de curso y la CF idcarrera. La tabla carreras tiene tres dominios: el nombre de carrera es el que se presenta en el resultado (ya está definido); la CP es el mismo dominio que la CF de Materias –por lo tanto, esa variable también está definida–; por último, se define la duración en años. Al utilizar el mismo dominio para la CP de Carreras y la CF de Materias, se evita la comparación entre ambas.

Seguridad en las expresiones

Las expresiones en CRD deben cumplir los mismos requisitos de seguridad que una expresión en CRT.

Cuestionario del capítulo

1. ¿Qué es un lenguaje de consulta para una BD?
2. ¿Qué significa que un lenguaje sea procedural y qué lo diferencia de uno no procedural?
3. ¿Cuáles son las diferencias entre el AR y los cálculos?
4. ¿Por qué los operadores básicos de AR son seis?
5. ¿Por qué el producto natural o la intersección no son operadores básicos de AR?
6. ¿Cuál es la principal diferencia entre CRT y CRD?
7. ¿Cómo actúan los cuantificadores universales y existenciales en una consulta en cálculo?
8. ¿Por qué en los cálculos no tiene sentido diferenciar entre producto natural y producto cartesiano?
9. ¿Por qué los cálculos necesitan expresiones seguras?

Ejercitación

Dado el siguiente modelo físico de datos:

Autor = (idautor, nombreautor, fechanacimiento, idlocalidadnacimiento)

Libro = (idlibro, titulolibro, isbn, número de páginas, ideditorial)

Editorial = (ideditorial, nombreeditorial, direcciónCasacentral, idlocalidadcasacentral)

Autoria = (idautoria, idlibro, idautor)

Localidad = (idlocalidad, nombre)

1. Resuelva las siguientes consultas en AR, CRT y CRD:
   a. Nombre de los libros contenidos en la tabla.
   b. Nombre de los autores nacidos antes de 1900 (para ello, suponga que existe una función YEAR que retorna el año de la fecha de nacimiento).
   c. Todos los libros de la editorial Pearson.
   d. Autores de cada libro.
   e. Nombre del libro y de la editorial para aquellos libros con más de 500 páginas.
   f. Libros cuyo autor haya nacido en la misma localidad donde esté la sede central de la editorial que edita el libro.

2. Resuelva las siguientes consultas en AR:
   a. Agregue a la tabla libros el siguiente material:
      Nombre del libro: *Introducción a las Bases de Datos*
      ISBN: 779-234243-123
      Número de páginas. 780
      idEditorial: 3
   b. Quite todos los autores del libro *Ingeniería de Software*, cuyo ISBN es 779-343323-123.
   c. Modifique la fecha de nacimiento del autor Nelson Castro.

Capítulo 15

SQL. y QBE.

Objetivo

El origen de SQL está ligado a las bases de datos relacionales. Como se indicó anteriormente, en 1970 E. F. Codd propone el modelo de datos relacional y asociado a éste, lenguajes de consulta procedural (algebra relacional) y no procedural (cálculos). Estos lenguajes propuestos por Codd fueron la base para la creación de SQL (Structured Query Languaje, Lenguaje de Consultas Estructurado).

SQL termina siendo una de las principales razones del éxito de las BD relacionales. Esto se debió a que la mayoría de los fabricantes de DBMS optaron por SQL como el lenguaje de trabajo, generando de esta forma un estándar respetado en general, por la mayor parte de los productos de mercado. Así, una consulta SQL puede migrar de producto sin necesitar mayores cambios para ser ejecutada.

SQL es un lenguaje de consultas de BD, que está compuesto por dos submódulos fundamentales:

- Módulo para definición del modelo de datos, denominado DDL (Data Definition Languaje) con el cual se pueden definir las tablas, atributos, integridad referencial, índices y restricciones del modelo.
- Módulo para la operatoria normal de la BD, denominado DML (Data Manipulation Languaje) con el cual se pueden realizar las consultas, altas, bajas y modificaciones de datos sobre las tablas del modelo.

En este capítulo se presenta en detalle el DML de SQL 92 debido a que este estándar se adecua, por sus características, a las necesidades básicas que requiere un analista para trabajar contra una BD. Asimismo, se presentan algunas características básicas del DDL.

En este capítulo se define además, el lenguaje denominado QBE (query by example), el cual se encuentra disponible también junto con la mayoría de los DMBS de mercado. Este lenguaje implementa el cálculo relacional de dominios, permitiendo generar consultas sobre una BD a partir de la definición de qué necesita la consulta, sin tener que indicar la forma en que se realiza el proceso.

Introducción al SQL

Un poco de historia

En 1986 SQL es estandarizado por el ANSI (American National Standart Institute, la organización de estándares de Estados Unidos), dando lugar a la primera versión estándar de este lenguaje, el "SQL-86" o "SQL1". Al año siguiente este estándar fue adoptado por la ISO (International Standart Organization).

En 1992 se amplía la definición del estándar para cubrir mayores necesidades de los desarrolladores. Surge de esta forma el SQL-92 o el SQL2. Esta versión será la base de los temas tratados en este capítulo. Si bien hay versiones posteriores, para la finalidad del presente material los parámetros definidos en ese estándar resultan suficientes. Se podrá notar a lo largo del capítulo que no se hace referencia al SQL de ningún DBMS comercial. Este material no está ligado con ninguno de ellos, y cada una de las consultas propuestas debería ser ejecutada sin problemas en los DBMS comerciales que soporten SQL. Si esto no ocurriera así, o si el resultado no fuese el esperado, es por la definición particular de cada producto comercial.

El estándar tuvo varias modificaciones posteriores: en 1999 se creó el SQL2000, al que se le incorporaron expresiones regulares, consultas recursivas y características orientadas a objetos. En 2003, surge SQL3 que agrega características de XML y, posteriormente en 2006 ISO define ISO/IEC 9075-14:2006 con características que acercan a SQL al mundo W3C.

En las dos secciones subsiguientes se presentan el DDL (lenguaje de definición de datos) y el DML (lenguaje de manipulación de datos).

Lenguaje de Definición de datos

El modelo de datos relacional utiliza el concepto de relación, tupla y atributo, en tanto en SQL los términos corresponden a tabla, fila y columna. En este libro tabla-relación, tupla-fila y atributo-columna, se utilizan indistintamente.

Para administrar un modelo físico de datos, SQL presenta tres cláusulas básicas: CREATE, DROP y ALTER. Los mismos se corresponden con crear, borrar o modificar el esquema existente.

No es objetivo de este libro profundizar detalladamente en la definición del DDL de SQL2. En general, los productos de mercado ofrecen una interfaz amigable que permite generar BD, sus tablas, índices, etc. Además, existen herramientas que asisten al modelado de BD que, una vez terminado dicho modelo, permiten generar de manera automática el código SQL que genera la BD sobre el DBMS.

Crear o borrar una BD

Para generar una BD la sentencia SQL es:

```
CREATE DATABASE nombre_BD;
```

Esta sentencia genera una BD cuyo nombre se indica.

Para eliminar una BD completa la sentencia es:

```
DROP DATABASE  nombre_BD;
```

La cual borra la BD: esto incluye las tablas y los datos contenidos en ellas.

Operar contra Tablas de BD

Para generar una tabla en una BD, la instrucción SQL es CREATE TABLE. El siguiente ejemplo presenta la creación de una tabla denominada empresas.

```
CREATE TABLE empresas (
  idempresa INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  empresa VARCHAR(100) NOT NULL,
  abreviatura VARCHAR(10) NULL,
  cuit VARCHAR(13) NULL,
  direccion VARCHAR(100) NULL,
  observaciones TEXT NULL,
  PRIMARY KEY(idempresa),
  UNIQUE INDEX empresas_index19108(empresa));
```

Para definir una tabla es necesario indicar cada uno de los atributos que la componen. En este caso algunos de los atributos son:

- Idempresa: con dominio Integer sin signo, el atributo no puede ser nulo y el dato es autoincremental (tal como se definiera oportunamente en el capítulo de modelado físico).
- Empresa: con dominio String y con longitud máxima 100, no puede ser nulo.
- Obsevaciones: dominio texto (a diferencia del String que tiene longitud máxima, el tipo de dato texto no está limitado)

La sintaxis utilizada, si bien es ANSI, tiene similitud con la utilizada por el DBMS MySQL. Los valores posibles para los dominios de los atributos están ligados directamente con los productos comerciales. Asimismo ocurre con la definición de los atributos autoincrementales.

Por último, en el ejemplo anterior, se encuentran definidos dos índices para la tabla Empresas. Estos índices se definen asociados a la clave primaria y clave candidata respectivamente.

El siguiente ejemplo presenta la creación de otra tabla, donde se agrega la definición de una clave foránea.

```
CREATE TABLE pacientesempresas (
  idpacienteempresa INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  idpaciente_os INTEGER UNSIGNED NOT NULL,
  idempresa INTEGER UNSIGNED NOT NULL,
  fecha_desde DATE NOT NULL,
  fecha_hasta DATE NULL,
  PRIMARY KEY(idpacienteempresa),
  INDEX empleadosempresas_FKIndex1(idempresa),
  INDEX pacientesempresas_FKIndex2(idpaciente_os),
  FOREIGN KEY(idempresa)
    REFERENCES empresas(idempresa)
      ON DELETE RESTRICT
      ON UPDATE NO ACTION,
  FOREIGN KEY(idpaciente_os)
    REFERENCES pacientes_os(idpaciente_os)
      ON DELETE RESTRICT
      ON UPDATE NO ACTION);
```

Se agrega aquí la definición de dos índices secundarios: empleadosempresas y pacientesempresas. Luego se definen las clave foráneas, con las referencias a las tablas de donde se obtiene la CP. Se indica, además, el tipo de restricción de integridad definido: el borrado está restringido y por la modificación no se toma acción.

Para eliminar una tabla del modelo la sentencia es:

```
DROP TABLE  nombre_tabla;
```

Para modificar una tabla del modelo la sentencia es:

```
ALTER TABLE  nombre_tabla ( … ) ;
```

Esta sentencia debe indicar, además, que tipo de modificación se desea realizar sobre la tabla. Así entre paréntesis se pueden agregar, modificar o borrar atributos, índices o restricciones de integridad.

A modo de ejemplo, la siguiente sentencia SQL agrega el atributo razón social a la tabla empresa generada anteriormente, se quita el atributo cuit y se modifica el dominio del atributo dirección por un string de longitud 50.

```
ALTER TABLE empresas (
  Add column razon_social VARCHAR(100) NOT NULL,
  Drop column cuit,
  Alter column direccion VARCHAR(50) NULL);
```

Lenguaje de Manipulación de datos

En esta sección se presenta la sintaxis y semántica asociada de SQL2 (SQL92) para recuperar información de una BD. Además, se presentarán las sentencias básicas para agregar, borrar o modificar información contenida en la BD.

Para desarrollar esta sección se utilizarán varios ejemplos, basados en el modelo de datos presentado en el capítulo anterior.

Estructura Básica

La estructura básica de una consulta SQL tiene el siguiente formato:

```
SELECT  lista_de_atributos
FROM   lista_de_tablas
WHERE  predicado
```

Donde la lista_de_atributos indica los nombres de los atributos que serán presentados en el resultado. Estos atributos deben estar contenidos en las tablas indicadas en la consulta. Lista_de_tablas indica las tablas de la BD necesarias para resolver la consulta; sobre las tablas indicadas en la lista se realiza un producto cartesiano. Por último, predicado indica que condición deben cumplir las tuplas de las tablas para estar en el resultado final de la consulta.

La analogía entre SQL y AR es importante. AR representa la base teórica de SQL, por lo tanto las consultas expresadas en ambos lenguajes es similar en aspectos semánticos. Al igual que AR, una consulta SQL siempre retorna como resultado una tabla temporal. Una consulta SQL del tipo

```
SELECT  atr1, atr2, atr3
FROM    tabla1, tabla2
WHERE   atr4 = 'valor'
```

Equivale a la siguiente consulta en AR:

$\pi_{atr1, atr2, atr3} (\sigma_{atr4 = 'valor'} (tabla1 \times tabla2))$

La sentencia SELECT equivale a la operación de proyección del AR (con la única diferencia que no se eliminan automáticamente las tuplas repetidas), la sentencia FROM equivale al producto cartesiano y el WHERE a la selección.

Ejemplo 1: presentar todos los alumnos que figuran en la tabla Alumnos

```
SELECT  nombre
FROM    alumnos
```

Se puede notar que la sentencia WHERE no es obligatoria. En este ejemplo para presentar el nombre de todos los alumnos no es necesario aplicar ningún filtro de tuplas. Solo se requiere proyectar al atributo deseado. La tabla temporal obtenida como respuesta a este ejercicio, de acuerdo con los datos presentados en la figura 14.2 es:

Nombre
García
Perez
Gomez
Pizarro
Castelli
Pettorutti
Montenzanti
Suarez

Ejemplo 2: presentar todos los datos de los alumnos que figuran en la tabla alumnos.

```
SELECT  idalumno, nombre, dni, idlocalidad, idcarrera
FROM    alumnos
```


La lista de atributos debe definir en este ejemplo a todos los atributos de la tabla alumnos. Esta tarea puede tornarse tediosa cuando la lista de atributos es extensa.

SQL presenta un operador que permite reemplazar la escritura literal de todos los atributos de la tabla. Este operador es el * , su aparición indica que todos los atributos de las tablas definidas en el FROM, serán presentados en el resultado de la consulta.

El ejemplo anterior puede definirse como:

```
SELECT  *
FROM    alumnos
```

Ambas soluciones del ejemplo 2 retornan la tabla Alumnos tal y como se presenta en la figura 14.2

Ejemplo 3: presentar todos los alumnos que cursen la carrera cuyo código es 2.

```
SELECT  nombre
FROM    alumnos
WHERE   idcarrera = 2
```

En este caso es necesario utilizar la condición para filtrar en el resultado las tuplas deseadas.

Nombre
Gomez
Pizarro
Pettorutti

Ejemplo 4: presentar todos los alumnos que cursen la carrera cuyo código es 2 y vivan en la localidad con código 1.

```
SELECT  nombre
FROM    alumnos
WHERE   idcarrera = 2 AND idlocalidad = 1
```

En este caso es necesario utilizar la condición para filtrar en el resultado las tuplas deseadas.

Nombre
Gomez
Pizarro

Ejemplo 5: presentar todas las materias y el año en que se cursan.

```
SELECT  nombre, año_curso
FROM    materias
```

Nombre	Año_curso
Programación	1
Matemática	1
IBD	2
IS	2
Conceptos	3
Concurrente	3
Física	1
Química	1
Organica	2
Inorganica	2
Contabilidad	1
Empresas	2
Comercio	2

Ejemplo 6: mostrar todos los códigos de materias que estén en la tabla Inscripciones que tengan al menos a un alumno aprobado.

```
SELECT  idmateria
FROM    inscripciones
WHERE   resultado > 3
```

La Tabla temporal obtenida como resultado de la consulta es:

Idmateria
1
2
3
3
4
7
9
8
10
4
10
11
12
1
2

Se puede notar que el código 2 aparece varias veces en el resultado. Esto significa que más de un alumno ha aprobado la materia cuyo código es 2. En determinadas circunstancias puede no ser necesario que el mismo resultado aparezca varias veces. Para eliminar tuplas repetidas se debe utilizar la cláusula DISTINCT.

```
SELECT  DISTINCT idmateria
FROM    inscripciones
WHERE   resultado > 3
```

SQL define, además, el operador ALL. Este operador muestra todas las tuplas, aún las repetidas. En caso de no poner explícitamente el operador DISTINCT, SQL asume el operador ALL, por este motivo en general nunca se explicita.

Ejemplo 7: presentar todas las carreras que tiene entre 4 y 6 años de duración.

```
SELECT  nombre
FROM    carreras
WHERE   duracion_años >= 4 AND duración_años <= 6
```

Además, es posible realizar la misma consulta utilizando el operador BETWEEN. La consulta anterior puede expresarse como:

```
SELECT  nombre
FROM    carreras
WHERE   duracion_años BETWEEN 4 and 6
```

Los atributos utilizados en el SELECT de una consulta SQL pueden tener asociados operaciones válidas para sus dominios.

Así, por ejemplo sería válida una consulta que devuelva el salario de un empleado menos un 10% de retención.

```
SELECT  nombre_empleado, salario * 0.9
FROM    empleados
WHERE   ocupacion = "Gerente"
```

Esta consulta devuelve el nombre de cada gerente y el valor correspondiente al 90 % de su salario.

Sobre los atributos definidos en un SELECT se pueden realizar cualquiera de las operaciones básicas definidas para su domino. Cada DBMS en particular define su mapa de operaciones.

Hasta el momento, se presentaron consultas que solamente involucraron una tabla del modelo de datos. Para realizar un producto cartesiano, basta con poner en la cláusula FROM dos o más tablas.

Ejemplo 8: presentar el nombre de todos los alumnos y la carrera que cursa:

```
SELECT  alumnos.nombre, carreras.nombre
FROM    alumnos, carreras
WHERE   alumnos.idcarrera = carreras.idcarrera
```

El resultado de la consulta se presenta en la siguiente tabla de resultados:

alumnos.nombre	carreras.nombre
García	Informática
Perez	Informática
Gomez	Química
Pizarro	Química
Castelli	Informática
Pettorutti	Química
Montezanti	Contador
Suarez	Informática

La cláusula WHERE es necesaria por el mismo motivo que el producto cartesiano necesitó en AR de una selección para filtrar las tuplas con sentido. Asimismo, en la consulta se puede observar que fue necesario definir un prefijo para los atributos nombre e idcarrera, que se encontraban repetidos sintácticamente en ambas tablas.

Ejemplo 9: presentar el nombre de todos los alumnos, la carrera que cursa y de que localidad proviene:

```
SELECT a.nombre, c.nombre, l.nombre
FROM alumnos a, carreras c, localidades  l
WHERE a.idcarrera=c.idcarrera AND a.idlocalidad=l.idlocalidad
```

Esta consulta es similar a la anterior, ahora a partir de tres tablas. Se puede notar que seguido de los nombres de las tablas en la cláusula FROM, se agregó una letra. Esta letra actúa como un alias de cada tabla. Si bien en esta consulta no es necesario renombrar las tablas, es frecuente cambiar el nombre de la relación por otro más corto, con el propósito de escribir más rápidamente la consulta.

De la misma forma que se puede renombrar una tabla en la cláusula FROM, es posible renombrar un atributo en al cláusula SELECT. La siguiente consulta resuelve nuevamente el ejemplo 9 renombrando los atributos.

```
SELECT  a.nombre AS nombrealumno, c.nombre AS nombrecarrera,
        l.nombre AS nombrelocalidad
FROM    alumnos a, carreras c, localidades  l
WHERE a.idcarrera=c.idcarrera AND a.idlocalidad=l.idlocalidad
```


Se utiliza el operador AS para renombrar un atributo. La consulta genera la siguiente tabla:

nombrealumno	nombrecarrera	nombrelocalidad
García	Informática	General Pico
Perez	Informática	La Plata
Gomez	Química	General Pico
Pizarro	Química	General Pico
Castelli	Informática	La Plata
Pettorutti	Química	Tres Arroyos
Montezanti	Contador	La Plata
Suarez	Informática	Tres Arroyos

A partir de la cláusula básica en SQL, se han presentado las operaciones del AR correspondientes a selección, proyección, producto cartesiano y renombrado. Las restantes operaciones básicas (unión y diferencia) también están presente en SQL.

Ejemplo 10: presentar todas las materias aprobadas del alumno Pizarro, o rendidas durante el 2008.

```
(SELECT  m.nombre
  FROM   alumnos a, materias m, inscripciones i
  WHERE  a.nombre = "Pizarro" AND a.idmateria = i.idmateria AND
         i.idalumno = a.idalumno AND i.resultado > 3)
UNION
(SELECT  m.nombre
  FROM   alumnos a, materias m, inscripciones i
  WHERE  a.nombre = "Pizarro" AND a.idmateria = i.idmateria AND
         i.idalumno = a.idalumno AND i.año = 2008)
```

La primera consulta devuelve los nombres de las materias rendidas por Pizarro que resultaron aprobadas, mientras que la segunda consulta devuelve las materias rendidas por Pizarro durante 2008. La cláusula UNION reúne ambos resultados.

En SQL92 la UNION se resuelve de manera tal que las tuplas duplicadas en ambas subconsultas solo quedan una vez en el resultado final. Para obtener las tuplas duplicadas se puede utilizar la cláusula UNION ALL.

La cláusula definida por SQL92 para la diferencia de conjuntos es EXCEPT, en tanto que para la operación de intersección se utiliza la cláusula INTERSECT.

Operadores de strings

Una de las características interesantes y que dotan a SQL de gran potencia en la generación de consultas que relacionan Springs, resulta del uso del operador de comparación LIKE.

Cuando una cadena se compara por igualdad (=) el resultado será verdadero si ambas cadenas son iguales, falso en caso contrario.

Suponga que se desea saber el DNI y la localidad donde nació un alumno, cuando lo único que se recuerda del alumno es que su nombre comienza con P. Si se definiera al condición (nombre = "P") no retornaría ninguna tupla, dado que no hay nombre alguno de alumno que sea solamente P.

Dicha consulta puede resolverse de la siguiente forma:

```
SELECT  nombre,   idlocalidad, dni
FROM    alumnos
WHERE   nombre LIKE "P%"
```

El resultado obtenido se muestra en la siguiente tabla.

Nombre	DNI	Idlocalidad
Perez	17876234	2
Pizarro	25678965	1
Pettorutti	19023487	3

El operador LIKE se conjuga con el carácter reservado %. Se compara el atributo nombre de la tabla alumnos contra el string que empieza con P y continúa con cualquier substring (ese es el comportamiento del carácter %, el cual considera válida a partir de la posición donde aparece, cualquier cadena de caracteres, inclusive la cadena vacía). Entonces, cualquier tupla cuyo atributo nombre comience con P será presentado en el resultado final.

Existe además otro carácter reservado, '_', el guión bajo sustituye solo el carácter del lugar donde aparece.

Ejemplo 11: presentar todas las materias que contengan el substring 'ga' dentro de su nombre.

```
SELECT  nombre
FROM    materias
WHERE   nombre LIKE "%ga%"
```

Se presentarán en el resultado:

Nombre
Organica
Inorganica

El carácter reservado % indica que la cadena puede comenzar y terminar con cualquier string, y además en algún lugar de la misma debe aparecer el substring 'GA'

Ejemplo 12: presentar el nombre de los alumnos que tengan DNI que comience con 23 millones.

```
SELECT  nombre
FROM    alumnos
WHERE   dni LIKE "23_ _ _ _ _ _"
```

Se puede notar que el atributo DNI se compara contra una cadena que comienza con 23. Luego se solicitan que aparezcan exactamente 6 caracteres más, sin importar cuales sean.

En algunas situaciones es necesario presentar la tabla temporal que resuelve la consulta ordenada por algún criterio. La cláusula **ORDER BY** permite ordenar las tuplas resultantes por el atributo indicado.

Ejemplo 13: presentar los nombres de todas las materias y el año de curso, para la carrera informática, ordenados por año de curso.

```
SELECT  nombre, año_curso
FROM    materias, carreras
WHERE carreras.nombre = 'Informática' AND
      materias.idcarrera = carreras.idcarrera
ORDER BY año_curso
```

El resultado obtenido es el siguiente:

Nombre	Año_curso
Programación	1
Matemática	1
IBD	2
IS	2
Conceptos	3
Concurrente	3

Por defecto la cláusula ORDER BY ordena las tuplas de menor a mayor, es decir en orden creciente. Si se desea obtener el resultado ordenado de mayor a menor, hay que utilizar el operador DESC.

Ejemplo 14: presentar los nombres de todas las materias y el año de curso, para la carrera informática, ordenados por año de curso de mayor al menor.

```
SELECT  nombre, año_curso
FROM    materias, carreras
WHERE carreras.nombre='Informática' AND
      materias.idcarrera = carreras.idcarrera
ORDER BY año_curso DESC
```

El resultado obtenido es el siguiente:

Nombre	Año_curso
Conceptos	3
Concurrente	3
IBD	2
IS	2
Programación	1
Matemática	1

Dentro de la cláusula ORDER BY es posible indicar más de un criterio de ordenación. El segundo criterio se aplica en caso de empate en el primero y así sucesivamente.

Ejemplo 15: presentar los nombres de todas las materias y el año de curso, para la carrera informática, ordenados por año de curso y nombre de la materia

```
SELECT  nombre, año_curso
FROM    materias, carreras
WHERE   carreras.nombre = 'Informática' AND
        materias.idcarrera = carreras.idcarrera
ORDER BY año_curso, nombre
```

El resultado obtenido es el siguiente:

Nombre	Año_curso
Matemática	1
Programación	1
IBD	2
IS	2
Conceptos	3
Concurrente	3

Consultas con funciones de agregación

Las funciones de agregación son funciones que operan sobre un conjunto de tuplas de entrada y producen un único valor de salida. SQL define cinco funciones de agregación (en orden alfabético):

- AVG: retorna como resultado el promedio del atributo indicado para todas las tuplas del conjunto.
- COUNT: retorna como resultado la cantidad de tuplas involucradas en el conjunto de entrada.
- MAX: retorna como resultado el valor más grande dentro del conjunto de tuplas para el atributo indicado
- MIN: retorna como resultado el valor más pequeño dentro del conjunto de tuplas para el atributo indicado
- SUM: retorna como resultado la suma del valor del atributo indicado para todas las tuplas del conjunto.

Ejemplo 16: Retornar el nombre del alumno que aparece primero en una lista alfabética.

```
SELECT  MIN(nombre)
FROM alumnos
```

Este ejemplo toma como entrada todas las tuplas de alumnos. Para el atributo nombre chequea el menor en orden alfabético y ese dato se presenta en el resultado.

Ejemplo 17: Retornar la máxima nota de alguna materia durante 2008.

```
SELECT  MAX( resultado )
FROM    inscripciones
WHERE año=2008
```

Aquí de la tabla inscripciones se filtran las tuplas correspondientes a 2008. Sobre ese subconjunto se obtiene la mayor nota obtenida y se muestra como resultado. Si hubiera varias tuplas con la misma nota esto no afecta la respuesta.

Ejemplo 18: Informar cuantos alumnos aprobaron el final de programación

```
SELECT COUNT(*)
FROM materias m, inscripciones i
WHERE i.idmateria=m.idmateria AND
      m.nombre = "Programación" AND  i.resultado > 3
```

En este ejemplo es necesario primero realizar un producto cartesiano entre las tablas materias e inscripciones. En la primera se tiene definido el nombre de la materia, en la segunda los resultados. Se filtran las tuplas con sentido del producto cartesiano, aquellas que corresponden a Programación y que estén aprobadas. Luego la función de agregación COUNT calcula cuantas tuplas hay en ese conjunto.

Ejemplo 19: informar cuantos alumnos aprobaron al menos un final durante 2008.

```
SELECT  COUNT (*)
FROM   inscripciones
WHERE resultado > 3 AND año = 2008
```

Esta consulta informa la cantidad de finales aprobados durante el 2008, sin embargo no informa cuantos alumnos fueron los que aprobaron. Suponga que un alumno aprobó 3 finales, entonces dicho alumno es contabilizado 3 veces, por lo tanto esta consulta no es una solución al ejercicio.

```
SELECT  COUNT (DISTINCT *)
FROM   inscripciones
WHERE resultado>3 AND año=2008
```

La segunda solución propuesta solo cuenta las tuplas diferentes. El objetivo es evitar contar a un mismo alumno más de una vez, en caso que haya aprobado varias materias. Nuevamente la consulta fracasa. Tener en cuenta que el operador DISTINCT evita contar las tuplas repetidas. Pero el conjunto en cuestión contiene tuplas con varios atributos proyectados, entre ellos el atributo idinscripcion, que está definido como CP y por lo tanto no se repite. El resultado obtenido por la segunda consulta es igual al obtenido con la primera y no corresponde, aún, a la solución del ejemplo 19.

```
SELECT  COUNT ( DISTINCT idalumno )
FROM   inscripciones
WHERE resultado > 3 AND año = 2008
```

Ahora, la consulta plantea proyectar del conjunto original solo el atributo idalumno. Así, si un idalumno aparece varias veces, significa que ese alumno aprobó varios finales durante 2008. Pero como solo interesa la cantidad de alumnos, el operador DISTINCT cuenta una sola aparición en caso que haya varias. Entonces, esta última solución es la correcta, retorna cuanto alumnos aprobaron al menos un final durante 2008.

Ejemplo 20: Informar la nota promedio (con y sin aplazos) del alumno Montezanti.

```
SELECT  AVG( resultado )
FROM   inscripciones i, alumnos a
WHERE a.nombre = "Montesanti" AND i.idalumno = a.idalumno
```


Tanto para la función de agregación AVG como para la función de agregación SUM el atributo a promediar o sumar debe ser de tipo número. En su defecto, la consulta retornará error.

Las funciones de agregación tienen una particularidad importante. Por definición estás funciones se aplican sobre un conjunto de tuplas. Por este motivo NO pueden estar definidas dentro de una cláusula WHERE. Intentar utilizar una función de agregación en el WHERE significa aplicar una función aplicable a conjuntos de una sola tupla.

Además, una función de agregación siempre retorna un resultado. No siempre es posible retornar además de dicho resultado, el valor de otro atributo.

Ejemplo 21: retornar la carrera de mayor duración.

```
SELECT  MAX( duracion_años ), nombre
FROM carreras
```

Esta consulta es incorrecta. La función de agregación retorna la máxima duración encontrada para cualquier carrera en la tabla Carreras. Suponga que en la tabla existen tres carreras con duración de 6 años y que la misma es máxima. ¿Cual de las tres carreras retorna asociado a los 6 años de duración? Se podría suponer que la consulta devuelve 3 registros. Sin embargo la función de agregación siempre retorna un resultado. Esto plantea una incongruencia. La consulta anterior es incorrecta, no es posible retornar ningún valor asociado a la función de agregación. Mas adelante en este capítulo se presenta una solución para esta consulta cuando se expliquen consultas anidadas.

Además, la siguiente sección plantea una excepción a la regla anterior.

Funciones de Agrupación

En muchos casos resulta necesario agrupar las tuplas de una consulta por algún criterio, aplicando una función de agregación sobre cada grupo. Por ejemplo si se quiere conocer el promedio de cada alumno que cursa informática, se deben agrupar todos los resultados obtenidos por cada alumno de informática y sobre cada grupo obtener un promedio. Para estas situaciones SQL prevé la cláusula GROUP BY.

```
SELECT  a.nombre, AVG (i.resultado)
FROM   inscripciones i, alumnos a
WHERE a.idalumno = i.idalumno
GROUP BY a.nombre
```

La cláusula GROUP BY genera n subconjuntos de tuplas, cada uno por un nombre de alumno diferente. Luego, dentro de cada grupo se obtiene el promedio del atributo resultado.

Se puede notar que en la cláusula SELECT se proyecta, además, el atributo nombre del alumno. Si bien en la última parte de la sección anterior esta situación no estaba permitida, se estableció que la regla tenía una excepción. Este es el caso, el promedio obtenido corresponde a un subconjunto (subgrupo) de un alumno en particular. Lo que se está proyectando junto con la nota promedio es el nombre del alumno por el cual se generó el subgrupo. Como este nombre es único para el grupo, se proyecta un solo nombre de alumno junto con su promedio.

Cuando se genera un agrupamiento por un criterio (atributo) es posible proyectar en el SELECT el atributo por el cual se genera el grupo.

Hay consultas que requieren obtener resultados para grupos que satisfagan una determinada condición. SQL prevé para esos casos la cláusula HAVING, la cual actúa como filtro de grupos. Dentro de la cláusula HAVING es posible indicar una condición que debe cumplir el grupo para ser tenido en cuenta.

Ejemplo 22: Presentar para cada carrera la cantidad de materias que la componen, solo mostrar en el resultado aquellas carreras con más de 20 materias.

```
SELECT  c.nombre, count ( * )
FROM    carreras c, materias m
WHERE c.idcarrera = m.idmateria
GROUP BY c.nombre
HAVING count ( * ) > 20
```

Se puede observar que dentro de la cláusula HAVING es posible utilizar, nuevamente, una función de agregación. En el HAVING se controla la validez de un grupo de tuplas, por lo tanto es posible utilizar una función de agregación. En este ejemplo la función de agregación retorna un valor entero, por lo que es posible mediante un operador compararlo contra un valor literal (20), el resultado será un valor booleano Verdadero o Falso. Si se da el primer caso el grupo se presenta en el resultado final.

Ejemplo 23: presentar para cada alumno el total de veces que rindió, siempre que el promedio de sus notas supere siete.

```
SELECT  a.nombre, COUNT( i.idmateria  )
FROM    alumnos a, inscripciones i
WHERE a.idalumno = i.idalumno
GROUP BY a.nombre
HAVING AVG ( i.resultado ) > 7
```


Consultas con subconsultas

Una consulta con subconsultas, también conocidas como subconsultas anidadas, consiste en ubicar una consulta SQL dentro de otra. Las subconsultas se pueden utilizar en varias situaciones: 1) comprobar la pertenencia o no a un conjunto, 2) analizar la cardinalidad de un conjunto, 3) analizar la existencia o no de elementos en un conjunto.

SQL define operadores de comparación para subconsultas. Se analizan en esta sección algunos de ellos.

Cuando una subconsulta retorna un único resultado, es posible compararlo contra un valor simple. De esta forma, se puede presentar una solución a la consulta del ejemplo 21.

```
SELECT  nombre
FROM    carreras
WHERE duración_años=(SELECT MAX(duración_años)
                     FROM carreras)
```

La subconsulta generada retorna un único valor que se corresponde con la duración máxima para una carrera. Luego, se compara la duración de cada carrera contra la duración máxima. Aquella carrera en la que coincidan ambos valores será presentada en el resultado final.

Operaciones de pertenencia a conjuntos

SQL define el operador IN para comprobar si un elemento es parte o no de un conjunto

Ejemplo 24: informar el nombre de los alumnos que están inscriptos en carreras de 5 años de duración.

Si bien este ejemplo puede resolverse utilizando conceptos ya explicados hasta el momento, se presenta aquí otra solución viable:

```
SELECT  nombre
FROM    alumnos
WHERE idcarrera IN ( SELECT idcarrera
                     FROM carreras
                     WHERE duración_años = 5 )
```

Aquí la subconsulta retorna un conjunto formado por los identificadores de carreras correspondientes a carreras de 5 años de duración. Para cada tupla de la tabla alumnos, se verifica si el código de carrera en la cual el alumno está inscripto está o no definido en el conjunto obtenido. Si está en el conjunto obtenido, el nombre del alumno es parte del resultado final debido a que se cumple la condición preestablecida (que el alumno esté inscripto en una carrera de 5 años de duración).

Ejemplo 25: mostrar las materias que no hayan sido aprobadas por ningún alumno.

```
SELECT  nombre
FROM    materias
WHERE idmateria  NOT  IN  ( SELECT idmaterias
                            FROM inscripciones
                            WHERE resultado > 3 )
```

En este caso, el operador IN está precedido por el operador booleano NOT. Para presentar una materia, su código no debe ser parte de subconjunto generado. En el subconjunto aparecen los códigos de todas las materias que han sido aprobadas por algún alumno.

Operaciones de comparación contra conjuntos.

Además del operador IN, se pueden utilizar operadores de comparación de elementos simples sobre conjuntos. Estos operadores, de funcionamiento similar a los operadores relacionales (=, <, >, >=, <= o <>), están combinados con las palabras reservadas SOME y ALL. Así, es posible realizar comparaciones del tipo (a modo de ejemplo)

- =SOME, significa igual a alguno. El resultado será verdadero si el elemento simple resulta igual a alguno de los elementos contenidos en el conjunto, falso en caso contrario.
- >ALL, significa mayor que todos. El resultado será verdadero si el elemento simple resulta mayor que todos los elementos del conjunto.
- <= SOME, significa menor o igual que alguno. El resultado será verdadero si el elemento simple resulta menor o igual que alguno de los elementos del conjunto.

Para definir algunos ejemplos adicionales se agrega al modelo original las siguientes tablas

DOCENTES = (iddocente, nombre, sueldo, idcarrera)
DOCENTESMATERIAS=(iddocentemateria, iddocente, idmateria, añodictado)

Que representa a los docentes, los cuales solamente pueden ser docentes de una carrera, y las materias que dictan año por año.

Ejemplo 26: mostrar aquellos docentes que cobren un sueldo superior a algún docente de la carrera de Quimica.

```
SELECT  nombre
FROM    docentes
WHERE sueldo >SOME ( SELECT sueldo
                     FROM docentes d, carreras c
                     WHERE d.idcarrera = c.idcarrera AND
                           c.nombre= "Quimica")
```


Ejemplo 27: Mostrar todos los docentes con sus salarios, excepto el docente que cobre mayor sueldo.

```
SELECT  nombre, sueldo
FROM    docentes
WHERE sueldo < SOME ( SELECT sueldo
                      FROM docentes )
```

Otra forma de resolver

```
SELECT  nombre, sueldo
FROM    docentes
WHERE sueldo < >    ( SELECT MAX( sueldo )
                      FROM docentes )
```

Cláusula Exist

La cláusula EXIST se utiliza para comprobar si una subconsulta generó o no alguna tupla como respuesta. El resultado de la cláusula EXIST es verdadero si la subconsulta tiene al menos una tupla, y falso en caso contrario. No es importante para la subconsulta el o los atributos proyectados, debido a que solamente se chequea la existencia o no de tuplas.

Ejemplo 28: Mostrar el nombre de los alumnos que hayan aprobado algún final durante 2007.

```
SELECT  nombre
FROM    alumnos
WHERE EXIST ( SELECT *
              FROM inscripciones
    WHERE alumnos.idalumno = inscripciones.idalumno
              AND resultado > 3 )
```

En el tabla temporal que resuelve esta consulta estará contenido el nombre de un alumno siempre y cuando exista alguna tupla en la subconsulta. Para que la subconsulta tenga una tupla es necesario encontrar al menos una inscripción para ese alumno que resulte aprobada.

Ejemplo 29: mostrar aquellos alumnos que no se inscribieron en materia alguna

```
SELECT  a.nombre
FROM    alumnos a
WHERE NOT EXIST ( SELECT *
                  FROM inscripciones i
                  WHERE a.idalumno = i.idalumno)
```

La subconsulta planteada devuelve para un alumno específico todas aquellas tuplas de inscripciones que le correspondan. Sino devolviera alguna inscripción, la cláusula NOT EXIST en la consulta principal resulta verdadera, y por lo tanto el alumno se presenta en la tabla resultado.

Ejemplo 30: mostrar aquellos alumnos que hayan inscripto en todas las materias de la carrera.

```
SELECT  a.nombre
FROM    alumnos a
WHERE NOT EXIST (SELECT *
                 FROM materia m
                 WHERE m.idcarrera = a.idcarrera AND
                       NOT EXIST(SELECT *
                                 FROM inscripciones i
                                 WHERE (i.idmateria = m.idmateria
                                       AND i.idalumno=a.idalumno )))
```

El análisis de esta consulta parece complejo. La primera cuestión a considerar para analizarla es el resultado al que se desea arribar: los alumnos que se hayan anotado en todas las materias de la carrera en la cual están inscriptos.

Es complejo resolver una consulta que involucre ".. todas las materias..:". Resulta más sencillo resolver esta consulta utilizando el razonamiento de la *contra-recíproca*. De este modo, se presentarán los alumnos tal que para ellos no exista ninguna materia en la que no estén inscriptos. Se puede notar que esta expresión es equivalente a "que estén inscriptos en todas".

Como primer medida, la consulta original exige que para mostrar un alumno no deben existir (NOT EXIST) tuplas correspondientes a materias. Ahora, una tupla de la primera subconsulta queda en el resultado, si corresponde a la misma carrera que el alumno original y no existen tuplas en la siguiente subconsulta.

La segunda subconsulta obtiene la inscripción de un alumno en una materia. Por lo tanto si un alumno no está inscripto en esa materia el conjunto queda vacío. Esto motiva a que el segundo NOT EXIST retorne verdadero y por ende la materia quede en el resultado ("el alumno no está inscripto en esa materia"). Al quedar el conjunto con un elemento, el primer NOT EXIST resulta falso y el alumno no se muestra.

Por el contrario, si un alumno se inscribió en todas las materias la segunda subconsulta siempre retorna un elemento (la inscripción del alumno en cada materia), lo que provoca que ninguna materia quede como resultado de la primera subconsulta. Así, el primer NOT EXIST dará como resultado verdadero y el alumno se presenta en el resultado final.

Suponga que las tablas ALUMNOS, MATERIAS, CARRERAS e INSCRIPCIONES tiene los datos presentados en al figura 15.1.

Figura 15.1

Alumnos

Idalumno	Nombre	Dni	Idlocalidad	Idcarrera
1	Juan Ignacio	23456876	1	1
2	Victoria	17876234	2	1
3	Ezequiel	33009876	1	2
4	Serena	25678965	1	2

Materias

Idmaterias	Nombre	Año_curso	Idcarrera
1	Programación	1	1
2	Matemática	1	1
3	IBD	2	1
4	IS	2	1
5	Conceptos	3	1
6	Concurrente	3	1

Carreras

Idcarrera	Nombre	Duración_años
1	Informatica	5
2	Quimica	3
3	Contador	4

Inscripciones

Idinscripcion	Idalumno	Idmateria	Año	Resultado
1	1	1	2007	6
2	1	2	2008	8
3	1	3	2008	9
4	1	4	2008	4
5	1	5	2009	6
6	1	6	2009	8
7	2	1	2007	10
8	2	2	2008	8
9	2	3	2008	9
10	3	1	2007	7
11	3	2	2007	6
12	3	3	2008	9
13	3	4	2008	8
14	3	5	2009	7
15	3	6	2009	8
16	4	1	2009	5

El resultado que queda en la consulta contiene a los alumnos:

Juan Ignacio
Ezequiel

Porque son los únicos que poseen inscripciones en todas las materias. La alumna Victoria solo tiene inscripción en tres materias, en tanto que Serena sólo está inscripta en una asignatura.

Operaciones con Valores Nulos

En SQL se debe tener especial cuidado cuando un atributo puede contener valores nulos. En general, cuando se resuelve un algoritmo y el programador debe preguntar si una variable tiene valor nulo genera una expresión del tipo:

Variable = ''

Al utilizar esta expresión, comillas sin ningún carácter en su interior se asume valor nulo. Sin embargo este tipo de expresiones no tienen un comportamiento similar cuando se trata de consultas SQL. Cuando un dominio de un atributo puede contener valores nulos no puede tratarse de esta forma. Los dominios con posibles valores nulos incorporan al conjunto de valores posibles el valor NULL. Este valor se almacena por defecto si el usuario no define otro. Entonces, una consulta que pregunta por un string nulo ('') no una respuesta satisfactoria. Para estas situaciones se define un nuevo operador IS NULL o su negación IS NOT NULL.

Ejemplo 30: presentar todos los alumnos que no tengan defina una localidad de procedencia.

```
SELECT  a.nombre
FROM    alumnos a
WHERE idlocalidad IS NULL
```

Productos naturales

En el capítulo anterior se presento al producto natural como una operación adicional del AR que permite resolver de una forma más eficiente el producto cartesiano. Hasta el momento, en SQL se presentó al producto cartesiano como única opción para ligar tablas.

Por las mismas cuestiones de *performance* planteadas en el capítulo 14 es necesario que SQL presente una alternativa al producto cartesiano, el producto natural. Las sentencias disponibles al efecto son: INNER JOIN, LEFT JOIN, RIGTH JOIN, FULL JOIN. A continuación se detallan y muestran ejemplos de cada una de ellas.

Para realizar un producto natural como fue definido en el capítulo 14, debe utilizarse la cláusula INNER JOIN.

Ejemplo 30: presentar todos los alumnos y su localidad de procedencia.

```
SELECT a.nombre, l.nombre
FROM   alumnos a
       INNER JOIN localidades l ON a.idlocalidad = l.idlocalidad
```

Nótese que en la definición del producto natural se realiza en la cláusula FROM indicando la tablas involucradas en el producto, y luego de la sentencia ON la condición que debe cumplirse.

El ejemplo 9 puede resolverse, ahora, de la siguiente forma

```
SELECT  a.nombre, c.nombre, l.nombre
FROM alumnos a
     INNER JOIN  carreras c ON (a.idcarrera = c.idcarrera )
     INNER JOIN localidades l ON (a.idlocalidad = l.idlocalidad)
```

Los otros productos tienen similar *performance*, pero su comportamiento es diferente. Mientras que el INNER JOIN realiza un producto natural clásico, reuniendo las tuplas de las relaciones que tienen sentido, tanto LEFT como RIGTH y FULL operan con algunas diferencias.

Suponer:

```
Tabla1  LEFT JOIN  Tabla2
```

En primera instancia se reúne cada tupla de la tabla 1 con su correspondiente tupla en la tabla2, similar al INNER JOIN. Luego, si alguna tupla de la tabla 1 no se reúne con una tupla de la tabla 2, igualmente aparece en el resultado, con los atributos de la tabla 2 con valor nulo.

Ejemplo 31: presentar cada carrera con sus materias.

```
SELECT c.nombre, m.nombre
FROM  carreras c
      LEFT JOIN  materias m ON (m.idcarrera = c.idcarrera)
```

Si se utiliza la figura 15.1 como contenido de las tablas carreras y materias el resultado final será:

Carrera	Materia
Informática	Programación
Informática	Matemática
Informática	IBD
Informática	IS
Informática	Conceptos
Informática	Concurrente
Química	NULL
Contador	NULL

Las restantes cláusulas operan de manera similar.

RIGTH JOIN genera, primero, un INNER JOIN para luego completar con las tuplas de la tabla 2 que no tienen correspondencia en la tabla 1. Se puede notar, entonces, que las siguientes expresiones resultan equivalentes:

```
Tabla1  LEFT JOIN tabla2
Tabla2  RIGHT JOIN tabla1
```

Por ultimo, FULL JOIN, genera el equivalente a un INNER JOIN, LEFT JOIN y RIGTH JOIN en conjunto. Es decir, reúne las tuplas con sentido, luego agrega las tuplas de las tabla 1 sin correspondencia en la tabla 2 y, por último, reúne las tuplas de tabla2 sin correspondencia en tabla1.

Otras operaciones: Insertar, Borrar y Modificar

Hasta el momento se discutieron y ejemplificaron alternativas, las más comunes y utilizadas, para realizar consultas sobre una BD. Como se indicó anteriormente en este libro, el 80% de las operaciones que se realizan en una BD son de consulta. Sin embargo, las restantes tres operaciones también son necesarias: agregar, borrar y/o modificar la información.

Insertar tuplas a la BD

La cláusula utilizada para agregar tuplas a una tabla es la cláusula INSERT INTO. El siguiente representa un ejemplo de inserción de una tupla a la tabla ALUMNOS.

```
INSERT INTO alumnos (nombre, dni, idlocalidad, idcarrera )
VALUES ('Julio Cesar', '12344321', 3, 1);
```

Se puede observar que se indican cuatro de los cinco atributos que componen la tabla alumnos. El atributo idalumno está definido como autoincremental. Por este motivo, será el DBMS el encargado de asignarle el valor correspondiente. Si el usuario intentara asignar un valor a un atributo definido como autoincremental, la operación de inserción falla.

Se debe tener cuidado, además, con los dos últimos atributos dado que representan claves foráneas. En caso de haber exigido integridad referencial estricta, se debe indicar identificadores válidos. Esto significa que el número 3 correspondiente a idlocalidad debería tener su correlato en la tabla localidades, y en forma análoga en número

1 debe existir en la tabla de carreras. En su defecto, nuevamente la consulta falla.

La cláusula INSERT INTO genera una sola tupla cada vez que es ejecutada.

Borrar tuplas de la BD

Para borrar una tupla o un conjunto de tuplas de una tabla se utiliza la cláusula DELETE FROM.

Ejemplo 32: borrar todos los alumnos de la localidad de La Plata.

```
DELETE FROM alumnos
WHERE idlocalidad = (SELECT idlocalidad
                     FROM localidades
                     WHERE nombre = "La Plata")
```

Si la cláusula definida fuera

```
DELETE FROM alumnos
```

Se borran todas las tuplas definidas en alumnos.

Por último, debe notarse que nuevamente la integridad referencial tendrá especial énfasis sobre la cláusula DELETE. Esto es, solamente pueden ser borradas de la tabla aquellas tuplas que cumplan las condiciones de integridad referencial definidas.

Modificar tuplas de la BD

Por último, la cláusula UPDATE ... SET se utiliza para modificar el contenido de uno o varios atributos de una tabla.

Ejemplo 33: modificar la duración de la carrera Contador, llevándola a cinco años.

```
UPDATE carreras
SET duración_años = 5
WHERE nombre = "Contador"
```

Se debe tener en cuenta que si varias tuplas cumplen la condición estipulada en el WHERE, todas ellas serán modificadas. El siguiente ejemplo muestra como se incrementaría el salario de los empleados en un 20%

```
UPDATE empleados
SET salario  = salario * 1.2
```

Vistas

Una vista de una BD es un resultado de una consulta SQL que puede involucrar una o varias tablas. Las vistas tienen la misma estructura que una tabla: tuplas y atributos y son el resultado de guardar en forma temporaria el resultado de una consulta SQL. La única diferencia es que sólo se almacena de ellas la definición de la consulta, no los datos. Cada vez que la vista es utilizada, la consulta almacenada es "recalculada" para obtener una imagen actual de la BD.

Las vistas se crean por diversos motivos. Uno de ellos es lograr modularizar las consultas. Existen determinadas subconsultas muy utilizadas en la BD. Por ello, para evitar tener que definirlas dentro de consultas más amplias, cada vez que se necesita, se define la vista y esa subconsulta se utiliza como una tabla más.

El otro motivo que sugiere la creación de vistas está ligado con la seguridad de la BD. El DBA puede decidir que para determinados usuarios de la BD, no es conveniente que operen sobre todos los atributos de todas las tablas. Se generan así vistas, que solamente tienen definidos aquellos atributos que el usuario puede manipular. Se les asigna derechos de trabajo sobre esas vistas, y por ende para esos usuarios solamente existen los atributos que son visibles.

Para crear una vista, SQL prevé la siguiente cláusula: CREATE VIEW.

Ejemplo 33: crear una vista de nombre alumnos de La Plata, que solamente permita manipular el nombre y carrera de los alumnos de dicha ciudad.

```
CREATE VIEW alumnoslaplata  AS
(SELECT a.nombre as nombrealumno, c.nombre as carrera
 FROM  alumnos a
       INNER JOIN carreras c ON (c.idcarrera = a.idcarrera)
       INNER JOIN localidades l ON (a.idlocalidad=l.idlocalidad)
 WHERE l.nombre = "La Plata" )
```

Ejemplos adicionales

Dado el siguiente modelo físico de datos:

PRODUCTOS = (idproducto, nombre, códigobarra, preciocosto)
CLIENTES = (idcliente, nombre, dirección, idlocalidad)
FACTURAS = (idfactura, fecha, montofactura, idcliente)
RENGLONES = (idfactura, #renglon, idproducto, precioventa, cantidad)
LOCALIDADES = (idlocalidad, nombre)

Se plantean los siguientes ejercicios y su resolución.

Ejercicio a: presentar todos las Facturas de agosto de 2008.

```
SELECT  *
FROM    facturas
WHERE YEAR( fecha) = 2008 AND MONTH( fecha ) = 8
```

Ejercicio b: mostrar la dirección de los clientes que viven en La Plata

```
SELECT  c.direccion
FROM    clientes c, localidades l
WHERE l.idlocalidad = c.idlocalidad AND l.nombre = "La Plata"

SELECT  c.direccion
FROM clientes c
     INNER JOIN localidades l ON (l.idlocalidad= c.idlocalidad)
WHERE l.nombre = "La Plata"
```

Ejercicio c: mostrar todas las facturas (fecha y monto) del cliente García ordenadas por fecha de emisión (de la más reciente a la menos reciente) y por monto.

```
SELECT  f.fecha, f.montofactura
FROM facturas f
     INNER JOIN clientes c ON (c.idcliente = f.idcliente )
WHERE c.nombre = "Garcia"
ORDER BY f.fecha DESC, f.montofactura
```

Ejercicio d: Presentar el monto total facturado en los primeros 15 días de enero del 2009.

```
SELECT  SUM( montofactura )
FROM    facturas
WHERE fecha BETWEEN  '01/01/2009' and '15/01/2009'
```

Esta consulta puede variar entre diferentes DBMS dependiendo de las operaciones para el manejo de datos tipo fecha que se tengan implementadas

Ejercicio e: presentar el nombre del cliente siempre que haya comprado el producto "Jabón en Polvo", ordenado por nombre del cliente.

```
SELECT  cl.nombre
FROM facturas fac
     INNER JOIN clientes cl ON (cl.idcliente= fac.idfacturas)
     INNER JOIN renglones ren ON (ren.idfactura=fac.idfactura)
     INNER JOIN productos pro ON (ren.idproducto= pr.idproducto)
WHERE productos.nombre = "Jabón en Polvo"
ORDER BY cliente.nombre
```

Ejercicio f: informar cuantos clientes compraron algo este mes:

```
SELECT  COUNT (DISTINCT fac.idcliente )
FROM facturas fac
     INNER JOIN clientes cl ON (cl.idcliente= fac.idfacturas)
     INNER JOIN localidades l ON (cl.idlocalidad=l.idlocalidad )
WHERE MONTH (CURRENT_DAY( ) ) = MONTH ( facturas.fecha ) AND
      YEAR( CURRENT_DAY( ) ) = YEAR( facturas.fecha )
```

En este caso se utilizan dos funciones definidas en ANSI SQL92, MONTH la cual retorna el mes de un valor de tipo fecha, mientras que CURRENT_DAY es una función que retorna la fecha del sistema. Lo que se compara, entonces, es el mes correspondiente a la fecha actual del sistema contra el mes de emisión de cada factura. Luego, con la función YEAR se hace lo mismo con el el año.

Ejercicio g: informar cual es el producto más barato.

```
SELECT  productos.nombre
FROM    productos
WHERE productos.preciocosto = (SELECT MIN (preciocosto)
                               FROM productos )
```

Ejercicio h: presentar todas la facturas del mes de enero del 2009 excepto la de mayor monto en ese mes.

```
SELECT *
FROM   facturas
WHERE (MONTH (fecha) = 1 AND YEAR (fecha) = 2009 AND
       montofactura < (SELECT MAX (montofactura)
                       FROM facturas
                       WHERE (MONTH(fecha)=1 AND YEAR(fecha)=2009))
```

Ejercicio i: informar las facturas que incluyan productos con costo inferior a 30$.

```
SELECT fac.idfactura, fac.fecha, fac.montofactura
FROM factures fac
     INNER JOIN renglones ren ON (fac.idfactura=ren.idfacturas )
WHERE  renglones.idproducto IN (SELECT productos.idproducto
                                FROM productos
                                WHERE ( preciocosto < 30 ) )
```

Ejercicio j: informar el total de las ventas realizadas a cada cliente

```
SELECT c.nombre, SUM(montofactura)
FROM facturas f
     INNER JOIN clientes c ON (c.idcliente= f.idfactura)
GROUP BY c.nombre
```


Ejercicio k: informar el total de ventas de los clientes que hayan comprado más de 5 productos (no necesariamente diferentes).

```
SELECT c.nombre, SUM( montofactura )
FROM facturas f
     INNER JOIN clientes c ON (c.idcliente=f.idfactura)
     INNER JOIN renglones r ON (r.idfactura=f.idfactura)
GROUP BY clientes.nombre
HAVING SUM( renglones.cantidad ) > 5
```

Ejercicio l: informar los nombres de las localidades correspondientes a clientes que hayan efectuado alguna compra.

```
SELECT  nombre
FROM  localidades
WHERE EXIST (SELECT *
             FROM clientes c
                  INNER JOIN facturas f ON (c.idcliente=f.idfactura)
             WHERE ( c.idlocalidad = localidades.idlocalidad ) )
```

Ejercicio m: informar los productos que no fueron vendidos alguna vez.

```
SELECT  nombre
FROM  productos
WHERE NOT EXIST (SELECT *
                 FROM renglones r
                 WHERE (r.idproducto= productos.idproducto))
```

Ejercicio n: indicar el número de factura y su monto, siempre que en la factura se hayan vendido todos los productos existentes.

```
SELECT  f.idfactura, f.montofactura
FROM  facturas f
WHERE NOT EXIST (SELECT *
                 FROM producto p
                 WHERE NOT EXIST (SELECT *
                                  FROM renglones r.
                                  WHERE (r.idproducto=p.idproducto AND
                                         r.idfactura = f.idfactura)))
```

Ejercicio ñ: se desea presentar el nombre de cada cliente, su dirección, y la localidad de residencia. Aquellos que no tengan definida localidad también deben figurar en el listado.

```
SELECT  c.nombre c.direccion, l.nombre
FROM clientes c
     LEFT JOIN localidades l ON ( c.idlocalidad=l.idlocalidad )
```

Ejercicio o: resolver el ejercicio anterior, pero ahora se debe informar, además aquellas localidades que no tienen algún cliente.

```
SELECT c.nombre c.direccion, l.nombre
FROM  clientes c
      FULL JOIN localidades l ON (c.idlocalidad=l.idlocalidad )
```

Introducción al QBE

Los lenguajes de consulta propuestos a principios de la década del 70 por E.Codd fueron de dos tipos: procedurales y no procedurales. Dentro de los primeros fue tratado el AR, y posteriormente en este capítulo, se presentó SQL como la derivación de AR que dispone la mayoría de los DBMS.

Dentro de los lenguajes no procedurales se presentaron el Cálculo de Tuplas y el de Dominios. Este último, es el origen teórico del denominado QBE, por sus siglas en ingles de Consulta por Ejemplo (Query by example).

QBE es de los primeros lenguajes de consulta gráfico con un requerimiento mínimo de sintaxis para expresar una consulta a la BD. Se basa en expresar las necesidades de usuario a partir de plantillas, donde se seleccionan primero las tablas de la BD de donde se extrae la información, para luego indicar qué atributos se requieren presentar, y qué condiciones deben cumplimentar estos atributos.

La mayoría de los DBMS comerciales en la actualidad disponen la posibilidad de realizar consultas utilizando QBE. El aprendizaje de este lenguaje es bastante intuitivo, y no es propósito de este libro profundizar en este concepto.

Cuestionario del capítulo

1. ¿En cuál lenguaje teórico está basado SQL?
2. ¿Cuáles son las componentes básicas y obligatorias de toda consulta SQL?
3. Si en la cláusula FROM se definen dos tablas, ¿qué operación de AR se produce?
4. ¿Qué son las funciones de agregación?
5. ¿Por qué las funciones de agregación no se pueden definir en la cláusula WHERE?
6. ¿Por qué una función de agregación debe aparecer sola en la cláusula SELECT?
7. ¿En qué caso no se cumple la respuesta anterior y por qué?
8. ¿Las funciones de agregación pueden utilizarse en un HAVING? ¿Por qué motivo?
9. ¿La respuesta anterior podría permitir que las funciones de agregación se utilicen en la cláusula WHERE? ¿Por qué motivo?
10. ¿En qué casos resulta útil la definición de subconsultas?
11. ¿Cómo actúa la cláusula EXIST?
12. ¿En qué casos se pueden utilizar los operadores SOME y ell ALL?
13. ¿Cómo define el producto natural SQL?
14. Indique las diferencias entre INNER JOIN, LEFT JOIN, RIGHT JOIN y FULL JOIN.

Ejercitación

Dadas las siguientes tablas:

Cliente (id_cliente, nombre, renta_anual, idciudad)
Envio (idenvio, id_cliente, peso, fecha, idtransporte, idciudad)
Transporte (id_transporte, nombreconductor, idtipotransporte)
Ciudad (idciudad, nombre)
TipoTransporte (idtipotransporte, descripción)

Resolver las siguientes consultas en SQL:

a. Nombre del cliente cuyo id es 321.

b. Fecha del envío número 1134.

c. Ciudad de destino de los envíos de más de 50 unidades de peso.

d. ¿A qué ciudad fueron los envíos de más de 20 kilogramos?

e. Presentar una lista alfabética de clientes de Buenos Aires.

f. Presentar los id de todos los clientes ordenado de mayor a menor.

g. Presentar la fecha del envío y el cliente que lo realizó, para todos los envíos de Mayo del 2009.

h. Presentar el nombre del cliente y el peso del envío, para todos aquellos envíos realizados por clientes que vivan en la misma ciudad hacia donde se realiza el envío.

i. Nombres de los clientes que tienen una H como segunda letra del nombre.

j. De los nombres de los clientes cuya renta anual esté entre 200000 y 500000 pesos.

k. Nombre de los clientes que han enviado paquetes a Bariloche. Resuelva el ejercicio con subconsulta.

l. Total de kg. enviados por el cliente cuyo id es 34 durante el 2008.

m. Nombre de las ciudades que recibieron envíos el 3 de enero del 2009. Ordenado por nombre de ciudad. Utilizar para resolverlo una subconsulta.

n. Nombre del cliente con mayor renta anual.

o. Nombre de los clientes que hayan realizado envíos a todas las localidades.

p. Conductores que hayan realizado viajes a todas las ciudades.

q. Conductores que hayan visitado más de 15 ciudades.

r. Total de kg. recibidos por cada localidad.

s. Clientes que realizaron envíos en todos los medios de transporte.

t. ¿Cuál es el peso promedio de los envíos realizados por avión?

u. Informar el código del envío más pesado de cada cliente.

v. ¿Cuántos envíos han sido realizados por el cliente 433 a Villa La Angostura durante 2008?

w. Agregue la ciudad de Bahía Blanca a la tabla ciudades.

x. Agregue el cliente Gómez, Alberto, con renta anual de 140.000$, y que es de Bahía Blanca. Para ello, debe tener en cuenta que el atributo idciudad debe contener el código asignado en la tabla ciudades a Bahía Blanca.

y. Borre los tipos de transporte que no tengan definido ningún transporte.

z. Los pesos de los envíos están expresados en Kg. Genere una modificación de la tabla envíos para expresarlos en gramos.

aa. Cree una vista para cada uno de los siguientes casos:

i. Clientes con renta anual por debajo de un millón.

ii. Clientes con renta anual entre uno y 5 millones.

iii. Clientes con renta anual mayor de 5 millones.

bb. Utilice las vistas anteriores para responder a las siguientes consultas:

i. ¿Qué conductores han transportado embarques a General Pico provenientes de clientes con renta superior a 5 millones?

ii. ¿Cuál es la población de las ciudades que han recibido embarques de clientes con renta entre uno y 5 millones?

iii. ¿Qué conductores han transportado embarques de clientes con renta por debajo del millón y cuál es la población de las ciudades hacia las que se han enviado dichos embarques?

CAPÍTULO 16

Optimización de consultas

Objetivo

Este capítulo tiene por objetivo principal estudiar las consultas generadas y determinar el óptimo para su resolución. Tal como se presentó a partir de los ejemplos en los Capítulos 14 y 15, una consulta en AR o en SQL puede ser resuelta de muchas formas diferentes sin afectar el resultado final.

Cada resolución a un problema puntual tiene ventajas y desventajas respecto de otras alternativas. En algunos casos, resulta más sencillo para el programador resolver la consulta utilizando una determinada heurística; en otras situaciones, el estado actual de la BD hace más conveniente otro tipo de resolución. En general, el diseñador de la consulta trata de centrar sus objetivos en obtener el resultado esperado, en vez de analizar con detalle cuál sería la mejor secuencia de pasos a aplicar, para lograr una solución en el menor tiempo de respuesta posible.

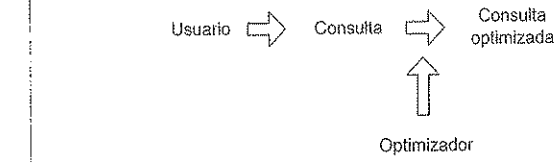
A lo largo de este capítulo, se estudiarán situaciones muy claras que presentan una alternativa de solución más eficiente para determinadas consultas. Además, se analizarán los optimizadores de consulta que en general están definidos en los SGBD de mercado. Nuevamente, sin tomar una herramienta particular, se discutirá la forma en que las heurísticas de resolución pueden plantearse, para poder encontrar consultas equivalentes más eficientes, basadas no solo en las características del AR, sino también en el estado temporal de la BD.

Análisis de procesamiento de consultas

Los SGBD presentan en general un optimizador de consultas. Se denomina optimizador de consultas a un proceso del gestor de BD, encargado de encontrar una consulta equivalente a la generada por el usuario, que sea de óptima en términos de *performance* para obtener el resultado deseado.

El proceso de optimización, como se presenta en la Figura 16.1, comienza con la consulta generada por el usuario; el optimizador de consulta la analiza y la compara contra el estado actual de la BD, y genera una consulta equivalente en cuanto a la respuesta que se obtendrá de ella, pero más eficiente en términos de procesamiento.

FIGURA 16.1



Analizando con más detalle el comportamiento del optimizador de consulta (Figura 16.2), el proceso de optimización de consultas comienza con la consulta generada por el usuario, aplicando la siguiente secuencia de pasos:

1. Un analizador sintáctico (*parser*) genera una expresión manipulable por el optimizador de consultas. El análisis sintáctico convierte al texto de entrada –en este caso, la consulta del usuario– en una estructura tipo árbol, que resulta más útil para su análisis.
2. A partir de la expresión interna, el optimizador obtiene una consulta equivalente más eficiente. En el apartado sobre evaluación del costo de las expresiones, se presenta la comparación entre operaciones equivalentes.
3. Por último, el proceso de optimización considera el estado actual de la BD y los índices definidos, para resolver la consulta que se tiene hasta el momento, con el acceso a disco posible.

A lo largo de este capítulo, se analiza con detalle el comportamiento de los dos primeros pasos de este proceso.

Medición del costo de consultas

En el Capítulo 5, se estableció, como base para el análisis de la eficiencia de búsqueda en archivos, la cantidad de operaciones de entrada/salida que deben realizarse sobre disco. Este parámetro fue considerado el más importante debido a que la velocidad de transferencia (medida en milisegundos) opaca los tiempos del procesamiento interno en memoria principal (medidos en nano o en milisegundos).

Para analizar con detalle el comportamiento del optimizador de consultas, se considera el número de accesos a disco como el factor determinante para estudiar la eficiencia de un método. Así, si una consulta genera 1.000 tuplas intermedias y se la compara contra una alternativa que solo genera 100 tuplas intermedias, la segunda solución será considerada más eficiente debido a que el acceso a 100 tuplas es más rápido que el acceso a 1.000 tuplas.

Para el análisis del costo de las consultas, se utilizarán algunos conceptos genéricos. Estos conceptos tienen que ver con el estado temporal de cada tabla de datos, los que permiten definir algunos valores.

El primero de estos valores, denominado CT_{tabla}, indica la cantidad de tuplas que actualmente tiene la tabla. El segundo valor, CB_{tabla}, indica el tamaño en bytes de cada tupla de la tabla. Luego, para cada atributo de la tabla se define un valor, denominado CV (a, tabla), que indica cuántos valores diferentes existen en la tabla para el atributo a. Así, suponiendo la tabla inscripciones de la Figura 14.2, en el Capítulo 4:

CV(año, inscripciones) = 3 (están definidos los años 2007, 2008 y 2009)
CV(resultado, inscripciones) = 6 (están definidos los resultados 4, 6, 7, 8, 9 y 10)

Para obtener $CT_{inscripciones}$, se puede realizar la siguiente consulta SQL:

```
SELECT COUNT (*)
FROM inscripciones
```

Para obtener CV (año, inscripciones) se puede realizar la siguiente consulta SQL:

```
SELECT COUNT (DISTINCT (año))
FROM inscripciones
```

En apartados posteriores, se analizarán los costos de consultas a partir de la utilización de estos valores.

Evaluación de operaciones

El proceso de optimización de consultas comienza analizando la consulta generada por el analizador sintáctico. Se pueden aplicar diferentes consideraciones, en función de las operaciones indicadas. Para analizar estas operaciones se utilizará el AR, dado que este lenguaje resulta más claro para explicar el comportamiento.

Operación de selección

Existen varias consideraciones respecto de la operación de selección.

Una de ellas consiste en realizar la operación de selección lo antes posible en una consulta. Suponga que sobre las tablas de la Figura 14.1 se desean obtener los DNI de los alumnos de La Plata y que para ello se genera la siguiente consulta AR:

$$\pi_{dni}(\sigma_{localidades.nombre = \text{"La Plata"}}(\text{alumnos} \mid\times\mid \text{localidades}))$$

Si se tiene en cuenta el estado de la BD presentado en la Figura 14.2, la secuencia de operaciones a realizar a partir de la expresión generada es

(alumnos |×| localidad)

Idalumno	Nombre	DNI	Idlocalidad	Idcarrera	Idlocalidad	Nombre
1	García	23456876	1	1	1	General Pico
2	Perez	17876234	2	1	2	La Plata
3	Gomez	33009876	1	2	1	General Pico
4	Pizarro	25678965	1	2	1	General Pico
5	Castelli	14239034	2	1	2	La Plata
6	Pettorutti	19023487	3	2	3	Tres Arroyos
7	Montenzanti	23434564	2	3	2	La Plata
8	Suarez	30212305	3	1	3	Tres Arroyos

Luego, la selección solo deja las tuplas con idalumno 2, 5 y 7. Se puede notar que el resto de las tuplas (cinco en total) fueron generadas inútilmente.

La siguiente expresión es equivalente a la anterior, pero más eficiente:

$$\pi_{dni}(\text{alumnos} \mid\times\mid (\sigma_{localidades.nombre = \text{"La Plata"}}(\text{localidades})))$$

En este caso, se aplica primero la selección sobre la tabla localidades. De esta forma solo queda una tupla, la correspondiente a La Plata. Esta tupla se relaciona con algunas tuplas de alumnos pero no con todas, generando solamente las tres tuplas que se obtenían con la consulta planteada inicialmente.

De esta manera, resolviendo la selección lo antes posible, se filtran las tuplas innecesarias del problema. Se debe recordar que los productos cartesianos, y aun los naturales, generan mucha información. Cuanto más pequeño sea el conjunto de entrada, más rápidamente será resuelta la consulta.

Este tipo de expresiones puede generalizarse a operaciones que involucren unión y diferencia. De ser posible, no solo debe realizarse la selección antes que el producto, sino que esto debe extenderse a las operaciones de unión y diferencia.

La segunda consideración que se puede efectuar sobre la selección tiene que ver con el predicado generado. Suponga que desea saber cuáles son los alumnos de la localidad cuyo id es 1, que están cursando la carrera cuyo id es 3. La siguiente expresión resuelve el problema:

$$\pi_{nombre}(\sigma_{idcarrera = 3 \text{ AND } idlocalidad = 1}(alumnos))$$

Suponga que hay 1.000 tuplas en la tabla alumnos. Sobre esas 1.000 tuplas se aplican los predicados (idcarrera = 3) e (idlocalidad = 1); en total se resuelven 2.000 expresiones lógicas. Para obtener el resultado final del predicado completo, debe resolverse el conectivo lógico AND, nuevamente para las 1.000 tuplas. Esto significa agregar 1.000 resoluciones más al resultado final, lo que significa a su vez un total de 3.000 expresiones lógicas resueltas.

Si la expresión se convierte a

$$\pi_{nombre}(\sigma_{idcarrera = 3}(\sigma_{idlocalidad = 1}(alumnos)))$$

se resuelve primero la selección de localidad. Solo si las 1.000 tuplas correspondieran a la localidad con id igual a 1, la consulta intermedia dejaría las 1.000 tuplas, pero si 20% correspondieran a la misma idlocalidad, quedarían 200 tuplas en el resultado.

Luego, la segunda expresión se aplica, en el peor de los casos, a 1.000 tuplas (o a las que quedaran resultantes de la consulta anterior), con el mismo resultado que la primera expresión. Lo importante en este caso es notar que, bajo la peor condición posible, solo se resuelven 2.000 expresiones lógicas (el primer filtro deja pasar todas las tuplas, 1.000 comparaciones, y el segundo filtro se debe aplicar 1.000 veces). En este caso, se eliminó la resolución del operador AND. No obstante, es esperable un resultado mejor, debido a la probabilidad de filtrar tuplas con la primera subconsulta.

Operación de proyección

En líneas generales, el análisis de una proyección indica que esta operación también debería realizarse lo antes posible.

Si bien los efectos de una selección son más importantes que los de una proyección debido a que la primera quita tuplas del resultado, la proyección reduce el tamaño de cada tupla. Esto significa que cada tupla ocupará menos espacio en el *buffer* en memoria y, por consiguiente, será posible almacenar más elementos.

Operación de producto natural

Una especial atención requiere el proceso de optimización del producto natural. Esto se debe a la frecuencia de aparición de este operador en las consultas sobre una BD. El producto natural es utilizado en muchas expresiones y genera como resultado un número importante de tuplas; por lo tanto, se debe tener en cuenta cómo se puede expresar una consulta.

Expresiones del tipo

tabla1 |×| tabla2 |×| tabla3

deben intentar resolverse en pasos, es decir, resulta más eficiente resolver primero

tabla1 |×| tabla2

y con el resultado obtenido realizar el producto natural con la tabla restante.

Además, desde un punto de vista de *performance*:

tabla1 |×| tabla2 <> tabla2 |×| tabla1

Debe tenerse en cuenta que el resultado de ambas expresiones es el mismo; lo que se plantea en este caso es que los tiempos de respuesta son diferentes.

El siguiente apartado analiza en términos numéricos los costos de cada consulta, y demuestra el caso planteado.

La tabla que sigue presenta ejemplos de expresiones equivalentes. En la columna de la derecha, aparece la consulta que puede resultar (en algunos casos) más eficiente en términos de *performance*.

Tabla 16.1

Consulta original	Expresión equivalente más eficiente
$\sigma_{predicado}$ (tabla1 × tabla2)	($\sigma_{predicado}$ (tabla1) × $\sigma_{predicado}$ (tabla2))
$\sigma_{predicado\ 1\ AND\ predicado\ 2}$ (tabla1)	$\sigma_{predicado\ 1}$ ($\sigma_{predicado\ 2}$ (tabla1))
$\sigma_{predicado}$ (tabla1 U tabla2)	($\sigma_{predicado}$ (tabla1) U $\sigma_{predicado}$ (tabla2))
$\sigma_{predicado}$ (tabla1 – tabla2)	($\sigma_{predicado}$ (tabla1) – $\sigma_{predicado}$ (tabla2))
(tabla1 |×| tabla2 |×| tabla3)	((tabla1 |×| tabla2) |×| tabla3))
(tabla1 |×| tabla2)	(tabla2 |×| tabla1)

Evaluación del costo de las expresiones

Los parámetros necesarios para evaluar el costo de las consultas ya fueron definidos. Se utilizarán CT_{tabla}, CB_{tabla} y CV (a, tabla) para evaluar el costo de las principales operaciones de AR. En cada caso, se analizarán la cantidad de tuplas resultantes y el tamaño en bytes de cada tupla.

Costo de la operación de selección

Sea la expresión

$$\sigma_{atributo\ =\ valor}(tabla1)$$

El tamaño en bytes de cada tupla es CB_{tabla}, debido a que no se genera proyección alguna.

Para analizar la cantidad de tuplas resultantes, se tendrán en cuenta la cantidad de valores diferentes para el atributo del predicado (CV(atributo, tabla)), y se supondrá que la distribución de valores es uniforme.

Suponga que una tabla tiene 1.000 tuplas y que para el atributo1 se tienen 20 valores diferentes. La distribución uniforme supone que es esperable tener 50 tuplas por cada valor diferente (1.000/20). Si bien esta consideración no tiene por qué ser real, es la suposición indispensable que se debe realizar para poder poner en práctica este análisis. Como conclusión, la cantidad de tuplas esperables de la consulta planteada será:

$$CT_{tabla1} / CV(atributo, tabla1)$$

Costo de la operación de proyección

La operación de proyección no afecta la cantidad de tuplas del resultado. Así, la expresión

$$\pi_{atributo}(tabla1)$$

genera CT_{tabla1} tuplas, pero el tamaño en bytes de las tuplas resultantes se reduce al espacio que requiere el dominio del atributo proyectado.

Costo de la operación de producto cartesiano

La operación de producto cartesiano es sencilla de analizar. La expresión

tabla1 |×| tabla2

genera tuplas de tamaño (CB_{tabla1} + CB_{tabla2}), debido a que cada tupla de tabla1 se agrupa con cada tupla de tabla2. Por esa misma causa, la cantidad de tuplas obtenidas será (CT_{tabla1} * CB_{tabla2}).

Costo de la operación de producto natural

Nuevamente, la operación de producto natural es la más interesante para considerar. El costo de la operación de producto natural tiene varias aristas. Estas dependen de las condiciones mismas del producto, así como la utilización de índices de la BD.

Al analizar el costo de la consulta se podrá, además, estudiar la razón de la última expresión de la Tabla 16.1.

Suponga primero que se desea realizar el producto natural entre tabla1 y tabla2, y entre ambas tablas no existe ningún atributo común. En este caso:

tabla1 |×| tabla2 equivale a tabla1 × tabla2

La operación de producto cartesiano es conmutativa en cuanto a *performance*:

tabla1 × tabla2 Número de tuplas: CT_{tabla1} * CT_{tabla2} Tamaño de tuplas: CB_{tabla1} + CB_{tabla2}	tabla2 × tabla1 Número de tuplas: CT_{tabla2} * CT_{tabla1} Tamaño de tuplas: CB_{tabla2} + CB_{tabla1}

Suponga ahora que entre tabla1 y tabla2 existe un atributo común, A. Este atributo es CP en tabla1 y CF en tabla2.

Si se realiza el producto natural

tabla2 |×| tabla1

cada tupla t de tabla2 se reúne con una y solo una tupla de tabla1. El atributo común (A) es CP en tabla1; por lo tanto, el valor que tiene A en tabla2 solo aparece una vez en tabla1. Además, la búsqueda se realiza por CP; por lo tanto, la respuesta es óptima.

Suponga que realiza el producto

tabla1 |×| tabla2

El atributo A en tabla2 es CF; por lo tanto, se administra como un índice secundario. Las búsquedas en índices secundarios son más lentas, y, además, una tupla t de tabla1 se reúne con varias tuplas de tabla2.

Sin dudas, cuando el producto natural se realiza entre dos tablas con un atributo común y este atributo es CP en una de las tablas y CF en otra, conviene siempre realizar la operación entre la tabla con CF sobre la tabla con CP.

Esta opción es en general la más común cuando se trabaja con la operación de producto natural, y se ha demostrado que el resultado en cuanto a *performance* no es igual.

Queda un caso más por analizar, que no es muy común que aparezca en las consultas sobre la BD. Entre tabla1 y tabla2 hay un atributo común, A, el cual no es CP en ninguna de las tablas.

El análisis en este caso es el siguiente: sea t1 una tupla de tabla1, con un determinado valor para el atributo A; suponiendo nuevamente distribución uniforme de valores para el atributo A, se puede esperar que en tabla2 existan (CT_{tabla2} / CV(A, tabla2)) tuplas que se relacionen con t1.

Ahora, en tabla1 se tienen CT_{tabla1} tuplas diferentes; por lo tanto, el resultado será:

$$\frac{CT_{tabla1} * CT_{tabla2}}{CV(A, tabla2)}$$ costo de la operación tabla1 |×| tabla2

Analizando ahora t2 ∈ tabla2, se reúne con (CT_{tabla1} / CV(A, tabla1)) tuplas de tabla1, y como en tabla2 hay CT_{tabla2} tuplas diferentes, el resultado será:

$$\frac{CT_{tabla2} * CT_{tabla1}}{CV(A, tabla1)}$$ costo de la operación tabla2 |×| tabla1

¿Cuál será la forma más adecuada de resolver la consulta anterior? La respuesta es tomar aquel camino cuyo costo sea el menor. Como

el dividendo es igual, lo importante para analizar es el divisor de la ecuación anterior. Se debe tomar aquella opción que tenga divisor mayor. Es decir, el producto debe realizarse sobre aquella tabla que tenga mayor cantidad de elementos diferentes.

Queda demostrado, a partir del análisis realizado, que el producto natural no es conmutativo en cuanto a *performance*.

El optimizador de consultas puede utilizar todos los conceptos definidos hasta el momento para modificar la consulta original, y encontrar aquella que más se adecue al caso particular de la BD.

Para poder resolver situaciones como la última planteada, el optimizador dispone de estadísticas de la BD. El SGBD mantiene estadísticas con respecto a la cantidad de tuplas de las tablas, así como la distribución de valores para los atributos. De esta forma puede decidir rápidamente el costo de cada alternativa de solución y aplicar la más conveniente.

Las estadísticas se mantienen parcialmente actualizadas. El costo de tenerlas todo el tiempo actualizadas es alto; cualquier inserción, borrado o modificación altera los valores. Para poder efectuar decisiones, basta con tener los datos parcialmente actualizados. Cada cierto tiempo, el gestor de BD ejecuta un proceso de actualización de las estadísticas.

Cuestionario del capítulo

1. ¿Qué significa optimizar una consulta?
2. ¿Por qué motivo es importante optimizar una consulta?
3. ¿Qué proceso del SGBD es el responsable de optimizar una consulta?
4. ¿En qué afectan los índices al proceso de optimización?
5. ¿Por qué las selecciones deben resolverse "lo antes posible" en una consulta?

V

Seguridad e integridad de datos

CAPITULO 17

Conceptos de transacciones

Objetivo

Preservar la integridad de los datos contenidos en una BD es una de las principales tareas que debe llevar a cabo el diseñador de la BD. Para poder asegurar la información en todo momento, es necesario tomar algunos recaudos.

Uno de los primeros aspectos a considerar tiene que ver con asegurar la información contra destrucción malintencionada. Esos conceptos serán analizados con más detalle en el Capítulo 19.

Sin embargo, existen muchas otras causas que pueden atentar contra el contenido de la BD sin que puedan atribuirse a la mala intención. Suponga que se daña el disco rígido de la computadora, que se cae un enlace de comunicaciones con el servidor de la BD o que sencillamente se corta la provisión de energía eléctrica. Si algún usuario se encuentra en ese momento operando contra la BD, puede ocurrir que se pierda parte del trabajo realizado. Esto acarrea un inconveniente importante; si el trabajo se realizó parcialmente, puede significar que parte de la BD tenga valores actualizados y otra parte no. Esta situación claramente viola el concepto de integridad de los datos. Una operación de usuario realizada de modo parcial atenta fuertemente contra la integridad de la información.

En este capítulo, se analizan técnicas que permiten asegurar la integridad de la BD ante situaciones imprevistas, no deseadas por el usuario y que carecen de mala intención para producirse.

En primer lugar, se presentan el concepto de transacción como unidad lógica de trabajo, sus propiedades, estados y entornos. Luego, se analizan distintos tipos de fallos que pueden ocurrir y como estos afectan la BD. Por último, se explican los métodos de recuperación de integridad de la información más difundidos: bitácora y doble paginación.

Concepto de transacción

Una transacción es una unidad lógica de trabajo. Para una BD, una transacción actúa como una única instrucción que tendrá éxito si se ejecuta completamente, o, en su defecto, nada de ella deberá reflejarse en la BD.

Una **transacción** es un conjunto de instrucciones que actúa como unidad lógica de trabajo. Esto es, una transacción se completa cuando se ejecutaron todas las instrucciones que la componen. En caso de no poder ejecutar todas las instrucciones, ninguna de ellas debe llevarse a cabo.

Por ejemplo, si se realiza el pago de un servicio por Internet, la operación consiste en transferir dinero desde una cuenta bancaria hacia otra cuenta bancaria. Las operaciones involucradas restan el monto correspondiente al pago de la cuenta del usuario, y lo suman a la cuenta del prestatario del servicio. Se puede observar que el número de instrucciones involucradas es importante. Sin embargo, desde el punto de vista lógico de la BD, toda la operación de pago debe realizarse, o absolutamente nada debe llevarse a cabo. La BD quedaría inconsistente si se registrara la salida de dinero de la cuenta del usuario, sin reflejarlo en la cuenta del prestatario.

Para garantizar que una transacción mantenga la consistencia de la BD, es necesario que cumpla con cuatro propiedades básicas:

1. **Atomicidad:** garantiza que todas las instrucciones de una transacción se ejecutan sobre la BD, o ninguna de ellas se lleva a cabo.
2. **Consistencia:** la ejecución completa de una transacción lleva a la BD de un estado consistente a otro estado de consistencia. Para esto, la transacción debe estar escrita correctamente. En el ejemplo del pago de un servicio, si la transacción resta $100 de la cuenta del usuario y suma 120 en la cuenta del prestatario, no resulta en una transacción consistente.
3. **Aislamiento (*isolation*, en inglés):** cada transacción actúa como única en el sistema. Esto significa que la ejecución de una transacción, T1, no debe afectar la ejecución simultánea de otra transacción, T2.
4. **Durabilidad:** una vez finalizada una transacción, los efectos producidos en la BD son permanentes.

Estas cuatro propiedades son conocidas bajo la denominación **ACID (Atomicidad, Consistencia, *Isolation* y Durabilidad).**

Suponga que la transacción para generar el pago de un servicio es la siguiente:


```
Leer (cuentacliente)
cuentacliente = cuentacliente – importefacturaservicio
Escribir (cuentacliente)
Leer (cuentaprestatario)
cuentaprestatario = cuentaprestatario + importefacturaservicio
Escribir (cuentaprestatario)
```

Se debe notar que las operaciones de entrada/salida (leer y escribir) operan sobre *buffers* de memoria. Por ende, la ejecución correcta de la última instrucción de una transacción no implica que la transacción haya finalizado. Puede ocurrir un error antes de bajar a disco el *buffer* de la BD.

En cuanto a la consistencia, antes de comenzar la ejecución de la transacción (cuentacliente + cuentaprestatario), genera un determinado resultado. Luego de la ejecución completa de la transacción, el estado se mantiene: se ha restado de la cuenta del cliente el importe del pago y se ha sumado a la cuenta del prestatario. Esta transacción es consistente, dado que la suma de dinero de ambas cuentas antes de comenzar la transacción es la misma que la suma de dinero de ambas cuentas luego de la transacción. La consistencia de una transacción es responsabilidad total de quien la programe.

Con respecto a la atomicidad, una transacción debe ejecutarse en forma completa o, en su defecto, nada de ella debe llevarse a cabo. Suponga que durante el procesamiento de la transacción se produce un corte de suministro de energía eléctrica y el servidor donde está instalada la BD no tiene una unidad de energía auxiliar. El corte de energía implica el cese de ejecución de la transacción. Si este corte se produce luego de escribir el nuevo saldo en la cuenta del cliente antes de reflejarlo en la cuenta del prestatario, la integridad de los datos se pierde (cuentacliente + cuentaprestatario no tiene el mismo resultado antes y después de ejecutar la transacción).

La Figura 17.1 representa gráficamente el problema (se supone un importe de factura a pagar de $150). Si se analiza la ejecución completa de la transacción, se puede observar cómo la BD queda en un estado de consistencia. Si el corte se produce en algún momento anterior a la escritura de la cuenta del prestatario, posterior a la escritura de la cuenta del usuario, la BD queda inconsistente. En la figura, se representa en tono gris el momento del corte de suministro de energía, y cómo afectó este corte a la BD.

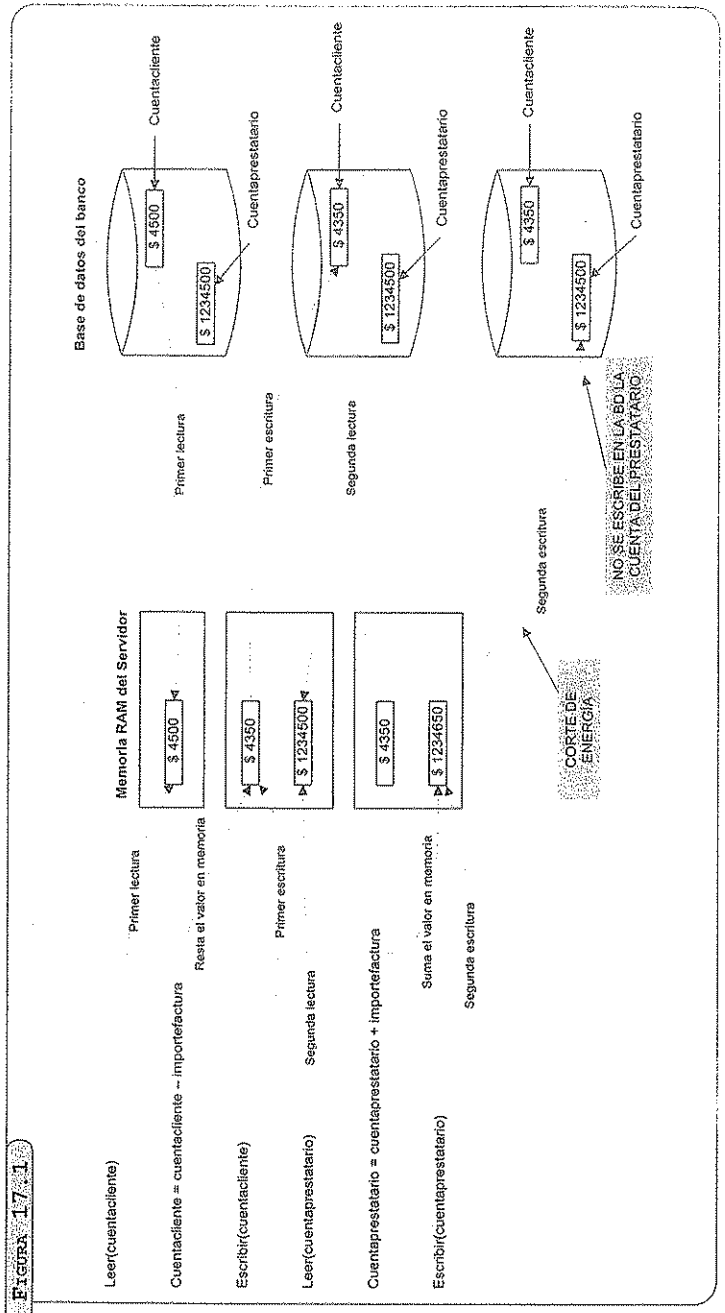


FIGURA 17.1

Para garantizar la atomicidad, hay dos acciones posibles: ejecutar toda la transacción o que no se lleve a cabo. Es responsabilidad del gestor del SGBD implementar los algoritmos necesarios para que se controle la atomicidad de una transacción.

La durabilidad es una propiedad que indica que los cambios llevados a cabo por una transacción quedan almacenados en la BD una vez que la transacción ha finalizado, sin la posibilidad de retrotraerlos.

Suponga, en el ejemplo del pago de un servicio, que dos usuarios intentan pagar la misma factura. Ambos generan sendas transacciones respectivas para efectuar el pago. Si ambas transacciones se realizan, provocan un doble pago de una misma factura, y esto significa que el segundo pago se realizó por error.

Si ambas transacciones finalizaron correctamente, no hay posibilidad de anular alguna de ellas. La condición de durabilidad así lo exige. En este caso, se deberá realizar una nueva transacción compensatoria que deshaga lo generado por una de las transacciones originales. Pero las transacciones finalizadas permanecen invariantes.

La última propiedad, el aislamiento, agrega un nuevo concepto. Aunque la ejecución de transacciones asegure consistencia y atomicidad, aún puede ocurrir que la BD quede inconsistente. Esto se debe a la condición de aislamiento. Para que una transacción mantenga la BD consistente, debe cumplirse que su ejecución no sea alterada por la ejecución de otra transacción que se ejecute en simultáneo.

La propiedad de aislamiento garantiza que una transacción se ejecute como única, es decir, que no pueda ser afectada por ninguna otra operación que se realice en simultáneo sobre la BD.

En el ejemplo de la Figura 17.1, se ejecuta solamente una transacción; por lo tanto, el aislamiento está garantizado. En el Capítulo 18, se analizarán con detalle las situaciones generadas por ejecución simultánea de transacciones, y cómo se solucionan esas situaciones.

Estados de las transacciones

Una transacción puede estar en alguno de los siguientes cinco estados:

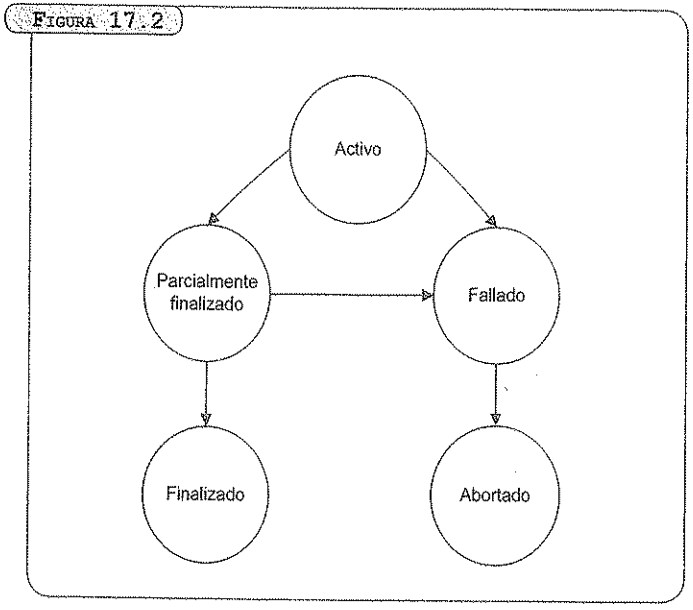
1. **Activo:** este estado se tiene desde que comienza su ejecución y se mantiene hasta completar la última instrucción, o hasta que se produzca un fallo.

2. **Parcialmente finalizado:** este estado, también conocido como parcialmente cometido, se alcanza en el momento posterior a que la transacción ejecuta su última instrucción.
3. **Finalizado:** este estado, también conocido como cometido, se obtiene cuando la transacción finalizó su ejecución, y sus acciones fueron almacenadas correctamente en memoria secundaria.
4. **Fallado:** a este estado se arriba cuando la transacción no puede continuar con su ejecución normal.
5. **Abortado:** este estado garantiza que una transacción fallada no ha producido ningún cambio en la BD, manteniendo la integridad de la información allí contenida.

La Figura 17.2 presenta el grafo de estados posibles para una transacción. Este grafo muestra los diferentes caminos por los cuales la ejecución de una transacción puede transcurrir.

Una transacción siempre comienza en estado activo. De dicho estado puede migrar al estado de parcialmente finalizada, si ejecuta la última instrucción sin error, o al estado de fallada, en caso contrario. Una transacción fallada debe abortarse. Esto significa que todos los cambios que esta pudiera haber efectuado sobre la BD deben deshacerse, para garantizar la consistencia. Una vez deshechos los cambios, el estado de una transacción cambia a estado abortado.

Figura 17.2



Desde el estado de parcialmente cometida existen dos caminos disponibles. El primero de ellos se presenta cuando la transacción puede almacenar en la BD los cambios producidos. En este caso, la transacción cumple su cometido y finaliza.

La segunda acción se produce cuando se intenta descargar a disco el *buffer* de la BD y se produce un fallo. En ese caso, la transacción parcialmente cometida no puede continuar su ejecución, y debe cambiar al estado de fallo para posteriormente abortar.

Para graficar la situación anterior, se presenta la Figura 17.3 (ver pág. 382). En ella, de modo similar a la Figura 17.1, se describe el momento en que la transacción alcanza el estado de parcialmente cometida. Allí la memoria principal contiene el *buffer* con el valor que debería quedar almacenado en la BD. El siguiente paso consiste en escribir el *buffer* en disco. En ese instante se genera un fallo y la escritura no puede llevarse a cabo. La transacción finalizó parcialmente bien, pero como no puede escribir en memoria secundaria el nuevo valor de cuentaprestatario, debe abortarse completamente para garantizar la integridad de la BD. En la BD, cuentacliente tiene un nuevo valor y cuentaprestatario tiene el viejo valor; por este motivo, debe abortarse la transacción retrotrayendo el cambio efectuado en cuentacliente.

Hasta el momento se presentó la forma teórica de enfrentar un fallo. Resta aún discutir cómo el gestor del SGBD lleva a cabo las acciones que garantizan el cumplimiento de las cuatro propiedades de las transacciones. Cada SGBD implementa una serie de algoritmos que actúan para garantizar la integridad y la consistencia de la información.

Cualquier transacción que falla debe abortarse, para mantener la BD consistente. Sin embargo, suponga la siguiente situación: un usuario requiere pagar un servicio, genera el pago y este no puede llevarse a cabo debido a un fallo producido. Por ende, la transacción generada falla y aborta, impidiendo que el usuario realice el pago. La pregunta en este punto es: ¿qué hacer con la transacción abortada? Las opciones son dos:

1. **Reiniciar la transacción:** esta acción se puede llevar a cabo cuando el error se produjo por el entorno y no por la transacción. Si se interrumpió el suministro de energía, se perdió el enlace de comunicaciones o el servidor de la BD no respondió al requerimiento del usuario, la transacción falló y abortó. En estos casos, para el usuario es posible volver a intentar realizar el pago. Se genera una nueva transacción. La transacción anterior que falló ya terminó, su estado es abortada y por la propiedad de durabilidad no puede reiniciarse.

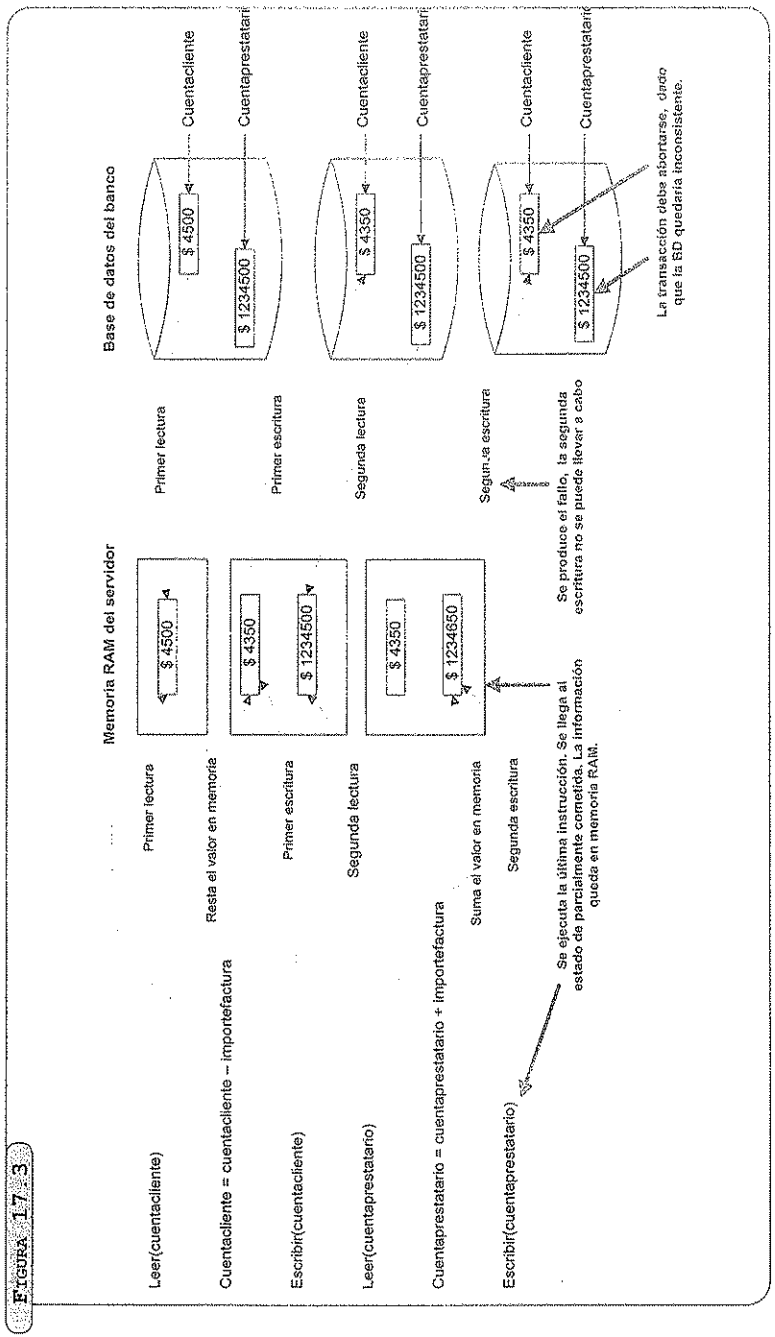


FIGURA 17.3

2. **Cancelar definitivamente la transacción:** esta acción se lleva a cabo cuando se detecta un error interno en la lógica de la transacción. Este error solamente puede ser salvado si la transacción es redefinida.

Entornos de las transacciones

Las transacciones pueden ser analizadas desde tres perspectivas básicas.

La primera de ellas consiste en presentar el problema desde una concepción antigua pero sencilla: analizar el comportamiento de las transacciones en entornos monousuarios. Estos entornos tienen por característica básica que admiten un solo usuario operando con la BD, y a este usuario solamente se le permite realizar una actividad por vez. Este tipo de entornos es muy poco realista en la actualidad; sin embargo, es útil para presentar el problema de manera gradual.

De las cuatro propiedades que caracterizan a una transacción, la propiedad de aislamiento se obtiene por definición. Si solo se puede ejecutar una transacción por vez, no hay forma de que la ejecución de la transacción t0 afecte la ejecución de la transacción t1, dado que solo una puede estar activa.

La propiedad de consistencia es responsabilidad del programador; por lo tanto, para el análisis de integridad de la BD, se asume que la transacción está bien escrita.

La propiedad de durabilidad indica que una transacción cometida no puede ser retrotraída. Esta condición se presentará cuando se analicen los algoritmos de tratamientos de fallos.

Por último, se debe considerar la propiedad de atomicidad. En el resto del capítulo, se presenta su tratamiento en entornos monousuarios.

El Capítulo 18 analiza las transacciones bajo entornos concurrentes. Estos entornos admiten varios usuarios simultáneos, cada uno de los cuales puede generar múltiples transacciones a la vez. En ese caso, el énfasis quedará sobre la propiedad de aislamiento. El comportamiento de las otras tres propiedades –atomicidad, consistencia y durabilidad– es similar tanto para entornos monousuarios como concurrentes.

El último escenario para las transacciones se presenta con los entornos distribuidos. Estos entornos son los más comunes en la práctica actual de BD, y el estudio de las transacciones se torna más complejo.

Una transacción generada por un usuario puede requerir modificar una BD residente en varios servidores. Los tipos de fallos que se pueden producir son mayores. Asegurar la integridad de una BD distribuida físicamente en varias computadoras es mucho más complejo. No obstante, no es propósito de este material profundizar el tratamiento de las transacciones distribuidas.

Fallos

Antes de continuar con el tratamiento de las transacciones y presentar los métodos de recuperación, es importante analizar los tipos de fallos que se pueden producir cuando se opera con una computadora.

Una computadora es un dispositivo propenso a fallos. Estos fallos pueden ser de diversos tipos: rotura de disco o fuente de alimentación, errores de *software* (por ejemplo, que deje de funcionar el SO o que falle el SGBD), etcétera. Además, podría ocurrir que una catástrofe (terremoto, incendio, inundación) afecte el lugar físico donde está la computadora, o que sencillamente un sabotaje impida el funcionamiento del sistema, y en particular que una transacción no pueda continuar su ejecución.

Si bien los tipos de fallos que se pueden producir son muy variados, y alguno de ocurrencia es muy poco probable, son situaciones que deben ser consideradas a fin de asegurar la integridad de los datos.

Los protocolos de trabajo tienen por objetivo asegurar la integridad y la consistencia de la BD, ante cualquier situación que lleve a no cumplir alguna de las cuatro propiedades definidas para una transacción.

Los fallos pueden ser clasificados en dos tipos:

1. **Fallos que no producen pérdida de información:** si una transacción que solamente está consultando la BD falla y debe abortarse, no produce pérdida de información. Una transacción de consulta no modifica, borra ni agrega datos en la BD; por lo tanto, si se produjese un fallo en su ejecución, no se alteraría el contenido de la BD.
2. **Fallos que producen pérdida de información:** involucran transacciones que incluyen las cláusulas SQL: INSERT, DELETE o UPDATE. Estas transacciones afectan el contenido de la BD y debe asegurarse que la integridad de los datos se mantenga.

Los fallos que producen pérdida de información son los más complejos de tratar, y los métodos de recuperación de información –que se

discuten en el apartado siguiente– tienen por objetivo salvar las situaciones de pérdida de integridad generadas por estos fallos.

Se puede advertir que algunos de los ejemplos de fallos son de baja probabilidad de ocurrencia en los entornos informáticos actuales. Por ejemplo, la pérdida de energía eléctrica es salvada con **UPS (Uninterruptible Power Supply**, por sus siglas en inglés, **o Sistema de Alimentación Ininterrumpida)** o grupos electrógenos. La rotura de disco rígido también puede ocurrir, pero para asegurar un sistema estable, es probable que se disponga de al menos dos discos trabajando en espejo. Esto es, si un disco se rompe, el segundo disco (que contiene exactamente la misma información) entra en operación de manera automática, y esto es directamente controlado por el *hardware* de la computadora.

Sin embargo, los fallos se pueden producir: la **CPU (Central Processing Unit**, por sus siglas en inglés, **o Unidad de Procesamiento Central)** de la computadora puede fallar, el SO puede dejar de funcionar, el SGBD puede interrumpir su ejecución por errores internos, etc., y estas situaciones pueden llevar a la pérdida de consistencia de la BD.

Por lo tanto, se necesitan métodos de recuperación de integridad de la BD. El siguiente apartado aborda este tema.

Métodos de recuperación de integridad de una base de datos

El problema que se genera ante un fallo es la posible pérdida de consistencia en la BD. Esto se debe a que una transacción puede haberse ejecutado parcialmente sobre la BD.

El primer concepto a tener en cuenta está relacionado con el tipo de almacenamiento que utiliza una transacción: almacenamiento volátil (RAM) y almacenamiento no volátil (disco rígido). Como se presentó en la Figura 17.3, si el fallo se produce antes de escribir en disco, la BD queda inconsistente.

Los métodos de recuperación de integridad de la BD ponen especial énfasis en tratar la información contenida en memoria RAM y disco rígido. Es importante que se comprenda que cualquier tipo de fallo producido afectará a la información contenida en RAM, debido a que esta memoria es volátil. Sin embargo, salvo la rotura o robo de disco rígido, los fallos no afectan a los datos contenidos en memoria no volátil.

Acciones para asegurar la integridad de los datos. Atomicidad

Ante un fallo de una transacción, ¿cuáles son las acciones que se deberían llevar a cabo para asegurar la integridad de la BD? Una alternativa es volver a ejecutar completamente la transacción fallida.

Suponga que la transacción t0, que se ejecuta para el pago de una factura de un servicio, falla. La BD queda en el estado de inconsistencia presentado en la Figura 17.3. Se genera una nueva transacción, t1, similar a t0, para asentar el pago. Dicha transacción se ejecuta sin fallos. ¿Cómo queda la BD? Nuevamente queda inconsistente, debido a que la ejecución correcta de t1 comenzó desde un estado de inconsistencia. Por lo tanto, volver a ejecutar una transacción que anteriormente falló no retrotrae la BD a un estado de consistencia.

La alternativa a ejecutar nuevamente la transacción es no hacer nada, pero en ese caso, la BD permanecerá inconsistente.

Como conclusión, reejecutar o no una transacción fallida no asegura la consistencia de la BD. El motivo de la inconsistencia es que la transacción efectúa cambios sin la seguridad de finalizar correctamente. Para evitar los problemas que puedan surgir, es necesario realizar acciones que permitan retrotraer el estado de la BD al instante anterior al comienzo de ejecución de la transacción. Solamente se deben registrar los cambios en la BD ante la seguridad de que la transacción no va a fallar o, en su defecto, se deben tomar todas las acciones preventivas posibles para que, ante un fallo, se pueda mantener la consistencia.

Estas acciones preventivas tienen que ver con dejar constancia de las actividades llevadas a cabo por la transacción, que permitirán deshacer los cambios producidos.

Existen dos métodos de recuperación: el de bitácora (log) y el de doble paginación. Cualquiera de estos métodos de recuperación necesita de dos algoritmos básicos:

1. Algoritmos que se ejecutan durante el procesamiento normal de las transacciones, que realizan acciones que permiten recuperar el estado de consistencia de la BD ante un fallo.
2. Algoritmos que se activan luego de detectarse un error en el procesamiento de las transacciones, que permiten recuperar el estado de consistencia de la BD.

En los apartados siguientes, se explican con detalle los métodos más difundidos.

Bitácora

El método de bitácora (log) o registro histórico plantea que todas las acciones llevadas a cabo sobre la BD deben quedar registradas en un archivo histórico de movimientos.

Para la Real Academia Española, una bitácora es un "libro pequeño o conjunto de papel en el que se lleva la cuenta y razón, o en el que se escriben algunas noticias, ordenanzas o instrucciones". Por lo tanto, una bitácora es un repositorio donde se registran acciones.

El método de bitácora consiste en registrar, en un archivo auxiliar y externo a la BD, todos los movimientos producidos por las transacciones **antes** de producir los cambios sobre la BD misma. De esta forma, en caso de producirse un error, se tiene constancia de todos los movimientos efectuados. Con este registro de información, es posible garantizar que el estado de consistencia de la BD es alcanzable en todo momento, aun ante un fallo.

La estructura del archivo de bitácora es simple. Cada transacción debe indicar su comienzo, su finalización y cada una de las acciones llevadas a cabo sobre la BD que modifiquen su contenido. El gestor del SGBD es el responsable de controlar la ejecución de las transacciones, de administrar el archivo de bitácora y de utilizar los algoritmos de recuperación cuando el SGBD se recupera de una situación de fallo.

Para poder cumplir con su cometido, el gestor identifica cada transacción asignándole un número correlativo único a cada una de ellas. El formato de una transacción en bitácora es el siguiente:

<T0 comienza>
<T0, dato**1**, valor viejo, valor nuevo>
<T0, dato**2**, valor viejo, valor nuevo>

<T0, dato**n**, valor viejo, valor nuevo>

.....
<T0 finaliza>

La primera entrada en bitácora para cada transacción será Ti **comienza**, indicando que la transacción Ti inicia su ejecución. Cada operación de escritura sobre la BD deberá quedar indicada con el formato

<T0, dato**i**, valor viejo, valor nuevo>

el cual indica a su vez que la transacción **T0** modifica el *i*-ésimo dato de la BD. **Valor viejo** registra el valor anterior que tenía el *i*-ésimo dato, y **valor nuevo** registra el valor actualizado que se le asigna a ese dato.

Cuando la transacción T0 termina su ejecución sin fallos, en bitácora se agrega una instrucción, T0 finaliza.

Podría ocurrir que la transacción fallara; en ese caso, se procederá a abortarla. Una vez abortada la transacción, en bitácora debe agregarse una entrada, <T0 aborta>.

Según la naturaleza del fallo, puede ocurrir que en el registro histórico para una transacción se escriba un **comienza**, algunos cambios en la BD, y luego no se registre la entrada Ti **finaliza** o Ti **aborta**. Suponga que el fallo producido se debe a un corte abrupto del suministro de energía. En este caso, no es posible agregar nada en la bitácora, con lo cual la transacción solo pudo registrar su comienzo y algunas operaciones de modificación.

Para el método de bitácora, una transacción que registra un **comienza** y no tiene registro de fin es una transacción que debe abortarse.

A continuación, se describe cómo debe quedar en bitácora la transacción que efectúa el pago de un servicio:

<Ti comienza>
<Ti, cuentacliente, 4500, 4350>
<Ti, cuentaprestatario, 1234500, 1234650>
<Ti finaliza>

Solamente se registran en bitácora las instrucciones que modifican el contenido de la BD; por ende, solo se registran las operaciones de escritura.

Los algoritmos relacionados con el proceso de bitácora deben garantizar que primero se graben los *buffers* correspondientes al archivo de bitácora en memoria secundaria, antes de grabar en disco los *buffers* de la BD. Si esta condición no se respeta, nuevamente se corre el riesgo de perder la integridad de la BD.

El manejo de los *buffers* de archivo es responsabilidad del SO. Si este tuviera la necesidad de escribir en disco el *buffer* de la BD, el gestor de BD deberá operar contra el SO, indicando que primero debe escribir el *buffer* correspondiente al registro histórico (bitácora), aun si esto no fuese necesario en ese momento.

Durante el procesamiento normal de las transacciones, se genera el registro en bitácora y luego se actualiza la BD.

Para todas las transacciones que alcanzan el estado de finalizada, el trabajo registrado en bitácora es innecesario. Solamente cuando se produce un error, se deben activar los algoritmos que subsanan dicho error, utilizando para ello el registro histórico.

La bitácora presenta dos alternativas para implementar el método de recuperación: modificación diferida de la BD y modificación inmediata de la BD. Se describe a continuación cada una de ellas.

Modificación diferida de una base de datos

El método de bitácora con modificación diferida de la BD demora todas las escrituras en disco de las actualizaciones de la BD, hasta que la transacción alcance el estado de finalizada en bitácora. Así, mientras la transacción está en estado activo, se demora cualquier escritura física en la BD.

Esta variante posee una ventaja interesante. Si una transacción activa falla, se puede asegurar que los cambios producidos por la transacción no tuvieron impacto sobre la BD. Por lo tanto, el fallo de la transacción es ignorado y la BD permanece íntegra.

La Figura 17.4 (ver pág. 390) ilustra esta situación. Se utiliza para ella la misma transacción presentada en figuras anteriores. Los cambios en la BD se demoran y recién se escriben en disco cuando la transacción finaliza. En la figura se produce un fallo; por lo tanto, la BD permanece inalterada.

Para la modificación diferida de la BD, no es necesario guardar en bitácora el valor viejo del dato. La transacción solamente debe registrar

<Ti, daton, valor nuevo>

Esto se debe a que no es posible modificar la BD hasta que la transacción alcance el estado de finalizada.

Existe una condición bajo la cual la BD puede perder la integridad. Suponga que la transacción alcanza el estado de parcialmente cometida. Esto significa que en bitácora se han registrado las siguientes operaciones:

<Ti comienza>
<Ti, cuentacliente, 4500, 4350>
<Ti, cuentaprestatario, 1234500, 1234650>

La BD aún no ha sufrido cambio alguno.

El paso siguiente consiste en grabar en el registro histórico la entrada

<Ti finaliza>

La transacción alcanza entonces el estado de finalizada, y la actualización de la BD puede comenzar. Se actualiza el estado de cuentacliente y en ese momento se produce un fallo. La BD queda inconsistente debido a que cuentaprestatario aún no fue actualizado, como muestra la Figura 17.5 (ver pág. 391).

FIGURA 17.4

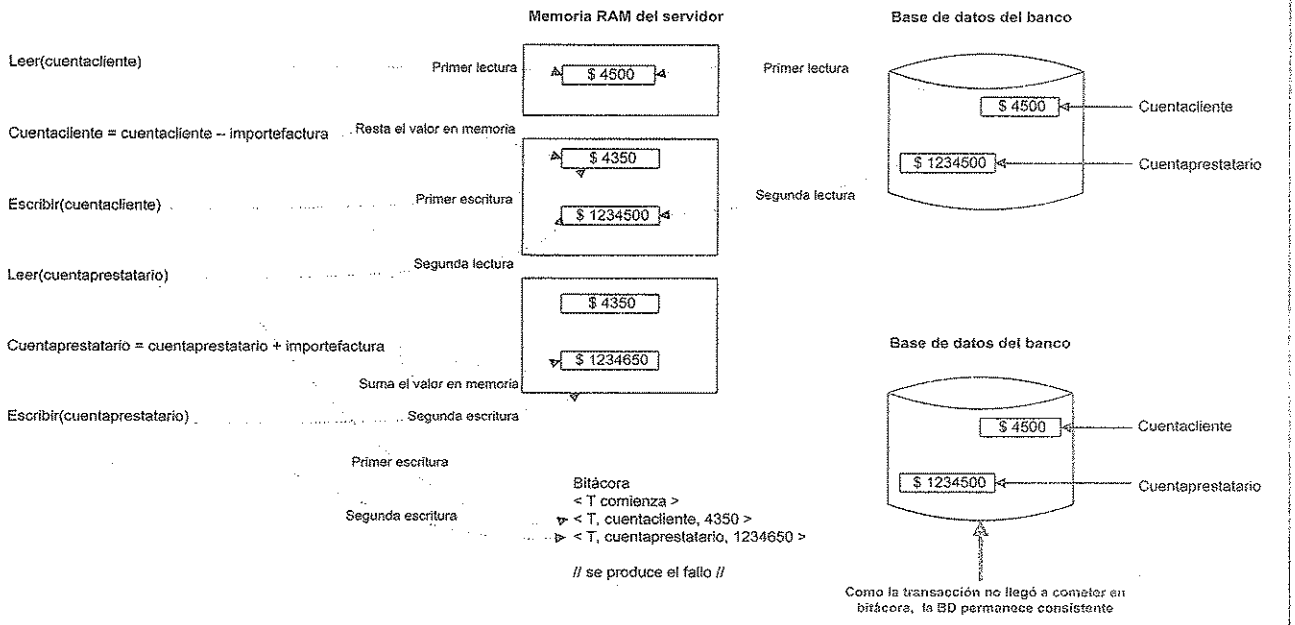
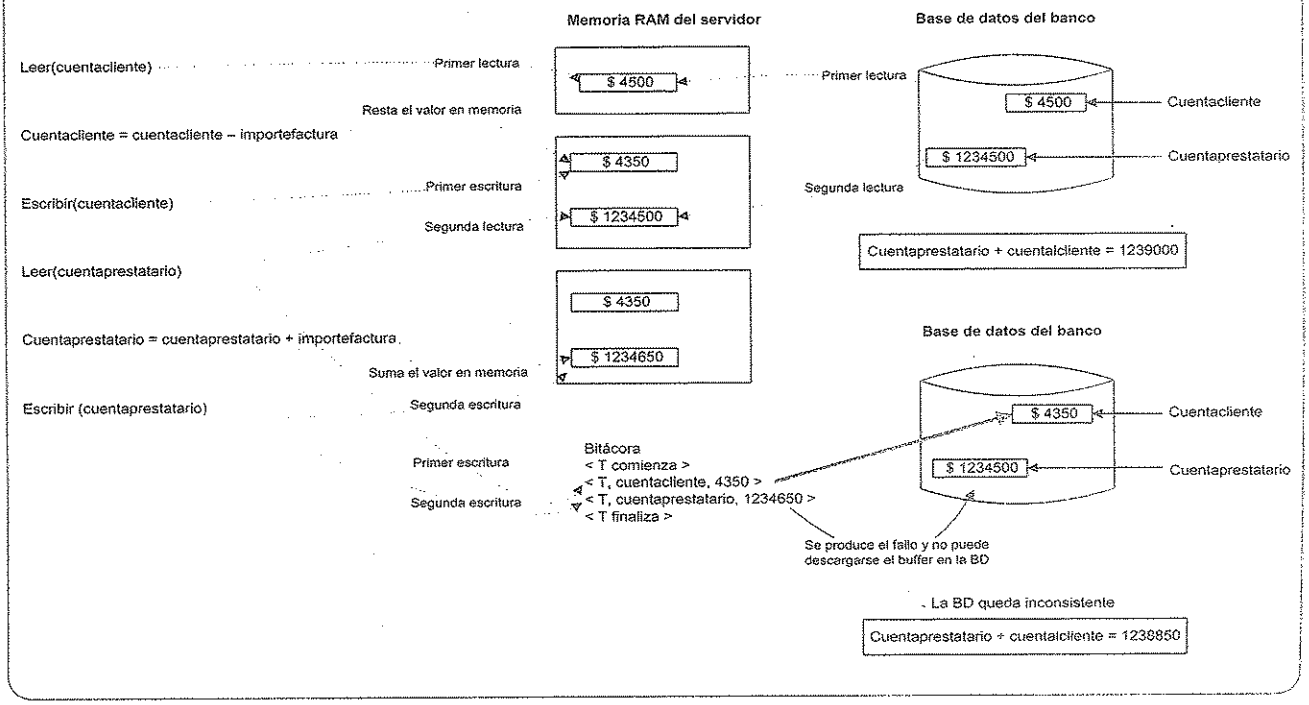


FIGURA 17.5



El problema se genera porque si bien para la bitácora la transacción terminó (alcanzó el estado de finalizada), no se terminaron de grabar en el disco los datos de la BD.

Cuando el sistema se recupera del fallo, actúa el segundo algoritmo que permite recuperar la integridad de la BD. Este algoritmo revisa la bitácora. Toda transacción que tenga registrado un <Ti comienza> y no tenga un <Ti finaliza> será ignorada.

Una transacción que no finalizó no pudo haber realizado cambios en la BD.

Por el contrario, una transacción que tiene registrado un <Ti finaliza> puede haber ejecutado parcialmente cambios en la BD, y como lo muestra el ejemplo de la Figura 17.5, haber dejado la BD en estado de inconsistencia.

El algoritmo de recuperación debe ejecutar alguna funcionalidad que salve esta situación. En bitácora están registradas todas las operaciones realizadas por la transacción; por ende, se puede reejecutar la transacción y dejar la BD consistente.

Considerando nuevamente el ejemplo antes descripto, la transacción falla y solamente se escribió en disco el nuevo valor de cuentacliente. Es necesario reejecutar la transacción para que también se modifique cuentaprestatario. La técnica de modificación diferida de la BD activa un algoritmo, denominado REHACER, para todas aquellas transacciones que en bitácora tengan registrado un <T finaliza>.

Existe una última situación que es interesante para analizar. Suponga que la transacción T2 alcanza el estado de finalizada y así lo registra en bitácora. Luego, actualiza la BD y tiene éxito en esa operación. Una vez finalizada la actualización, se produce un fallo. Cuando el SGBD se recupera, el algoritmo de recuperación detecta que T2 tiene en el registro histórico un <T2 comienza> y un <T2 finaliza>, y por consiguiente ejecuta el algoritmo REHACER(T2).

Se puede notar que la ejecución de este algoritmo resulta innecesaria porque la BD "estaba bien"; el fallo se produjo luego que T2 finalizó su actualización. Sin embargo, el algoritmo de recuperación no tiene forma de detectar esta situación; la transacción T2 se reejecuta, dejando la BD nuevamente en estado de consistencia.

Las transacciones tienen una condición llamada de idempotencia. Esta condición asegura que aunque se reejecute *n* veces una transacción, se genera siempre el mismo resultado sobre la BD.

La condición de idempotencia es fundamental en el siguiente caso. Suponga que se genera un fallo y durante la recuperación se decide reejecutar con REHACER la transacción Ti. Mientras se está reejecutando Ti, se produce un nuevo fallo. Nuevamente, cuando se recupera del fallo se deberá ejecutar otro REHACER. Es fundamental, entonces, que ante cada REHACER, el resultado sobre la BD sea siempre el mismo.

Modificación inmediata de una base de datos

Las características propias de modificación inmediata de una BD hacen que el método resulte muy interesante. No obstante, plantea alguna situación de desventaja.

Suponga que una transacción durante su ejecución modifica varios valores de la BD. Los cambios sobre la BD se aplazan hasta que la transacción finaliza, y en ese momento se deben escribir a disco. Esta acción significa una importante sobrecarga de trabajo cada vez que la transacción finaliza.

Si, por el contrario, los cambios en la BD se efectúan a medida que la transacción avanza en su ejecución, la sobrecarga de trabajo tiende a desaparecer, distribuyéndose en el tiempo.

La modificación inmediata de la BD plantea actuar de esta forma. Los cambios en la BD se producen a medida que se producen en la transacción que se está ejecutando, siempre bajo la consigna que indica que un cambio se registra primero en bitácora para luego grabarse en la BD.

Esta pequeña modificación en el proceso altera el análisis de fallos realizado. Ahora, puede suceder que el fallo de una transacción ocurra y que la BD esté parcialmente modificada. Suponga que en bitácora se registró

<Ti comienza>
<Ti, cuentacliente, 4500, 4350>

y en ese momento se produjo un fallo en la ejecución de Ti. Con el algoritmo de modificación diferida, se tenía la certeza de que la BD no había sido alterada, y permanecía consistente.

Con el algoritmo de modificación inmediata, puede ocurrir que cuentacliente se haya modificado en la BD, y que se presente una situación de inconsistencia. La Figura 17.6 (ver pág. 394) muestra el caso.

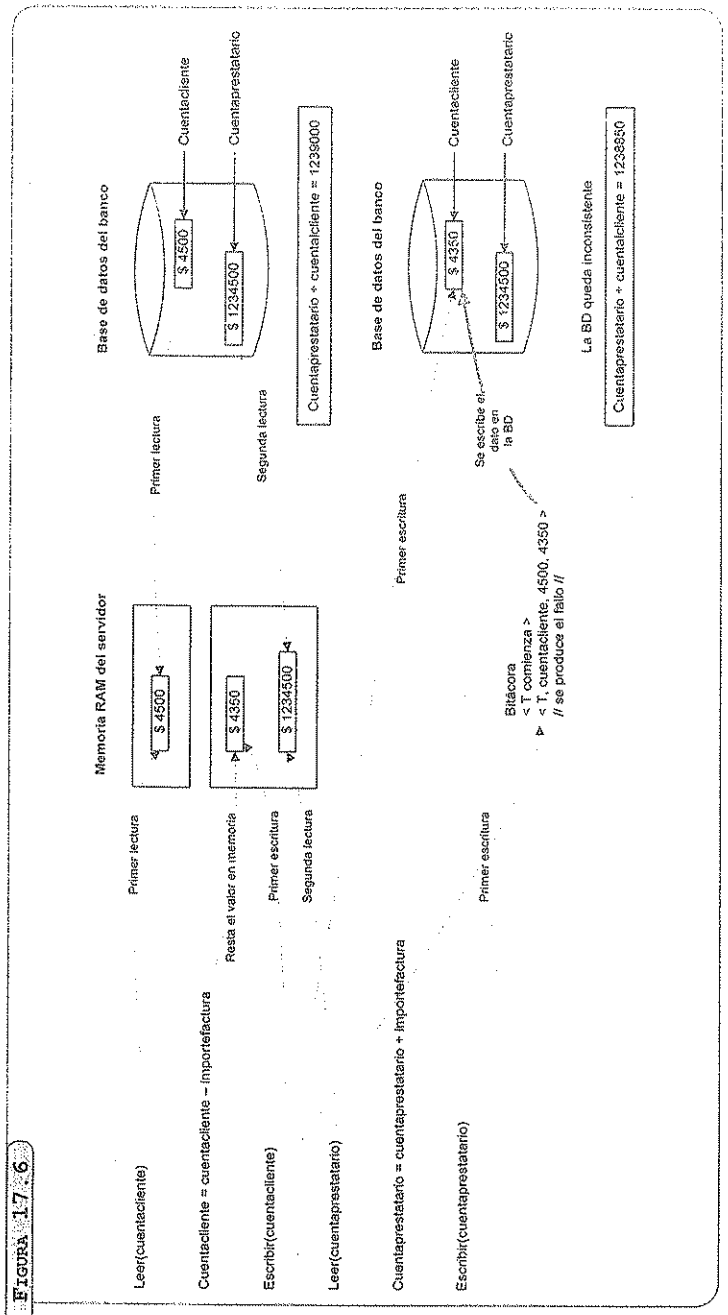


FIGURA 17.6

El algoritmo de recuperación en caso de fallo debe actuar de manera diferente. La BD quedó en estado de inconsistencia, y la transacción no puede reejecutarse (REHACER) porque no alcanzó el estado de finalizada.

En este caso, es necesario contar con otra operación, DESHACER, cuyo efecto será retrotraer la BD al estado que tenía antes de comenzar la transacción. Para lograr este propósito, la función DESHACER vuelve a la BD los valores viejos de cada dato modificado por Ti registrados en la bitácora.

El ejemplo planteado para modificación diferida de una BD puede ocurrir nuevamente. Esto es, la transacción T puede fallar luego de haber registrado en bitácora un <T finaliza>, habiendo registrado parcialmente los cambios en la BD: Será necesario utilizar la función REHACER.

De esta forma, el método de bitácora con modificación inmediata de una BD requiere dos funciones de recuperación:

1. Rehacer (T).

2. Deshacer (T).

En resumen, si se produce un error, se debe REHACER toda transacción de la bitácora que haya finalizado, y se debe DESHACER toda transacción iniciada que no haya finalizado.

Puntos de verificación del registro histórico

El método de bitácora requiere una última consideración. Suponga que durante un determinado tiempo no se han producido incidentes en la operación con la BD. Básicamente, las transacciones ejecutadas durante los últimos tres meses finalizaron sin problemas. Toda esta información quedó registrada en bitácora. Luego, la transacción Ti comienza su ejecución y falla. Sin importar si el método de recuperación utilizado es con modificación inmediata o diferida de la BD, deberán rehacerse todas las transacciones que tengan registrados <T comienza> y <T finaliza>.

Es altamente probable que un gran porcentaje de las transacciones hayan finalizado bien, dejando la BD consistente. Rehacer todo ese trabajo es muy lento e innecesario.

Con el fin de evitar la revisión de toda la bitácora ante un fallo, el gestor de BD agrega en forma periódica, al registro histórico, una entrada

<punto de verificación>

Estos puntos de verificación se agregan cuando se tiene la seguridad de que la BD está en estado consistente. La finalidad es indicar que, ante un fallo, solo debe revisarse la bitácora desde el punto de verificación en adelante, dado que las transacciones anteriores finalizaron correctamente sobre la BD.

En entornos monousuarios siempre es posible colocar el punto de verificación en un instante del tiempo sin transacciones activas. Esta situación resulta un tanto más compleja para entornos concurrentes, y será abordada en el Capítulo 18.

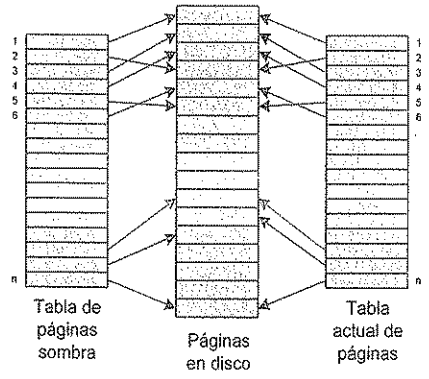
Por último, se debe analizar la periodicidad de los puntos de verificación. Colocarlos "muy cerca" significa tener que rehacer poco trabajo en el caso de producirse un fallo, pero agrega una sobrecarga a la escritura de la bitácora de la BD. Por el contrario, puntos de verificación "lejanos" entre sí significan menos sobrecarga de escrituras en bitácora, pero generan la necesidad de rehacer mayor cantidad de trabajo ante un fallo.

Doble paginación

El método alternativo al de bitácora recibe el nombre de doble paginación o paginación en la sombra. Este método plantea dividir al BD en nodos virtuales (páginas) que contienen determinados datos.

Se generan dos tablas en disco y cada una de las tablas direcciona a los nodos (páginas) generados. La Figura 17.7 presenta esquemáticamente el concepto general de doble paginación.

Figura 17.7



El método trabaja de la siguiente forma. Ante una transacción que realiza una operación de escritura sobre la BD, la secuencia de pasos es la siguiente:

1. Se obtiene desde la BD el nodo (página) sobre el cual debe realizarse la escritura (si el nodo está en memoria principal, este paso se obvia).
2. Se modifica el dato en memoria principal y se graba en disco, en un nuevo nodo, la página actual que referencia al nuevo nodo.
3. Si la operación finaliza correctamente, se modifica la referencia de la página a la sombra para que apunte al nuevo nodo.
4. Se libera el nodo viejo.

En caso de producirse un fallo, luego de la recuperación, se desecha la página actual y se toma como válida la página a la sombra.

Considere que si las páginas referencian nodos diferentes, se tiene la seguridad de que los datos del nodo viejo (apuntado por la página a la sombra) son correctos. No es posible garantizar que la escritura del nodo nuevo fue exitosa.

Este método presenta dos ventajas evidentes sobre el método de bitácora: se elimina la sobrecarga producida por las escrituras al registro histórico, y la recuperación ante un error es mucho más rápida (se recupera la página a la sombra como página actual, descartando la página actual insegura).

Sin embargo, el método también tiene algunas desventajas importantes. Primeramente, exige dividir la BD en nodos o páginas, y esta tarea puede ser compleja. En segundo lugar, la sobrecarga del método aparece en entornos concurrentes. Suponga que la transacción T1 modifica un dato del nodo 2 y, por lo tanto, comienza la escritura de este nodo en una nueva dirección. Además, T2 necesita modificar un dato diferente de T1, pero también contenido en el nodo 2. Esta modificación debería demorarse, o en su defecto, se tendrían muchas copias del mismo nodo.

Por último, el método puede generar varios nodos ocupados no referenciados. En caso de fallo en una transacción, una vez recuperada la situación de fallo, el nodo nuevo no se utiliza; sin embargo, sigue ocupando lugar en el disco. Para salvar estas situaciones, es necesario definir algoritmos denominados recolectores de basura (*garbage collectors*), que detecten estos nodos y liberen el espacio ocupado no utilizado.

1. Explique el concepto de transacción. ¿Por qué las transacciones son unidades de operación tan importantes en un SGBD?
2. ¿Por qué una transacción debe considerarse una unidad lógica de trabajo?
3. ¿Cómo se aplica el concepto de aislamiento de una transacción en un entorno monousuario?
4. ¿Cómo proceder ante una transacción que no cumpla la propiedad de consistencia?
5. Compare los esquemas de recuperación basados en bitácora con modificación inmediata y modificación diferida, en términos de facilidad de implementación y costo de procesamiento.
6. Si se utiliza modificación inmediata de la BD, muestre mediante un ejemplo cómo podría llegarse a un estado de inconsistencia de datos, si los registros de bitácora para una transacción no se graban en almacenamiento no volátil antes de finalizar esa transacción.
7. Explique cómo el *buffering* puede causar que la BD llegue a ser inconsistente, si alguno de los registros de bitácora pertenecientes a un bloque no se graba en almacenamiento no volátil antes de que el bloque se grabe en disco.
8. ¿La función REHACER para el método de bitácora es igual ya sea modificación inmediata o diferida? ¿Por qué motivo?
9. Explique el propósito de los mecanismos de punto de verificación, y la frecuencia con que deben aparecer.
10. ¿Cuáles son las ventajas del método de paginación en la sombra o doble paginación?
11. ¿El método de doble paginación es más eficiente que el método de bitácora? ¿En qué casos genéricos utilizaría cada uno?

1. Suponga que una transacción, T, se está ejecutando y se produce un error durante dicho procesamiento. El fallo se produce mientras T está activa. Explique qué debe hacerse luego de la recuperación del fallo si:
   a. Se utilizaba bitácora con actualización inmediata de la BD.
   b. Se utilizaba bitácora con actualización diferida de la BD.
   c. Se utilizaba doble paginación.
2. Suponga que una transacción, T, se está ejecutando y se produce un error durante el procesamiento. Luego de recuperarse del fallo, comienzan las acciones correctivas. ¿Qué sucede si, durante ese proceso, el sistema vuelve a fallar?
3. Suponga que una transacción, T, se está ejecutando y genera un error en el sistema al intentar dividir por cero. Luego de la intervención del administrador del sistema, el SGBD se recupera del error. ¿Cómo debe procederse con la transacción T?
4. Sea T1 la siguiente transacción en bitácora:

   <T1 Comienza>
   <T1, dato1, 3, 4>
   <T1, dato2, 34, 35>
   <T1 Finaliza>

   Indique:

   a Si la transacción T1 utiliza actualización inmediata o diferida de la BD.
   b. Cómo quedaría T1 si se tratase de una actualización diferente del caso anterior.
   c. En qué estado está la transacción T1.
5. Suponga que una transacción, T1, alcanzó el estado de parcialmente cometida. Luego, por algún inconveniente, falla. ¿Es posible que T1 vuelva al estado de activa? Justifique adecuadamente.

CAPÍTULO 18

Transacciones en entornos concurrentes

Objetivo

La motivación del Capítulo 17 fue mantener la BD consistente e íntegra mientras los usuarios acceden, agregan, borran o modifican los datos de la BD. Para ello, fue necesario introducir el concepto de transacción.

Las transacciones, al actuar como unidad lógica de trabajo, garantizan atomicidad, consistencia, aislamiento y durabilidad. Sin embargo, el Capítulo 17 alcanzó una de estas propiedades por definición del entorno de trabajo. La propiedad de aislamiento en entornos monousuarios se obtiene automáticamente, y no permite que más de una transacción esté activa en cada momento.

Sin embargo, los SI actuales no están concebidos para que solo un usuario a la vez opere contra ellos. Los entornos de trabajo, concurrentes o distribuidos, permiten que varios usuarios ejecuten simultáneamente consultas, inserciones, bajas o modificaciones sobre la BD. Por este motivo, se deben considerar con detalle los aspectos relacionados con la propiedad de aislamiento de las transacciones.

En este capítulo, se presenta un estudio detallado de transacciones en entornos concurrentes. Se analizan las nuevas características de trabajo, y cómo estas características afectan el procesamiento de las transacciones. Básicamente, se discuten la propiedad de aislamiento y los algoritmos necesarios para que dicha propiedad se cumpla.

Posteriormente, se discute el método de bitácora para lograr atomicidad en las transacciones. En general y como se verá, el método no muestra grandes variantes.

En el presente capítulo, se utilizan muchos conceptos de concurrencia, los cuales no son explicados con detalle, ya que tienen las mismas características que la concurrencia que necesita un SO para implementar su operatoria. Desde la perspectiva de BD, el control de concurrencia puede implementarse basándose en cualquier variante que utiliza el SO.

Ejecución concurrente de transacciones

Los SGBD, a través de los gestores de BD, permiten que varias transacciones operen simultáneamente sobre la BD, situación que puede generar problemas de consistencia.

Si bien dos transacciones, T1 y T2, son consistentes ejecutadas en entornos monousuarios, si la ejecución de estas se produce en simultáneo, puede alterarse la consistencia de la BD. Este problema se genera debido a que un entorno concurrente debe asegurar, además, la propiedad de aislamiento de una transacción. Esto significa que las dos transacciones T1 y T2 se interfieren en su ejecución.

Una solución simple para el problema consiste en secuencializar la ejecución de las transacciones. Suponga que T1 y T2 llegan en simultáneo a ejecutarse sobre el SGBD. Para evitar problemas de integridad de la BD y sabiendo que la condición de aislamiento es fundamental, el gestor de BD decide procesar primero la transacción T1, y cuando esta finaliza, procesar T2. De esta manera, el gestor de BD actúa como un cuello de botella que asegura la ejecución monousuaria de las transacciones BD.

Si bien la solución anterior es factible, suponga que en un entorno concurrente, un banco por ejemplo, existen 1.000 usuarios accediendo simultáneamente a la BD. Estos usuarios generan 1.000 transacciones, que demoran en promedio un segundo en ser atendidas. Como el gestor las procesa "en secuencia", la última transacción recibida será atendida luego de 1.000 segundos de espera; esto significa aproximadamente 15 minutos después de generada.

Suponga que está frente a un cajero automático y debe esperar ese tiempo, ¿haría la espera? La cola de atención generada detrás de un cajero automático (o humano) sería enorme. De hecho, se debería plantear la posibilidad de no usar computadoras, ¡ya que serían demasiado lentas!

Un entorno concurrente se genera para explotar la posibilidad que nos dan las computadoras de atender simultáneamente varios procesos.

Para lograr este objetivo, hay determinadas reglas que deben cumplirse, pero la atención en simultáneo es una característica que debe priorizarse. Si se pueden atender las 1.000 transacciones a la vez, el tiempo de espera podrá reducirse a solo algunos segundos.

Se puede notar que hasta el momento no se ha mencionado la palabra "fallos". Un entorno concurrente puede generar situaciones de pérdida de integridad de los datos, aun sin producirse fallos en el procesamiento de las transacciones.

A lo largo del capítulo, se analizan problemas de pérdida de consistencia en los datos, a partir del procesamiento de transacciones sin que estas fallen. Una vez discutidos los problemas y presentadas las soluciones, se incorporará, al estudio de los entornos concurrentes, la posibilidad de fallo en el procesamiento transaccional.

Para comprender los problemas que pueden surgir en un entorno concurrente, suponga que en un banco se realizan dos transacciones simultáneas, T1 y T2.

T1	T2
leer(cuentaA)	leer(cuentaA)
cuentaA = cuenta – 1000;	descuento = cuentaA * 0.2
escribir (cuentaA)	cuentaA = cuentaA – descuento;
leer (cuentaB)	escribir (cuentaA)
cuentaB = cuentaB + 1000;	leer (cuentaB)
escribir(cuentaB)	cuentaB = cuentaB + descuento;
	escribir(cuentaB)

Ambas transacciones son similares a la planteada en el Capítulo 17 para el pago de un servicio. Además, las transacciones están correctamente definidas; por lo tanto, su ejecución aislada lleva la BD de un estado consistente a otro estado consistente, si no se producen fallos. Suponga que la ejecución de T1 y T2 se realiza en un entorno libre de fallos; por consiguiente, cada transacción comienza, se ejecuta y termina, logrando atomicidad.

Observe la Figura 18.1 (¿el descuento de $600 en la segunda transacción sigue en memoria?); aquí se plantea un caso posible de ejecución de T1 y T2. El estado de las cuentas A y B antes de comenzar la ejecución de ambas transacciones era (cuentaA + cuentaB = 5000). En la figura se observa la ejecución de T1 y luego T2 (en secuencia); el estado de las cuentas A y B después de ejecutar las transacciones mantiene (cuentaA + cuentaB = 5000). Para cuentaA, el saldo final es $2.400, y para cuentaB, el saldo final es $2.600.

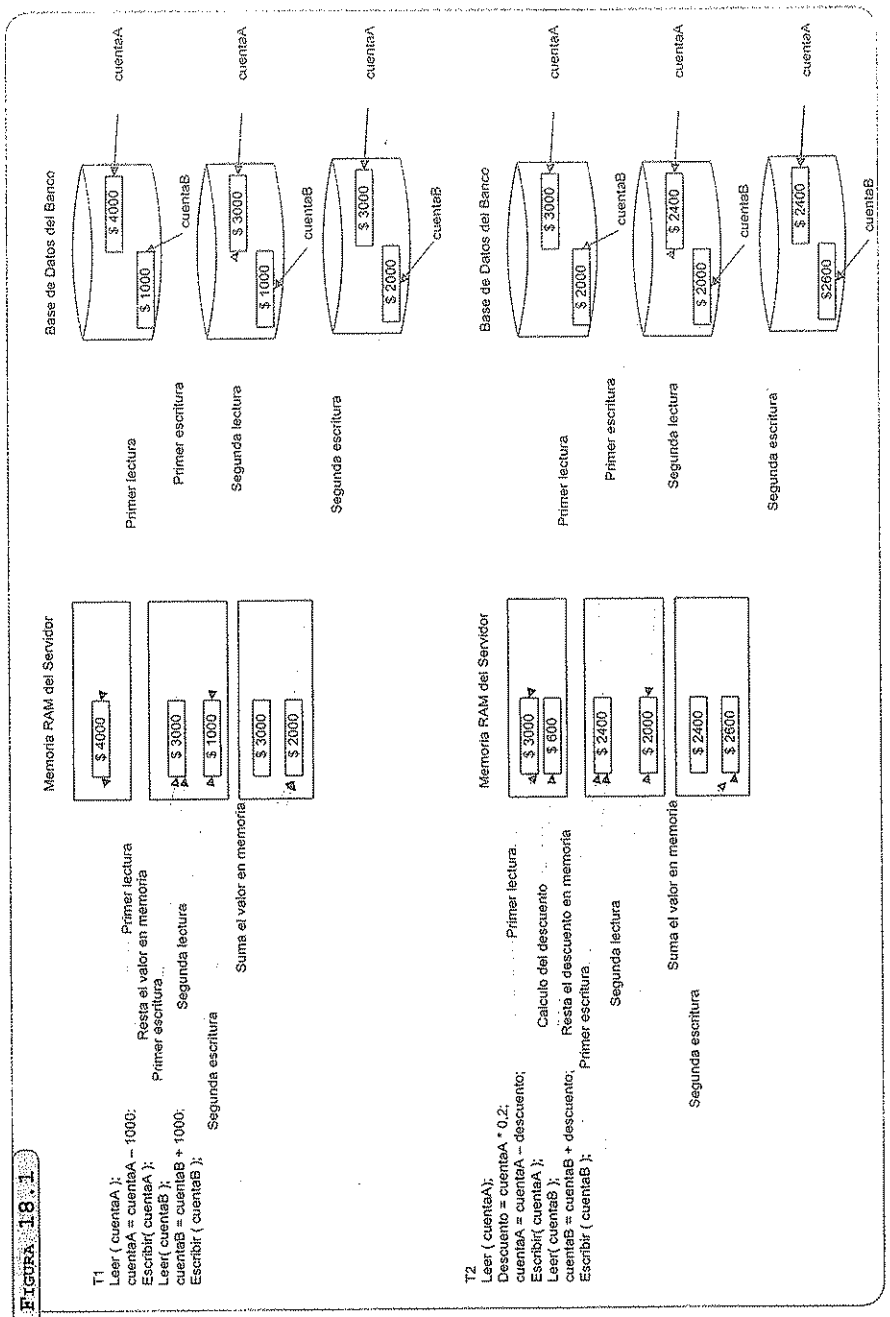


FIGURA 18.1

La Figura 18.2 (ver pág. 405) presenta la ejecución de T2 y luego T1 (nuevamente en secuencia). El estado de las cuentas A y B después de ejecutar las transacciones aún conserva (cuentaA + cuentaB = 5000). En este caso, el saldo final de cuentaA es $2.200 y el de cuentaB es $2.800.

La ejecución de (T1, T2) o (T2, T1) difiere en el resultado final, pero mantiene la BD en estado consistente. Cuando dos transacciones operan sobre el mismo dato, puede suceder que, de acuerdo con el orden de ejecución, el resultado final varíe; no obstante, lo importante para el gestor de BD es mantener la consistencia.

Como se aprecia en las Figuras 18.1 y 18.2, la ejecución de T1 y T2 mantiene la consistencia, y esto es fundamental cuando se procesan las transacciones.

Puede ocurrir que la ejecución de T1 y T2 se realice como muestra la Figura 18.3 (ver pág. 406), y que aun bajo condiciones libres de fallo, la ejecución concurrente de T1 y T2 pueda dejar la BD en un estado de inconsistencia. El estado final es en este caso (cuentaA + cuentaB = 4800), es decir, se perdieron $200 de la BD.

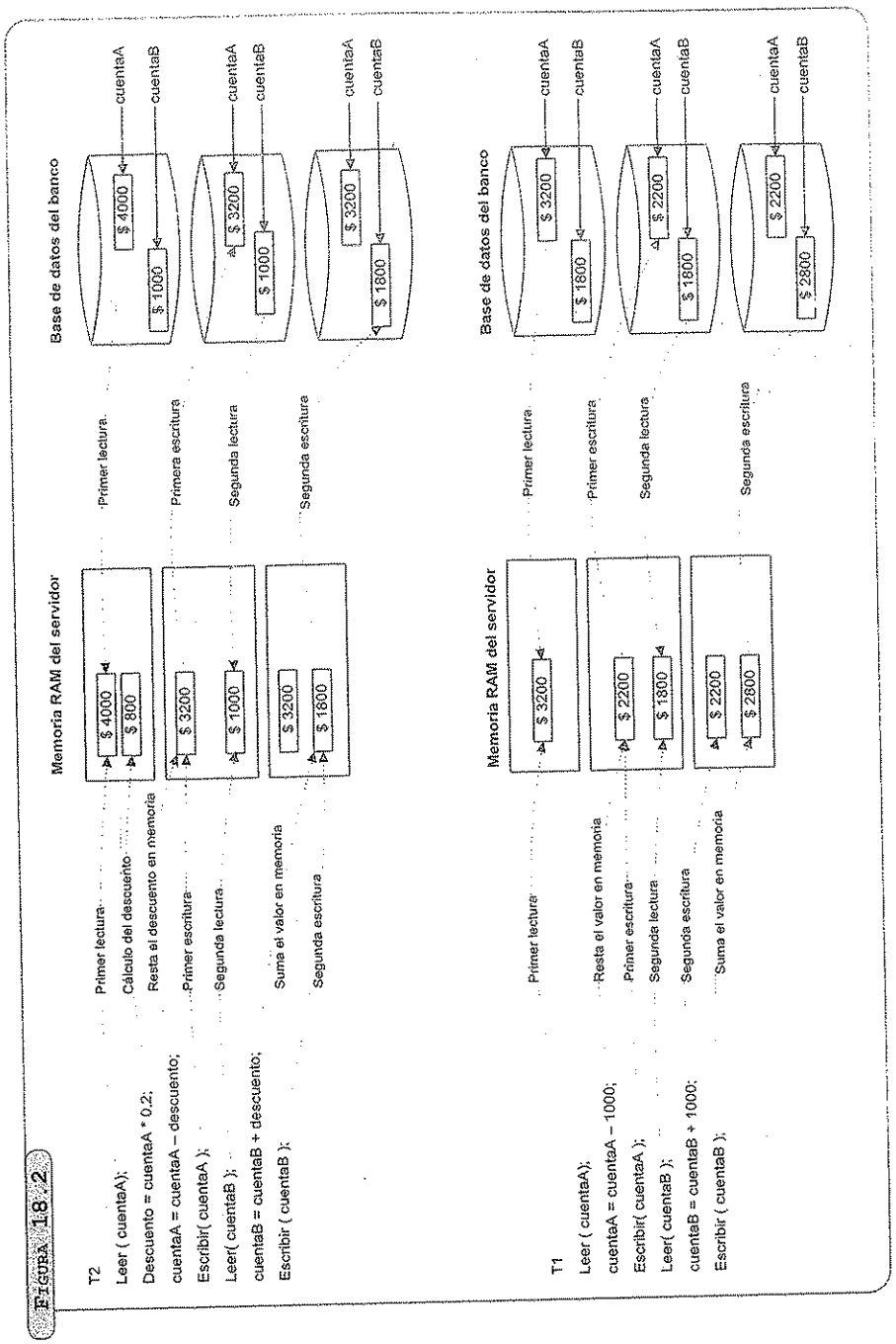
Concepto de transacciones serializables

Planificación

En un entorno que admite varias transacciones ejecutándose en simultáneo, es necesario establecer un criterio de ejecución para que la concurrencia no afecte la integridad de la BD. Las Figuras 18.1 y 18.2 presentaron la ejecución de dos transacciones, T1 y T2, en un entorno concurrente, y la ejecución de estas llevó la BD de un estado consistente a otro estado consistente.

A pesar de que el resultado final presenta diferencias en función del orden de ejecución, la integridad de la BD se respetó, y esta es la propiedad deseada cuando las transacciones acceden y modifican la BD.

En un entorno concurrente, el orden de ejecución en serie de transacciones se denomina planificación. Así, las Figuras 18.1 y 18.2 presentan dos planificaciones diferentes. Es posible notar que si se dispone de dos transacciones, se pueden generar dos planificaciones en serie. Si se ejecutaran tres transacciones (T1, T2 y T3), se podrían generar seis planificaciones en serie diferentes: (T1, T2, T3), (T1, T3, T2), (T2, T1, T3), (T2, T3, T1), (T3, T1, T2), (T3, T2, T1).



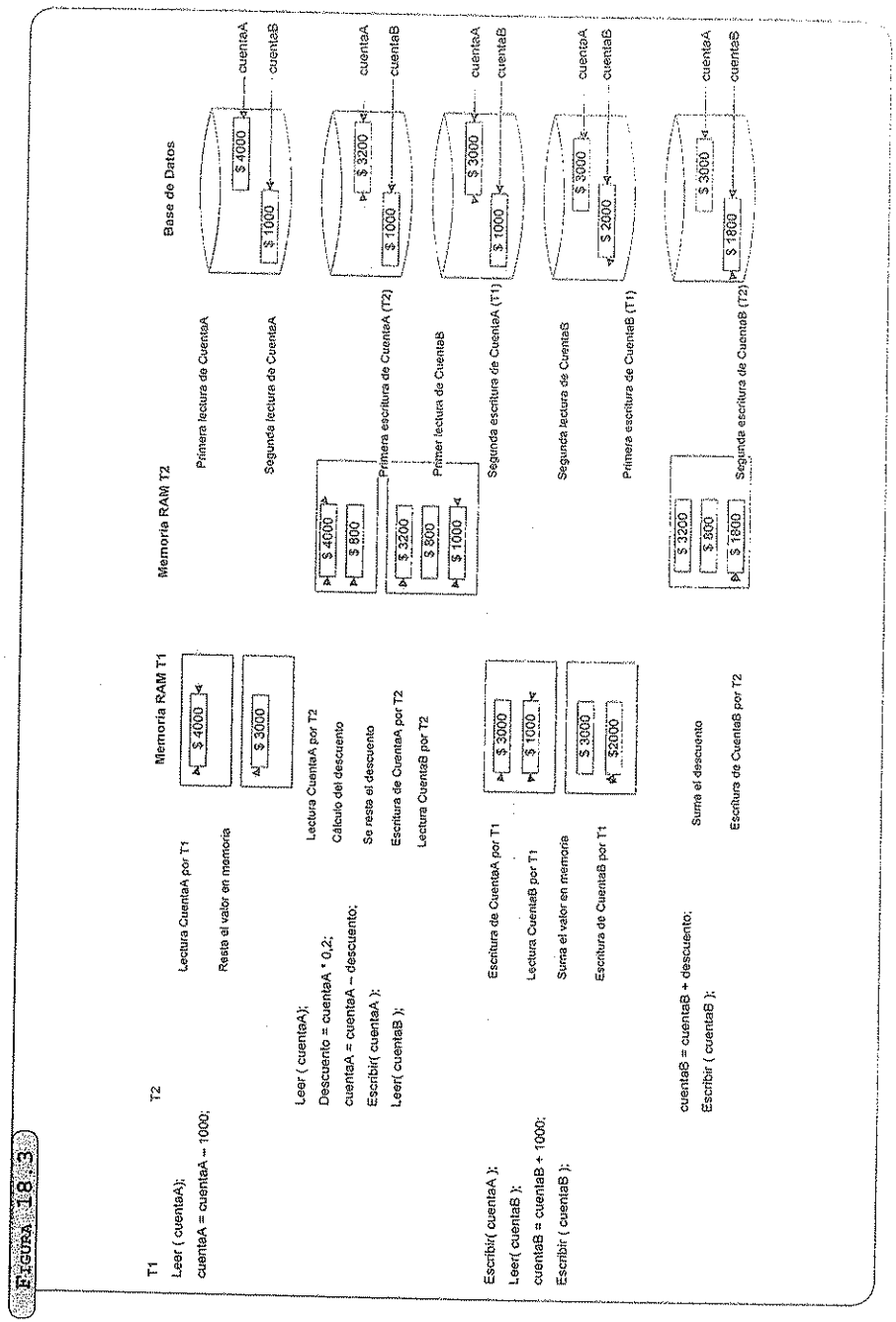


FIGURA 18.3

En general, un entorno concurrente con *N* transacciones en ejecución puede generar *N!* (factorial de *N*) trazas o planificaciones en serie diferentes.

Hasta el momento, una planificación tiene las siguientes características:

- Involucra todas las instrucciones de las transacciones.
- Conservan el orden de ejecución de estas transacciones.

Esto significa que cada transacción debe ser analizada como un todo, sin quitarle instrucción alguna, y que, cuando una transacción comienza, hasta que no finalice, ninguna otra puede iniciar. Esto último garantiza la secuencialidad en la ejecución y, por ende, el cumplimiento de la propiedad de aislamiento para cada transacción.

El problema con esta solución es que todas las ventajas de un entorno concurrente no se aprovechan, y el procesamiento secuencial de *n* transacciones demora mucho más tiempo.

Las planificaciones para un entorno concurrente no deben ser necesariamente en serie.

Cuando la ejecución de una transacción no afecta el desempeño de otra transacción sobre la BD, ambas pueden ejecutarse concurrentemente sin afectar la propiedad de aislamiento.

Suponga que el cliente García del banco abona el servicio de energía eléctrica y al mismo tiempo el cliente Pérez decide abonar el servicio de gas. Se generan dos transacciones, T1 y T2. T1 actúa sobre las cuentas de García y del prestatario de energía eléctrica; T2 actúa sobre las cuentas de Pérez y del prestatario de gas. Estas transacciones, dado que operan sobre subconjuntos de datos diferentes, no generan pérdida de consistencia en un entorno libre de fallos. El aislamiento se garantiza porque actúan sobre datos diferentes.

Planificaciones serializables

Tenga en cuenta el ejemplo analizado anteriormente:

T1	T2
leer(cuentaA)	leer(cuentaA)
cuentaA = cuenta – 1.000;	descuento = cuentaA * 0.2
escribir (cuentaA)	cuentaA = cuentaA – descuento;
leer (cuentaB)	escribir (cuentaA)
cuentaB = cuentaB + 1000;	leer (cuentaB)
escribir(cuentaB)	cuentaB = cuentaB + descuento;
	escribir(cuentaB)

Si T1 y T2 se ejecutan en serie, se garantiza la propiedad de aislamiento.

Sin embargo, en el apartado anterior, se hizo referencia a la necesidad de ejecutar ambas transacciones en forma concurrente.

Analizando la siguiente planificación concurrente:

T1	T2
leer(cuentaA)	
cuentaA = cuenta – 1000;	
escribir (cuentaA)	
	leer(cuentaA)
	descuento = cuentaA * 0.2
	cuentaA = cuentaA – descuento;
	escribir (cuentaA)
Leer (cuentaB)	
cuentaB = cuentaB + 1000;	
escribir(cuentaB)	
	leer (cuentaB)
	cuentaB = cuentaB + descuento;
	escribir(cuentaB)

primero se ejecutan las tres instrucciones iniciales de T1; luego, las cuatro instrucciones iniciales de T2, siguiendo con las últimas instrucciones de T1, y después, las de T2. Si la BD comienza con el estado presentado en la Figura 18.3 (cuentaA = 4000, cuentaB = 1000), se podrá notar que, en caso de no producirse fallo, el estado final de la BD será (cuentaA = 2400, cuentaB = 2600), manteniendo la integridad de la BD.

Por lo tanto, la planificación anterior es una planificación concurrente que lleva la BD de un estado consistente a otro estado consistente.

Suponga ahora la siguiente planificación:

T1	T2
leer(cuentaA)	
cuentaA = cuenta – 1000;	
	leer(cuentaA)
	descuento = cuentaA * 0.2
	cuentaA = cuentaA – descuento;
	escribir (cuentaA)
	leer (cuentaB)
escribir (cuentaA)	
Leer (cuentaB)	
cuentaB = cuentaB + 1000;	
escribir(cuentaB)	
	cuentaB = cuentaB + descuento;
	escribir(cuentaB)

Esta planificación presentada en la Figura 18.3 no mantiene la integridad de la BD. Por lo tanto, no todas las planificaciones concurrentes son válidas.

Las conclusiones que se pueden obtener hasta el momento son las siguientes:

- Se debe mantener la integridad de la BD.
- La inconsistencia temporal puede ser causa de inconsistencia en planificaciones concurrentes. La inconsistencia temporal se presenta durante la ejecución de una transacción que modifica la BD, hasta que termine su ejecución (finalizando correctamente o abortando).
- Una planificación concurrente, para que sea válida, debe equivaler a una planificación en serie. Si se puede demostrar que la ejecución concurrente tiene el mismo resultado que una planificación en serie, entonces la planificación concurrente es válida.
- Solo las instrucciones READ y WRITE son importantes y deben considerarse para realizar cualquier tipo de análisis. Esto se debe a que el resto de las operaciones se pueden llevar a cabo sobre la computadora del usuario, en su memoria local, sin afectar el contenido de la BD.

Para que la ejecución concurrente de T1 y T2 esté en conflicto, las instrucciones que las componen deben cumplir ciertos requisitos. Sean I1 e I2 dos instrucciones de T1 y T2, respectivamente. Si las instrucciones operan sobre datos diferentes, no generan conflicto. Suponga que I1 = WRITE(D1) e I2 = WRITE(D2); no importa cuál instrucción se ejecute primero, D1 y D2 serán modificados y no hay posibilidad de conflicto entre ambos.

Entonces, para que I1 e I2 generen conflicto, deben operar sobre el mismo dato. Suponga que I1 e I2 operan sobre el dato D.

- Si I1=READ(D) e I2=READ(D), no se genera conflicto. Cualquier instrucción que se ejecute primero tendrá el mismo resultado que aquella que se ejecute en segundo lugar, dado que solo se lee.
- Si I1=READ(D), I2=WRITE(D), se genera conflicto. Si se ejecuta primero I1, lee el dato anterior. Si se ejecuta en segundo lugar I1, el valor obtenido corresponderá al escrito por I2. La lectura tiene el mismo resultado si se ejecuta en primer o segundo lugar.
- Si I1=WRITE(D), I2=READ(D), se genera conflicto. Sucede lo mismo que en el caso anterior, pero con diferente orden.
- Si I1=WRITE(D), I2=WRITE(D), se genera conflicto. En este caso, la BD queda con el valor de aquella instrucción que se ejecute en último orden.

Como conclusión, dos instrucciones son conflictivas si operan sobre el mismo dato, y al menos una de ellas es una operación de escritura.

Dos planificaciones, P1 y P2, se denominan equivalentes en cuanto a conflictos, cuando P2 se logra a partir del intercambio de instrucciones no conflictivas de P1.

Una planificación, P2, es serializable en cuanto a conflictos, si existe otra planificación P1 equivalente en cuanto a conflictos con P2, y además P1 es una planificación en serie.

En otras palabras, si P1 es una planificación en serie, entonces su ejecución garantiza aislamiento (si no hay fallos). Si P1 y P2 son planificaciones equivalentes en cuanto a conflictos, significa que P2 obtiene un resultado similar a P1. Por lo tanto, P2 mantiene la consistencia en la BD. P2 será entonces una planificación concurrente que garantiza la integridad de la BD.

Se puede observar que la primera tabla con planificación concurrente entre T1 y T2 es equivalente a conflictos respecto de una planificación en serie. Es posible cambiar el orden de ejecución y obtener una planificación en serie, según muestra la Figura 18.4 (ver pág. 411).

Si se intenta obtener una planificación en serie equivalente, a partir de la segunda planificación concurrente analizada (aquella que generaba inconsistencia en la BD), se podrá observar que no es posible. Aquí se muestra empíricamente que una planificación concurrente debe ser equivalente a una planificación en serie, para que se mantenga la integridad de la BD.

Pruebas de seriabilidad

Existen métodos para demostrar la seriabilidad de planificaciones concurrentes. Estos algoritmos están diseñados para detectar conflictos en la ejecución de dos transacciones. No es propósito de este libro profundizar el estudio de estos algoritmos.

Se puede hacer mención del algoritmo que genera un grafo de precedencia entre las transacciones involucradas en una planificación. En dicho grafo, cada vértice representa una transacción de la planificación, y una arista dirigida entre Ti y Tj significa que Ti realiza una operación de escritura antes que Tj realice una de lectura o escritura. Así, el grafo construido puede tener ciclos. En caso de tener al menos un ciclo, la planificación no es serializable en cuanto a conflictos.

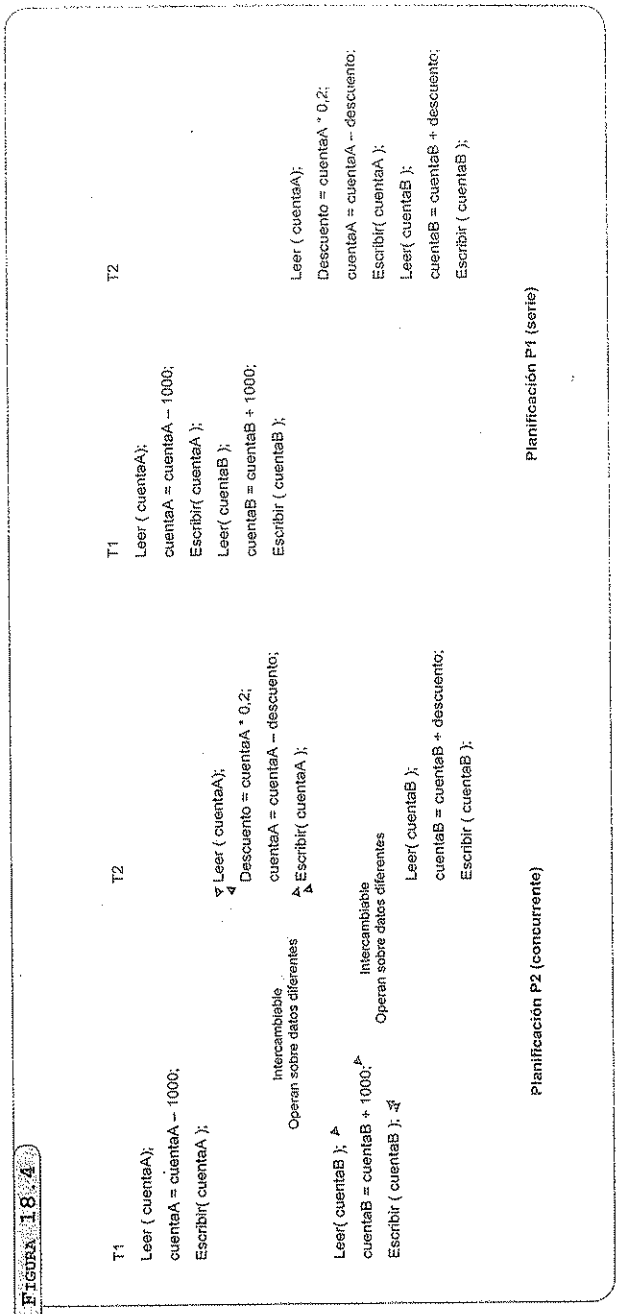


FIGURA 18.4

Implementación del aislamiento. Control de concurrencia

Hasta el momento, se ha discutido cuándo una planificación concurrente es válida y cuándo no. El SGBD debe constar de algo más que de una definición, para poder ejecutar de manera concurrente y correcta una planificación que involucre múltiples transacciones simultáneas.

Como primera medida, es de notar que para que dos transacciones generen conflicto, estas deben operar sobre los mismos datos, aunque, generalmente, cuando se opera con una BD es poco probable que dos transacciones, al mismo tiempo, utilicen la misma información en forma completa.

Existen dos variantes para lograr implementar el aislamiento:

1. Protocolo de bloqueo.
2. Protocolo basado en hora de entrada.

En este apartado, se presentará brevemente el comportamiento de los dos métodos.

Protocolo de bloqueo

El método de bloqueo se basa en que una transacción debe solicitar el dato con el cual desea operar, y tenerlo en su poder hasta que no lo necesite más. El funcionamiento es similar al bloqueo que realiza el SO.

Suponga que en una red de computadoras hay una impresora conectada y que desde cuatro computadoras se envía un listado al mismo tiempo. La impresora es un recurso compartido dentro de la red; por lo tanto, el SO decide cuál trabajo se imprime primero y el resto espera su turno en una cola de impresión.

En el caso de las transacciones, la BD –y, en particular, cada dato contenido en ella– actúa como un recurso compartido. Cada transacción debe bloquear la información que necesita para poder llevar adelante su ejecución, utilizarla y luego liberarla, para que otra transacción que necesite el dato pueda operar.

Básicamente, existen dos tipos de bloqueo a realizar sobre un elemento de datos:

1. Bloqueo compartido.
2. Bloqueo exclusivo.

Un bloqueo compartido debe ser realizado por una transacción para leer un dato que no modificará. Mientras dure este bloqueo, se impide que otra transacción pueda escribir el mismo dato.

Un bloqueo exclusivo se genera cuando una transacción necesita escribir un dato. Dicho bloqueo garantiza que otra transacción no pueda utilizar ese dato (ya sea lectura o escritura de ese dato).

Un bloqueo compartido puede coexistir con otro bloqueo compartido sobre el mismo dato. Suponga que dos transacciones requieren leer el dato D; ambas transacciones deben impedir que otra transacción pueda escribir ese dato, pero como ambas solo quieren leerlo, no se genera conflicto entre ellas. El bloqueo compartido recibe su nombre porque varias transacciones pueden pedirlo y obtenerlo.

Por el contrario, un bloqueo exclusivo es único. Una transacción que modifica un dato de la BD requiere que ninguna otra esté operando con ese dato en ese momento.

Cada transacción debe solicitar el tipo de bloqueo que necesite. Si obtiene el dato, debe utilizarlo y luego liberarlo, cuando la transacción finaliza. Si el dato está siendo usado en ese momento, la transacción tiene dos posibilidades: esperar por el dato, o fallar y abortar, para luego comenzar como una transacción diferente.

Una transacción que solicita un bloqueo y no lo obtiene, como primera alternativa, puede quedar en situación de espera. El programador de la transacción decide su comportamiento. Si se posee certeza de que el dato bloqueado será rápidamente liberado, es posible esperar. En caso contrario, resulta conveniente que la transacción falle y que el usuario sea el encargado de decidir si reinicia o no una nueva transacción.

En general, cuando se opera contra un SGBD, el porcentaje de transacciones que fallan por problemas de bloqueo es menor. Las transacciones suelen obtener los datos o esperar por ellos.

Sean dos transacciones, T1 y T2, como se presentan en la siguiente tabla:

T1	T2
leer(cuentaA) cuentaA = cuenta − 1000; escribir (cuentaA) Leer (cuentaB) cuentaB = cuentaB + 1000; escribir(cuentaB)	leer(cuentaA) leer (cuentaB) presentar (cuentaA + cuentaB)

T1 escribe sobre la BD, mientras que T2 se limita a consultar el contenido. El resultado que muestre T2 dependerá de si T1 se ejecutó o no previamente, pero no puede ocurrir que T1 se haya ejecutado parcialmente. En este último caso, T2 presentará un resultado inconsistente.

Si la ejecución es en serie (T1, T2) o (T2, T1), ya se ha mostrado que tanto el resultado como la BD son consistentes. Sin embargo, lo que se pretende es realizar una ejecución concurrente. Para garantizar una planificación serializable en cuanto a conflictos, se puede utilizar el protocolo de bloqueo.

T1 requiere utilización exclusiva de los datos, mientras que T2 necesita solo utilización compartida. La siguiente tabla agrega, a las transacciones T1 y T2, instrucciones para efectuar el bloqueo de los datos:

T1	T2
bloqueo_exclusivo(cuentaA) leer(cuentaA) cuentaA = cuentaA – 1000; escribir (cuentaA) Liberar_bloqueo(cuentaA) bloqueo_exclusivo(cuentaB) Leer (cuentaB) cuentaB = cuentaB + 1000; escribir(cuentaB) Liberar_bloqueo(cuentaB)	bloqueo_compartido(cuentaA) leer(cuentaA) Liberar_bloqueo(cuentaA) bloqueo_compartido(cuentaB) leer (cuentaB) Liberar_bloqueo(cuentaB) presentar (cuentaA + cuentaB)

Suponga la siguiente secuencia de ejecución concurrente por parte del SGBD, a partir de los valores iniciales de cuentaA=4000 y cuentaB=1000:

1. T1 → bloqueo_exclusivo(cuentaA), resultado → éxito
2. T2 → bloqueo_compartido(cuentaA), resultado → fracaso, T2 queda en espera
3. T1 → leer(cuentaA) (cuentaA = 4000)
4. T1 → cuentaA = cuentaA – 1000 (cuentaA = 3000)
5. T1 → escribir(cuentaA) (cuentaA = 3000, en BD)
6. T1 → liberar_bloqueo(cuentaA)
7. T2 → bloqueo_compartido(cuentaA), resultado → éxito, T2 continúa su ejecución
8. T2 → leer(cuentaA) (cuentaA = 3000)

9. T2 → liberar_bloqueo(cuentaA)
10. T2 → bloqueo_compartido(cuentaB), resultado → éxito
11. T1 → bloqueo_exclusivo(cuentaB), resultado → fracaso, T1 queda en espera
12. T2 → leer(cuentaB) (cuentaB = 1000)
13. T2 → liberar_bloqueo(cuentaB)
14. T2 → presentar (cuentaA + cuentaB) → presenta 4000
15. T1 → bloqueo_exclusivo(cuentaB), resultado → éxito, T1 continúa su ejecución
16. T1 → leer(cuentaB) (cuentaB = 1000)
17. T1 → cuentaB = cuentaB + 1000, (cuentaB = 2000)
18. T1 → escribir (cuentaB) (cuentaB = 2000, en la BD)
19. T2 → liberar_bloqueo(cuentaB)

Puede observarse que, en el Paso 14, la transacción T2 obtiene un valor nuevo para cuentaA y uno viejo para cuentaB. Hasta ese momento, T1 se encontraba parcialmente ejecutada.

Por lo tanto, aun utilizando protocolos de bloqueo, no se garantiza el aislamiento de una transacción. ¿Cuál es la solución en este caso? Una transacción debe pedir todo lo que necesita al principio, utilizarlo y luego liberarlo. De esta forma, se pueden mejorar las condiciones de aislamiento, llevando la ejecución de transacciones que pueden generar conflicto (dado que operan sobre el mismo dato) a una ejecución en serie. Solo las transacciones que operen sobre datos diferentes podrán ejecutarse concurrentemente con total aislamiento.

Trasladar los bloqueos al comienzo de la transacción no soluciona el problema en un 100%, y además puede plantear situaciones de *deadlock* (abrazo mortal).

Si bien no es objetivo de este libro profundizar el análisis de estas situaciones, se puede definir la condición de *deadlock* como una situación en que dos procesos obtienen un elemento y ambos solicitan el elemento que tiene el otro. Como ninguno de los dos procesos puede obtener el segundo elemento solicitado, se produce *deadlock*, que impide continuar con la ejecución de los procesos.

El siguiente ejemplo ilustra la situación:

T1	T2
bloqueo_exclusivo(cuentaA)	bloqueo_compartido(cuentaB)
bloqueo_exclusivo(cuentaB)	bloqueo_compartido(cuentaA)

1. T1 → bloqueo_exclusivo(cuentaA), resultado → éxito
2. T2 → bloqueo_compartido(cuentaB), resultado → éxito
3. T1 → bloqueo_exclusivo(cuentaB), resultado → fracaso, T1 en espera
4. T2 → bloqueo_compartido(cuentaA), resultado → fracaso, T2 en espera

El sistema se encuentra bloqueado (*deadlock*). Existen diversos tratamientos de situaciones de *deadlock*, aunque no son menester de este material.

Como conclusión, una transacción que traslada sus bloqueos al comienzo puede generar *deadlock*. Por el contrario, si los bloqueos se ejecutan cuando son necesarios, las posibilidades de *deadlock* disminuyen, pero el riesgo de pérdida de consistencia aumenta. No obstante, es preferible generar situaciones de *deadlock*, dado que estas situaciones no conducen a la pérdida de consistencia en los datos.

Para asegurar la integridad de la BD, el protocolo de bloqueo necesita reglas bien definidas. Aunque existen varias alternativas, en este libro se presenta el protocolo denominado de dos fases.

El protocolo de bloqueo de dos fases garantiza aislamiento en la ejecución de las transacciones, utilizando el principio de las dos fases: crecimiento y decrecimiento.

Para que una transacción sea correcta, el orden de pedidos de bloqueos debe ser como sigue.

- **Fase crecimiento:** se solicitan bloqueos similares o se crece de compartido a exclusivo.
- **Fase decrecimiento:** exclusivo, compartido, liberar datos.

El siguiente es un ejemplo no válido de transacción, con protocolo de bloqueo de dos fases:

Bloqueo_Compartido(dato2)

Bloqueo_Exclusivo(dato1) (crece)

Bloqueo_Compartido(dato3) (decrece)

Bloqueo_Exclusivo (dato4) (crece nuevamente)

El protocolo de bloqueo es mucho más amplio que el apartado abarcado en este libro. Las políticas de detección de *deadlock*, inanición, protocolo de dos fases y otras alternativas están desarrolladas con más detalle en materiales relacionados con SO o programación concurrente.

Desde la perspectiva de BD, los protocolos de control de concurrencia actúan como una caja negra que resuelve los problemas básicos de acceso a la información.

Protocolo basado en hora de entrada

Si se utiliza el protocolo de bloqueo, el SGBD decide cuál de las transacciones conflictivas se ejecuta primero, en función del orden de pedido de bloqueos compartidos y/o exclusivos. El método requiere protocolos adicionales, como por ejemplo dos fases, para asegurar que la integridad de la BD se mantiene, pero no se garantizan situaciones libres de *deadlock*.

El protocolo basado en hora de entrada es una variante del protocolo de bloqueo, donde la ejecución exitosa de una transacción se establece de antemano, en el momento en que la transacción fue generada. Cada transacción recibe una **Hora de Entrada (HDE)** en el momento en que inicia su ejecución. La HDE es una marca única asignada a una transacción. Esta marca única puede ser el valor de un contador o la hora del reloj interno del servidor de la BD.

Suponga que dos transacciones, T1 y T2, modifican la BD. Si T1 comenzó antes que T2, entonces HDE(T1) < HDE(T2). Si T1 fuera posterior a T2, entonces HDE(T1) > HDE(T2). En ningún caso puede ocurrir que HDE(T1) = HDE(T2).

El método continúa asignando a cada elemento de dato de la BD dos marcas temporales: la hora de última lectura y la hora de última escritura. Estas marcas corresponden a la HDE de la última transacción que leyó o escribió el dato. Se denomina HL(D) a la hora en que el dato fue leído por última vez, y HE(D) a la hora en que el dato fue escrito por última vez.

Para decidir si una transacción puede o no ejecutar su operación de lectura/escritura sobre la BD, el protocolo toma la decisión basado en el siguiente cuadro:

- Una operación de lectura. Suponga que T1 desea leer el dato D.
  - Para poder leer el dato, se debe cumplir que HDE(T1) > HE(D). Esta acción asegura que cualquier transacción que intente leer un dato debe haberse iniciado en un momento posterior a la última escritura del dato.
  - Si HDE(T1) < HE(D), el dato D es demasiado nuevo para que T1 pueda leerlo. Una transacción posterior a T1 pudo escribir el dato. Si se acepta la lectura del dato D por parte de T1, es posible que T1 plantee una situación de pérdida de consistencia de información.

    Para leer un dato de la BD, el protocolo basado en HDE no necesita chequear la HL(D). Esto se debe a que las lecturas pueden ser compartidas. Varias transacciones pueden leer el mismo dato sin generar conflicto entre ellas.

En caso de que la operación de lectura del dato D tenga éxito, se debe establecer la HL(D) como el máximo entre HDE(T1) y HL(D).

- Una operación de escritura necesita chequear más información. Suponga ahora que T1 necesita escribir el dato D.
  - Si HDE(T1) < HL(D), la operación falla. No es posible que T1 escriba el dato D si fue leído por una transacción posterior a ella. Si se aceptara esa operación, la transacción que leyó resultaría incorrecta.
  - Si HDE(T1) < HE(D), la operación nuevamente falla. No es posible que T1 escriba el dato D, cuando dicho dato ya fue escrito por una transacción posterior.
  - En cualquier otro caso, T1 puede ejecutar la operación de escritura, y establece HE(D) con el valor de HDE(T1).

Cualquier transacción que no puede seguir su ejecución falla y aborta su trabajo. Posteriormente, se puede generar una nueva transacción con una nueva HDE, e intentar ejecutar la operación deseada.

Existen variantes del método de HDE, así como otros métodos que aseguran el aislamiento en la ejecución de las transacciones, aunque están fuera del alcance de este libro.

Granularidad

El concepto de granularidad en el acceso a los datos está vinculado con el tipo de bloqueo que se puede realizar sobre la BD.

La granularidad gruesa permite solamente bloquear toda la BD, y a medida que la granularidad disminuye, es posible bloquear a nivel tabla, registro o hasta campos que la componen.

En general, los SGBD permiten diversos niveles de bloqueo sobre los datos. El nivel mayor de bloqueo es sobre la BD completa. Esta acción está normalmente reservada para el diseñador de la BD, y se produce en actividades de mantenimiento. Durante este bloqueo, ningún usuario puede acceder a la BD.

Las transacciones en entornos concurrentes necesitan bloqueos a nivel de registros. Los bloqueos compartidos y exclusivos de datos apuntan a bloquear el uso o la escritura de registros (tuplas) de la BD.

Algunos SGBD permiten generar un nivel de granularidad aún menor. En esos casos, el bloqueo se puede realizar a nivel de campos o atributos de un registro o tupla.

Otras operaciones concurrentes

No solamente las operaciones de lectura y escritura deben garantizar el acceso aislado a la BD. Operaciones como insertar o borrar tuplas de una tabla también necesitan garantizar su operación en forma aislada.

Cuando se ejecuta una instrucción SQL INSERT, se debe garantizar el acceso único de dicha operación. El proceso de escritura de una tupla requiere atomicidad; por lo tanto, mientras una transacción está ejecutando una operación de INSERT, cualquier otra transacción no puede leer, escribir ni borrar los datos que se están insertando.

De la misma forma, si mediante una transacción se está borrando una tupla de una tabla, no es posible que se intente leer o modificar la misma tupla.

Las operaciones de inserción y borrado sobre la BD cumplen los mismos requisitos que las operaciones de lectura/escritura. Para poder insertar o borrar una tupla, es necesario tener el control exclusivo de los datos o, en su defecto, el protocolo basado en HDE debe cumplir las mismas condiciones que una operación de escritura.

Bitácora en entornos concurrentes

El protocolo de bitácora o registro histórico definido en el Capítulo 17 se aplica casi sin cambios en entornos concurrentes. La finalidad del protocolo es garantizar la atomicidad de la transacción ante posibles fallos originados en su ejecución.

En el caso de entornos concurrentes, se agrega un nuevo tipo de fallo, una transacción que no puede continuar su ejecución por problemas de bloqueos o acceso a la BD. En ese caso, como en cualquier otra situación de fallo, la transacción debe abortar, dejando la BD en el estado de consistencia anterior al comienzo de su ejecución.

Para el resto del apartado, no se tendrá en cuenta si el protocolo de registro histórico trabaja con modificación inmediata o diferida de la BD. El comportamiento de ambas variantes es totalmente análogo en entornos concurrentes y monousuarios. Es decir, ambos tienen definida la función REHACER, y la modificación inmediata agrega la función DESHACER.

No obstante, es necesario realizar algunas consideraciones adicionales:

- Existe un único *buffer* de datos tanto para la BD como para la bitácora.
- Como se presentó anteriormente, cada transacción tiene su propia área donde administra la información localmente.

- El fallo de una transacción significa deshacer el trabajo llevado a cabo por ella. Esta acción, si bien puede ser sencilla y fácil de analizar dado que es similar a entornos monousuarios, presenta algunas características que deben ser consideradas.

Retroceso en cascada de transacciones

El retroceso de una transacción puede llevar a que otras transacciones también fallen. Suponga el siguiente ejemplo. La transacción T1 opera contra los datos 1, 2, 3 y 4; para su desarrollo utiliza varios ciclos de CPU. Obtiene, actualiza y libera el dato1. La transacción T2 obtiene el dato1, lo utiliza y finaliza su ejecución. Mientras tanto, la transacción T1 continuó su ejecución y produjo un fallo; la transacción T1, entonces, falla; por ende, aborta, retrocediendo todo lo actuado.

El problema en este punto es que la transacción T2 utilizó un dato modificado por T1, pero al producirse el fallo de esta última (T1), el valor utilizado por T2 no se puede considerar correcto. Por lo tanto, T2 también debe fallar. En este punto, T2 está en estado de finalizada, y por la condición de durabilidad no puede abortarse.

¿Cómo actuar en este caso? Se debe agregar una nueva condición al problema. Una transacción Tj no puede finalizar su ejecución, si una transacción Ti anterior utiliza datos que Tj necesita, y Ti no está en estado de finalizada. De esta forma, si Ti falla, Tj también deberá fallar. Este tipo de acción puede llevar a retrotraer varias transacciones.

Puntos de verificación de registro histórico

El único cambio que requiere el método de registro histórico con respecto a entornos monousuarios está vinculado con los puntos de verificación. En un entorno monousuario, un punto de verificación se agrega periódicamente luego de <Ti finaliza> y antes de <Tj comienza>.

El registro histórico de un esquema concurrente no puede garantizar la existencia de un momento temporal, donde ninguna transacción se encuentre activa. Por este motivo, la sentencia

<punto de verificación (L)>

agrega un parámetro L. Este parámetro contiene la lista de transacciones que, en el momento de colocar el punto de verificación, se encuentran activas.

Ante un fallo, el algoritmo de recuperación actúa como se definió en el Capítulo 17, a excepción de las transacciones que están en la lista del último punto de verificación. Para las transacciones allí contenidas, se debe revisar la bitácora aún más atrás del punto de verificación, dado que <Tj comienza> quedó definido previo a ese punto.

Cuestionario de capítulo

1. ¿Qué es el protocolo de bloqueo de dos fases? ¿Cómo garantiza la seriabilidad?
2. ¿Qué es una hora de entrada? ¿Cómo genera el sistema las HDE?
3. ¿El protocolo por HDE produce bloqueos? ¿Por qué?
4. ¿Por qué los retrocesos en cascada pueden resultar costosos cuando el sistema se recupera de una caída? ¿En qué orden se deben volver a rehacer y deshacer las transacciones? ¿Por qué dicho orden es importante?
5. ¿Qué diferencia presenta el método de bitácora entre entornos monousuarios y concurrentes?

Ejercitación

1. Considere las siguientes transacciones:

T0: read(A)	T1: read(B)
read(B)	read(A)
If A=0 then B:=B+1	If B=0 then A:=A+1
write(B)	write(A)

Sea el requisito de consistencia A = 0 o B = 0, siendo los valores iniciales A = B = 0.

   a. Demuestre que toda ejecución en serie de estas dos transacciones conserva la consistencia de la BD.
   b. Muestre una ejecución concurrente de T0 y T1 que produzca una planificación no serializable.
   c. ¿Existe una ejecución concurrente de T0 y T1 que produzca una planificación serializable?

2. Suponga que el sistema falla en el momento en que se está recuperando de una caída anterior, en un entorno concurrente. Cuando el sistema se recupere nuevamente, ¿qué acción deberá llevarse a cabo?

3. Dadas las siguientes transacciones:

T1	T2
read(x);	read(x);
x:=x−n;	x:=x+ m;
Write(x);	write(x);
Read(y);	
y:= y+n;	
write(y);	

haga una lista con todos los planes posibles para las transacciones T1 y T2, y determine cuáles son serializables en cuanto a conflictos y cuáles no.

4. Sean las siguientes transacciones:

T1	T2	T3
Read(x);	Read(z);	Read(y);
Write(x);	Read(y);	Read(z);
Read(y),	Write(y);	Write(y);
Write(y);	Read(x);	Write(z);
	Write(x);	

a. ¿Cuántos planes en serie existen para las tres transacciones?

b. ¿Cuáles son?

c. ¿Cuál es el número total de planes posibles?

CAPÍTULO 19

Seguridad e integridad de datos

Objetivo

Este capítulo estudia los casos y las soluciones viables para proteger la BD contra accesos no autorizados a la información.

En primer lugar, se compara el concepto de seguridad de la información contra el concepto de integridad. Si bien desde una perspectiva amplia la seguridad y la integridad de los datos pueden ser tratadas como sinónimos, existen interpretaciones diferentes para ambos conceptos. Mientras que la integridad procura mantener un estado consistente de la BD durante la operatoria normal, la seguridad busca asegurar la BD contra destrucciones malintencionadas.

Por último, se discuten genéricamente conceptos relacionados con cuestiones de ética y legalidad en el uso de una BD.

Concepto de seguridad

Proveer seguridad a una BD consiste en protegerla de intentos de acceso malintencionados. El concepto de seguridad tiene que ver con proteger la BD y, por consiguiente, la información almacenada en ella, contra actualizaciones realizadas por personas/usuarios no autorizados. Además, la protección de los datos se amplía al acceso para consulta. No todos los usuarios pueden acceder a consultar toda la información contenida en una BD.

Por otro lado, la integridad de la BD está ligada con la protección de los datos ante pérdidas accidentales de consistencia. Para ello, en capítulos anteriores se han analizado los problemas que pueden

surgir; se presentaron las transacciones y sus propiedades, como el tratamiento ante posibles pérdidas de consistencia. El procesamiento de transacciones provee una serie de mecanismos que aseguran la integridad y la consistencia de la BD, ante errores que puedan surgir a partir del procesamiento normal y cotidiano por parte de los usuarios.

Los problemas de integridad están relacionados con la pérdida de consistencia generada por el procesamiento normal de transacciones, sin que los usuarios pretendan generar situaciones de anomalía. Cuestiones como fallo de disco, interrupción de suministro de energía o acceso concurrente a los datos pueden generar que una transacción falle, y que la BD quede en un estado de inconsistencia.

La seguridad sobre una BD se puede implantar en varios niveles. El primer nivel de seguridad sobre una BD se denomina nivel físico. Para que una BD esté físicamente segura, se debe resguardar el servidor que la contiene de cuestiones que tienen que ver con robo, destrucción por catástrofes, incendios, etcétera. El ambiente físico para almacenar un servidor de datos debe cubrir requisitos de instalación y seguridad.

Las soluciones más viables para lograr mejoras de seguridad a nivel físico son las siguientes:

- Replicar el *hardware* de servidores, es decir, tener varios servidores de datos que actúen como espejos entre sí. En caso de producirse fallo en alguno de ellos, otro toma el control de manera transparente para los usuarios. Existe un nivel más de seguridad que consiste en tener los servidores replicados en sitios geográficamente distantes. Suponga que un servidor está ubicado en una región geográfica propensa a sismos; en ese caso, quizás una solución viable sería ubicar una réplica de dicho servidor a suficiente distancia para garantizar que ambos servidores no estén afectados por un eventual sismo.
- Replicar dispositivos de almacenamiento, ya que los servidores de datos están preparados para contener varios discos rígidos. Estos discos nuevamente pueden actuar como espejos entre sí. La rotura de un disco solamente implica que el otro toma automáticamente su lugar. Además, los discos pueden ser removibles; esto significa que pueden ser quitados del servidor y reemplazados por otro sin detener el funcionamiento de dicho equipo. Estas soluciones mejoran sin duda la seguridad de la BD.
- Dotar de alarmas, personal extra de seguridad, cámaras de vigilancia, etc., que permitan detectar el acceso de personas no autorizadas a los sitios donde estén ubicados los servidores de datos.

Otro nivel de seguridad es el denominado nivel humano. Los usuarios que tienen acceso a la BD deben registrar su conexión a través de una clave de usuario. Este punto representa el eslabón más débil de la seguridad en el acceso a una BD. Los usuarios no siempre mantienen en reserva sus claves de acceso. La mejor solución al respecto consiste en obligar a cambiar las claves cada cierto tiempo, e informarle además a cada usuario que sus accesos quedan registrados en la BD. Por ende, si un intruso accede utilizando la identidad de cierto usuario, todos los cambios producidos quedarán registrados bajo esa identidad.

A nivel de seguridad del SO, es posible controlar la cantidad de veces que un usuario se conecta al sistema ingresando palabras claves incorrectas. De esta forma, se puede cancelar momentáneamente la cuenta de usuario, previniendo accesos indebidos. Además, desde el SO se puede limitar el acceso a recursos de cada usuario; por ejemplo, obtener acceso parcial a las facilidades de manejo de archivos relacionados con la BD. Cercanos al nivel de seguridad del SO están los protocolos de seguridad del nivel de redes. Algunos aspectos a considerar tienen que ver con asegurar la comunicación entre servidores autorizados y proteger los accesos contra robos y modificación de mensajes, proveyendo mecanismos de identificación y cifrado para estos mensajes.

Por último, se definen dos niveles de seguridad, relacionados con el SGBD y el administrador de la BD.

El nivel de seguridad del SGBD resume los niveles anteriores agregando aspectos propios. Por cada BD específica se deben definir sus propios usuarios con sus claves de acceso. Cada uno de estos usuarios debe tener definidas autorizaciones de acceso a archivos, datos u operaciones que puede realizar. Esto significa que no todos los usuarios de una BD pueden acceder a toda la información. Puede ocurrir, además, que un usuario tenga acceso total (lectura y escritura) a una determinada tabla, pero que no tenga disponible la opción de crear un nuevo índice secundario sobre dicha tabla.

El nivel de seguridad del SGBD es más complejo en entornos concurrentes y en entornos distribuidos. Aunque los entornos distribuidos no son tema de este libro, el Capítulo 20 define brevemente sus características y algunas consideraciones adicionales.

Finalmente, el nivel de seguridad del administrador de la BD está ligado en forma directa con las políticas que este último implementa respecto del control de accesos y con la política de actualización de datos sobre cada BD en particular. Algunas preguntas cuyas respuestas debe determinar el administrador de la BD respecto del nivel de

seguridad son: ¿qué datos puede visualizar cada usuario?, ¿qué tipos de operaciones puede realizar?, ¿qué acciones puede llevar a cabo sobre el modelo de datos? La respuesta a este último interrogante está vinculada, por ejemplo, con permitir al usuario crear nuevos índices para mejorar el tiempo de respuesta ante un acceso particular a los datos. En estos casos, el administrador de la BD debe decidir entre permitir esta opción que favorece a un usuario específico o no permitirla priorizando la operatoria general sobre esa tabla.

El concepto de auditoría de BD está ligado al registro de todas las operaciones realizadas por cada usuario sobre la BD. Cuando una BD es auditada, se agrega una característica más de seguridad. Así, es posible reconocer el usuario, el tipo de operación que este realizó sobre los datos, y el día y la hora de dicha operación. Las auditorías son importantes en aquellas BD con información actualizada por múltiples transacciones y usuarios, donde se desea registrar al responsable de cada operación. La BD de un banco o el registro de calificaciones en una universidad son ejemplos de BD auditadas.

Autorizaciones y vistas

El concepto de autorización tiene que ver con generar cuentas de usuario que permitan acceso selectivo a cada tabla de la BD. Además, es posible controlar el tipo de operaciones que cada usuario puede llevar a cabo. Las autorizaciones pueden realizarse a dos niveles.

El primer nivel consiste en permitir interactuar contra el modelo de datos. De esta forma, un usuario podrá actualizar tanto el esquema de las tablas como los índices.

El segundo nivel tiene que ver con el acceso a los datos. Se establecen los privilegios de los usuarios para actualizar la información contenida en la BD. Los privilegios se pueden establecer a nivel de consulta, inserción, borrado y/o modificación de la BD. Para ello, el administrador de la BD determina, para cada usuario o grupo de usuarios, el tipo de operaciones SQL que son válidas: SELECT, INSERT, DELETE o UPDATE.

El concepto de vista puede utilizarse como un mecanismo de autorización y, por ende, como un nivel más de seguridad sobre la BD. Este concepto fue analizado en el Capítulo 15, cuando se presentó el lenguaje SQL. Al permitir el acceso a vistas a determinado perfil de usuario, solamente este puede manipular los datos que la vista contiene. De esta forma, para un usuario que solo tiene acceso a una vista específica, únicamente son visibles los datos que en ella aparecen.

Encriptado de información

La criptografía está vinculada con el cifrado y el descifrado de información, mediante técnicas o algoritmos especiales. Se emplea frecuentemente para permitir un intercambio de mensajes que solo puedan ser reconocidos por las personas o usuarios que los intercambian. Cuando este concepto se aplica a una BD, la idea es permitir que una BD (estructura y datos contenidos) solo sea accesible por los usuarios o personas autorizados.

La finalidad perseguida por la criptografía es garantizar la confidencialidad en la información, y asegurar que los datos no sean corrompidos.

Si bien desde la perspectiva de este libro el concepto de criptografía no será desarrollado con detalle, es posible mencionar dos tipos básicos de criptografía: simétrica y asimétrica.

La criptografía simétrica es un método que utiliza la misma clave para cifrar y descifrar la información. El algoritmo utilizado es el mismo. La clave utilizada es similar.

La criptografía asimétrica utiliza dos claves diferentes para el proceso de conversión de la información. El algoritmo de encriptado es público y conocido por todas las personas que desean enviar información, en tanto que el algoritmo de desencriptado es solamente conocido por el receptor del mensaje. Una clave es pública y se puede entregar a cualquier persona, la otra clave es privada y el propietario debe guardarla de modo que nadie tenga acceso a ella.

Bases de datos estadísticas

En alguna oportunidad podría ser necesario que una organización deba "abrir" su BD, para que esta sea consultada con fines estadísticos por parte de alguna organización gubernamental. Dichos estudios estadísticos sobre los datos no deben describir información alguna en particular, sino resumir los datos almacenados en la BD. Así por ejemplo, no se debe poder determinar qué producto específico compró cierto cliente, sino las estadísticas de venta mensual de productos y las ventas promedio realizadas a clientes, entre otras cuestiones.

En todo momento se debe asegurar el acceso a la información, pero manteniendo la intimidad de los datos contenidos en la BD. Para mantener la intimidad de la información, se debe habilitar, a los

usuarios que realizan estadísticas, solo las funciones de agregación para contabilizar, sumar o promediar datos.

Sin embargo, esta alternativa presenta puntos de fallo. Suponga que se realizan dos consultas; la primera de ellas obtiene el promedio de ventas para clientes de una determinada localidad, y la segunda, la cantidad de clientes de esa localidad. Si la respuesta a la última consulta fuese 1, el promedio de las ventas para clientes de esa localidad sería el total comprado por el único cliente.

Una solución al problema anterior es que las consultas estadísticas deben tener definido un umbral de trabajo. Por ende, si la consulta efectuada involucra menos tuplas que una población previamente establecida, la consulta es rechazada.

Otro punto de vulnerabilidad es que se realicen consultas que involucren siempre el mismo universo de tuplas, con el fin de violar restricciones de seguridad y privacidad en los datos. En este caso, se debe impedir que, a partir del análisis estadístico del mismo conjunto de datos, se pueda extraer información particular. Los SGBD presentan una alternativa para salvar estos casos. Para que un usuario obtenga estadísticas de una BD, la intersección de los conjuntos de tuplas utilizados en las consultas no debe superar otro umbral definido. Así, si la primera consulta utiliza N tuplas y la segunda utiliza m tuplas, la intersección de las tuplas de cada consulta no puede ser superior a r tuplas, siendo r el umbral definido.

Para solucionar los inconvenientes de seguridad con BD estadísticas, existen algoritmos alternativos que están fuera del alcance de este libro.

Cuestionario del capítulo

1. ¿Qué diferencia puede establecer entre los conceptos de seguridad e integridad de datos?
2. ¿Por qué es importante garantizar la seguridad de la BD?
3. ¿Qué prevenciones de seguridad de datos utilizaría en cada uno de los siguientes problemas?
   a. BD de un SI correspondiente a una farmacia que funciona en una intranet.
   b. BD de un SI que implementa un carrito de compras de un centro de compras.
   c. BD de un sistema bancario de un banco multinacional.
   d. BD de un SI que implanta la logística de distribución de una compañía de transporte presente en todo un país.
4. ¿Qué tipos de seguridad implanta el administrador de la BD?
5. ¿Por qué una vista de una BD permite aumentar la seguridad de la BD?
6. ¿Por qué se criptografía la información de una BD?
7. ¿Todas las BD necesitan estar criptografiadas? ¿Cuál es el motivo?
8. ¿Qué significa administrar una BD estadística?

VI

Conceptos avanzados

CAPÍTULO 20
Otras aplicaciones de BD

CAPÍTULO 20

Otras aplicaciones de BD

Objetivo

Los temas desarrollados hasta el Capítulo 19 describen el objetivo primario de este libro. Cada uno de los conceptos vertidos desarrolla los temas previstos para un curso de Bases de Datos que involucra conceptualización de información desde archivos, representación física y lógica de los datos, técnicas de accesos y recuperación de información, y seguridad e integridad de datos.

Sin embargo, estos conceptos solo presentan una introducción al problema completo que un estudiante de Bases de Datos debería conocer. Las BDOO y las BD distribuidas representan en la actualidad dos conceptos básicos para el estudio y la recuperación de la información en un sistema de gestión.

Este capítulo tiene la finalidad de brindar someramente conceptos generales sobre temas relacionados con tendencias actuales de BD. En el primer apartado, se define y describe el concepto de BD distribuidas. El segundo explica las características básicas de las BDOO. Los últimos apartados, en tanto, presentan el concepto de almacén de datos o *data warehouse*, como alternativa de repositorio para la toma de decisiones en grandes organizaciones, y lo comparan con el concepto de *data mart*. Se discute, además, el concepto de minería de datos o *data mining* para la recuperación de información no predecible de la BD.

Finalmente, se presentan conceptos más recientes, como sistemas de información geográfica, BD textuales y BD móviles.

Bases de datos distribuidas

Una **Base de Datos Distribuida (BDD)** es una colección lógicamente relacionada de datos compartidos, físicamente distribuidos en una red de computadoras. Esos datos compartidos contienen, además, una descripción de las estructuras de datos que los componen, el diccionario de datos.

Un **Sistema de Gestión de Bases de Datos Distribuidas (SGBDD)** es un SGBD que permite gestionar y administrar una BDD. Su característica esencial es que la distribución de la información sea transparente para el usuario.

Un SGBDD administra, para un problema particular, una única BD que se encuentra lógica y físicamente dividida en una serie de fragmentos almacenados en diferentes computadoras. El control de cada fragmento de la BD depende, en cada servidor, de un SGBD local. Este SGBD local actúa de manera independiente respecto de los otros SGBD de otros servidores, pero todos cooperan para que un usuario pueda acceder a los datos como si se tratara de una BD centralizada.

Cada usuario accede a la información de la BD a partir de transacciones. Las transacciones en un entorno distribuido pueden ser de dos tipos:

1. **Locales:** se genera una transacción que se ejecuta sobre un único servidor (sin necesidad de acceder a otros servidores). En este caso, la transacción actúa como las ya definidas en entornos centralizados.
2. **Globales:** se genera una transacción que debe ejecutarse sobre varios servidores, debido a que necesita información de la BD contenida en diversos fragmentos.

Un entorno distribuido de BD tiene un conjunto de características fundamentales:

- La información de la BD se encuentra fragmentada en porciones. La forma de fragmentación es uno de los temas más importantes de resolver cuando se diseña una BDD.
- Cada fragmento está contenido en diferentes sitios, localidades o servidores de una red interconectada de computadoras.
- Cada fragmento puede estar replicado.

Fragmentación y replicación de información

Fragmentar consiste en dividir la BD en porciones, con la finalidad de ubicar cada porción más cerca, en términos computacionales, de los usuarios que requieren esos datos. En todo momento, debe ser posible obtener la BD original, a partir de realizar operaciones de AR sobre los fragmentos generados.

Existen dos fragmentaciones diferentes. La fragmentación horizontal se aplica sobre una tabla separando en varios fragmentos las tuplas que la componen. Suponga que se tiene una tabla cliente con todos los clientes de una compañía. Esta compañía decide instalar un servidor de datos en cada una de las tres ciudades donde opera: Buenos Aires, Córdoba y Rosario. Por una cuestión operativa, se decide fragmentar la tabla clientes en tres partes, cada una de las cuales contiene los clientes de cada ciudad. La fragmentación horizontal genera *N*porciones de una tabla, y cada porción contendrá tuplas de la tabla original. Así:

Tabla original = F1 U F2 U F3 U ... U Fn

El otro tipo de fragmentación se denomina vertical. En este caso, una tabla se descompone separando los atributos en diferentes porciones. Suponga que se define una tabla con datos personales y laborales de los empleados de una compañía. Se coloca un servidor de datos para el área contable y otro para el área de Personal. Se decide fragmentar la tabla empleados dejando en un fragmento los datos personales de cada individuo, y en otro fragmento, los datos laborales. Así, si f1, f2, ..., fn conforman *n* fragmentos en los cuales se dividió verticalmente la tabla original, entonces:

Tabla original = f1 |×| f2 |×| f3 |×| ... |×| fn

Debe notarse que es necesario que cada fragmento tenga una copia de la CP de la tabla.

La fragmentación de la información refleja la estructura organizativa de la compañía, ubicando los datos cercanos a los usuarios.

Una réplica es una copia de una BD o de una porción de la BD. Replicar una BD consiste en generar copias idénticas de los datos. *A priori*, se puede suponer que el concepto de replicación atenta contra la integridad de los datos.

El proceso de normalización sobre una BD fue definido para evitar repeticiones innecesarias de información, que pueden llevar a pérdidas de consistencia. El problema aquí es no confundir el concepto de repetición con el concepto de replicación. En este último caso, replicar un dato consiste en generar una copia idéntica con la finalidad de:

- **Aumentar la disponibilidad de la información:** si una BD está replicada en diferentes servidores y se produce un fallo en alguno de ellos, es posible continuar el procesamiento. Suponga que ciertos datos están ubicados en el servidor 1 y el servidor 2. Una transacción intenta recuperar la información del servidor 1, y este no responde porque no se encuentra operativo; dicha transacción aún tiene disponible el servidor 2 para recuperar el dato. Este tipo de alternativa no está presente en sistemas centralizados.
- **Aumentar la fiabilidad del sistema:** cuando aumenta la disponibilidad de la información, se mejora la fiabilidad del sistema global. Al estar los datos más disponibles, se disminuye la probabilidad de que una transacción no pueda ser resuelta en el momento en que esta se genera.
- **Aumentar el paralelismo:** si una BD está replicada, aumenta la probabilidad de que un cliente acceda localmente a un dato.

Sin embargo, tener más réplicas de la BD presenta la desventaja de una sobrecarga en las actualizaciones. Si un dato está replicado *n* veces y es modificado por una transacción, las *n* réplicas deben actualizarse.

El esquema de fragmentación y replicación es el punto más crítico que se debe discutir en el diseño de una BDD. Desde un punto de vista de diseño conceptual, lógico y físico del modelo de datos, no hay diferencias entre un esquema centralizado y un esquema distribuido. Esto significa que los conceptos vertidos en este libro son aplicables para el diseño tanto de una BD centralizada como de una BDD. Sin embargo, una vez finalizado el esquema físico, una BDD necesita discutir el esquema de fragmentación y replicación de sus datos.

Ventajas y desventajas de las BDD y los SGBDD

Dentro de las ventajas de la administración de BDD, están los siguientes conceptos:

- **Economía:** es más barato generar redes de servidores tipo PC que las alternativas anteriores de instalar equipos *mainframes*.
- **Crecimiento modular:** siempre es posible agregar más servidores a la red de computadoras instaladas y distribuidas a lo largo de una red de área global.
- **Integración:** los datos están esparcidos en diversos servidores. Esto evita la centralización, y, por consiguiente, los puntos únicos de fallo y cuellos de botella en el procesamiento.
- **Capacidad de generar un nuevo tipo de negocio:** *e-business*.

Las desventajas más significativas que presentan los entornos distribuidos son las siguientes:

- **Potenciales problemas de seguridad inexistentes en entornos centralizados:** la tecnología de red necesaria para el soporte distribuido muestra características que pueden generar nuevos puntos de fallo en la seguridad. Por ejemplo, la captura de paquetes de mensajes entre servidores en la red global.
- **Control de integridad más complejo:** los datos se encuentran replicados, por lo que las modificaciones requieren actualizar todas las réplicas. El procesamiento transaccional es más complejo.
- **Diseño de BD más complejo:** esto es así dado que se agregan dos etapas adicionales, es decir, el diseño del esquema de fragmentación y el diseño del esquema de replicación.

Los conceptos vertidos sobre BDD solamente tienen la intención de presentar someramente el problema. En la bibliografía, pueden encontrarse referencias que desarrollan en profundidad el tema.

Bases de datos orientadas a objetos

El modelo de datos **Orientado a Objetos (OO)** representa el modelo más reciente dentro de las BD. La razón básica detrás del surgimiento de las BDOO y los SGBDOO es la explosión que ha sufrido la ISOO.

Las BD relacionales tradicionales son difíciles de utilizar en aplicaciones OO. Las características propias de los objetos (herencia, polimorfismo, encapsulamiento) no se pueden representar fácilmente en el modelo relacional. El modelo de datos OO se crea para cubrir estas falencias.

Este modelo presenta un conjunto de características que le permiten cumplir con requisitos de usuario y de diseño, de una forma mucho más natural y sencilla para el diseñador, el programador y el usuario de la BD. El diseñador no se limita a definir la estructura del modelo, sino que, además, tiene la posibilidad de definir las operaciones que cada objeto puede llevar a cabo.

El modelo de datos OO es una extensión del paradigma de programación OO. Los objetos entidad que se utilizan en los programas OO son análogos a las entidades que se utilizan en las BDOO puras, pero con una gran diferencia: los objetos del programa desaparecen cuando el programa termina su ejecución, mientras que los objetos de la BD permanecen. A esto se lo denomina persistencia.

En la década de 1990, un grupo de representantes de la industria de las BD formaron el **ODMG (Object Database Management Group)**, con el propósito de definir estándares para los SGBDOO. Este grupo propuso un modelo estándar para la semántica de los objetos de una BD. Los principales componentes de la arquitectura ODMG para un SGBDOO son los siguientes:

- Modelo de objetos.
- Lenguaje de definición de objetos.
- Lenguaje de consulta de objetos.
- Conexión con los lenguajes OO.

El modelo de objetos ODMG permite que tanto los diseños como las implementaciones sean portables entre los sistemas que lo soportan. Dispone de las siguientes primitivas de modelado:

- Los componentes básicos de una BDOO son los objetos y los literales. Un objeto es una instancia de una entidad de interés del mundo real; los objetos tienen un identificador. Un literal es un valor específico; los literales no tienen identificadores. Un literal no debe ser necesariamente un solo valor, sino que puede ser una estructura o un conjunto de valores relacionados que se guardan bajo un solo nombre.
- Los objetos y los literales se categorizan en tipos. Cada tipo tiene un dominio específico compartido por todos los objetos y literales de ese tipo. Los tipos también pueden tener comportamientos. Cuando un tipo tiene comportamientos, todos los objetos de ese tipo comparten los mismos comportamientos. En el sentido práctico, un tipo puede ser una clase de la que se crea un objeto, una interfaz o un tipo de datos para un literal (por ejemplo, integer). Un objeto se puede pensar como una instancia de un tipo. Lo que un objeto sabe hacer son sus operaciones. Cada operación puede requerir datos de entrada (parámetros de entrada) y puede devolver algún valor de un tipo conocido.
- Los objetos poseen propiedades, que incluyen sus atributos y las relaciones que tienen con otros objetos. El estado actual de un objeto está dado por los valores actuales de sus propiedades.
- Una BD es un conjunto de objetos almacenados, que se gestionan de modo que puedan ser accedidos por diversos usuarios y aplicaciones.
- La definición de una BD está contenida en un esquema que se ha creado mediante el lenguaje de definición de objetos ODL, que es el lenguaje de manejo de datos que se ha definido como parte del estándar propuesto para las BDOO.

ODL (Object Definition Language, por sus siglas en inglés, o **Lenguaje de Definición de Objetos)** es un lenguaje de especificación para definir tipos de objetos para sistemas compatibles con ODMG. ODL es el equivalente del DDL de los SGBD relacionales. ODL define los atributos y las relaciones entre tipos, y especifica la signatura de las operaciones.

OQL (Object Query Language, por sus siglas en inglés, o **Lenguaje de Consulta de Objetos)** es un lenguaje declarativo del tipo de SQL que permite realizar consultas de modo eficiente sobre BDOO, incluyendo primitivas de alto nivel para conjuntos de objetos y estructuras. Está basado en SQL-92, proporcionando un superconjunto de la sintaxis de la sentencia SELECT.

OQL no posee primitivas para modificar el estado de los objetos, ya que las modificaciones se pueden realizar mediante los métodos que estos poseen.

La sintaxis básica de OQL es una estructura SELECT...FROM...WHERE..., como en SQL. Por ejemplo, la siguiente expresión obtiene los nombres de las carreras de la Facultad de Informática:

```
SELECT d.nombre
FROM d in carreras
WHERE d.facultad = 'Informática' ;
```

En las consultas se necesita un punto de entrada, que suele ser el nombre de un objeto persistente. Para muchas consultas, el punto de entrada es la extensión de una clase. En el ejemplo anterior, el punto de entrada es la extensión carreras, que es un objeto colección de tipo set<Carreras>. Cuando se utiliza una extensión como punto de entrada, es necesario utilizar una variable de iteración que tome valores en los objetos de la colección. Para cada objeto de la colección (solo la forman objetos persistentes) que cumple la condición (que es de la Facultad de Informática), se muestra el valor del atributo nombre.

Data warehouse y data marts

La definición literal del concepto de **Data Warehouse (DW)** es "almacén o repositorio de datos". Sin embargo, es posible definir a un DW como:

- Un lugar donde las personas pueden acceder a sus datos. Se entiende como dato toda la información generada mediante el uso de computadoras a lo largo de la historia informática de una empresa, no solamente la contenida en el sistema transaccional actualmente en uso.

- Una BD que contiene los datos históricos producidos por los sistemas informáticos de la empresa.
- Son datos administrados que están situados fuera y después de los sistemas operacionales. Esto significa que un DW no representa la BD operacional de la empresa. Un DW no se utiliza para facturación o liquidación de sueldos. Los sistemas transaccionales no utilizan un DW.
- Se remite al adecuado manejo de la información histórica generada por una organización, a fin de que esta pueda otorgarles a sus ejecutivos el soporte adecuado para el proceso de toma de decisiones.

Resumiendo, un DW es netamente un tema de negocios. Consiste en generar una BD, con información que se utilizará para la toma de decisiones en una organización. La finalidad básica de un DW es integrar toda la información generada por sistemas transaccionales de una organización en un único repositorio, que permita luego obtener los datos para la toma de decisiones de dicha organización.

Los objetivos de un DW se resumen en los siguientes:

- Brindar acceso a los datos históricos de manera eficiente, fácil, consistente y confiable.
- Encontrar relaciones entre los datos producidos por diferentes áreas, determinando patrones.
- Identificar nuevas oportunidades de negocio.
- Monitorear las operaciones actuales de negocio y compararlas con las operaciones efectuadas previamente.
- Aumentar la productividad de la empresa.
- Separar y combinar los datos por medio de toda posible medida en el negocio. Por ejemplo, para análisis del problema en términos de dimensiones.

Si el DW no logra transformar los datos en información, para hacerla accesible a los usuarios en un tiempo lo suficientemente rápido como para marcar una diferencia, entonces el DW no sirve.

La siguiente tabla presenta una comparación entre el mundo transaccional y el mundo analítico representado por un DW.

	Mundo operacional (transacción)	Mundo analítico (DW)
En cuanto a objetivos	Mantienen la empresa en operación. Las operaciones realizadas consisten en altas, bajas, modificaciones y consultas sobre la BD.	Brindan lo necesario para el planeamiento y la toma de decisiones.
En cuanto a requisitos	Necesidad de *performance*. Las consultas deben ser resueltas dentro de los plazos establecidos en los requerimientos del problema.	Necesidad de flexibilidad y alcance amplio para el proceso de la información. El tiempo de respuesta no es crítico.
En cuanto a datos almacenados	Son para propósito operacional. Los datos contenidos son volátiles, se modifican a lo largo del tiempo (ejemplo: *stock* de productos). Contienen datos de períodos acotados de tiempo por cuestiones de *performance*. Los datos están encriptados por restricciones del lenguaje o BD (normalizaciones).	Son para propósito analítico. Los datos no son volátiles. Una vez que se ingresa un dato al DW, no es quitado ni modificado. Pueden mantenerse por tiempo indefinido, lo que no implica más costo. Los datos se hallan expresados en términos simples de negocio.
	Los datos giran alrededor de aplicaciones y procesos. No siempre están integrados (aunque deberían). Los datos deberían estar normalizados. Generalmente no hay necesidad de hacer referencia al tiempo en las estructuras claves.	Los datos giran alrededor de los sujetos de la empresa. Están fuertemente integrados para que el análisis tenga sentido. Los datos pueden no estar normalizados. Toda estructura clave contiene un elemento de tiempo. Puede tener datos externos (mercado, tendencia).
En cuanto a operaciones/ transacciones	Las transacciones están predefinidas. Consultas tradicionales SQL.	Las transacciones son difíciles de predecir, nunca se sabe qué se pedirá. Operaciones: carga inicial y acceso a datos (no modificaciones). Análisis multidimensional para consultas.

Los *data marts* están directamente relacionados con los DW. Un **Data Mart (DM)**, a diferencia de un DW, solo satisface necesidades específicas de un área de una organización. En una empresa pueden coexistir varios DM; por ejemplo, cada departamento puede tener el suyo.

Conceptualmente, un DM y un DW tienen la misma estructura. La diferencia está en que un DW representa el resumen de información de toda la organización, en tanto que un DM solamente resume un área o departamento de dicha organización.

La siguiente tabla sintetiza las diferencias básicas entre ambos conceptos.

DM	DW
Enfocado a una sola área.	Contiene información de toda la organización.
Menor costo de construcción.	Mayor costo de construcción.
Puede existir más de un DM.	Existe solo uno en toda la organización.
Fácil de entender y acceder.	Más complejo.
Mejor tiempo de acceso y respuesta.	Menor *performance* en cuanto al tiempo.
A mayor cantidad de DM, la administración es más costosa.	Administración más eficiente.
Peligro de crear "islas" difíciles de integrar.	Integrado por naturaleza.
Peligro de duplicación de datos en varios DM.	No existe duplicación de datos.

Para finalizar el concepto de DW, se mencionan algunas de las ventajas obtenidas a partir de su utilización:

- Ayuda a soportar el proceso de toma de decisiones.
- Sirve para la identificación de nuevas oportunidades en el mercado.
- Logra estructuras corporativas flexibles (distintos departamentos en una organización comparten la misma información).
- Permite la generación de informes críticos sin costo de tiempo.
- Tiene un formato de datos consistente.
- Sirve de plataforma efectiva de mezclado de datos desde varias aplicaciones corrientes.
- Brinda la posibilidad de encontrar respuestas más rápidas y más eficientes a las necesidades del momento.
- Permite la reducción de costos globales para la organización.

Data mining

El propósito de los DW es servir como soporte a la toma de decisiones en una organización. Para lograrlo, una de las técnicas desarrolladas es el *data mining* o minería de datos, cuya finalidad es la extracción no trivial de información implícita, previamente desconocida y potencialmente útil, contenida en un DW o una BD operacional.

La minería de datos se refiere a la extracción o el descubrimiento de nueva información, en términos de patrones o reglas procedentes de grandes cantidades de datos.

En general, los datos ocultan desviaciones, tendencias o anomalías que se pueden descubrir a partir del uso de reglas de inducción o redes neuronales.

Los objetivos de la minería de datos pueden resumirse en los siguientes:

- **Clasificación:** organizar los datos en diferentes grupos basados en combinación de características. Ejemplos: productos de venta estacional, productos de venta por determinadas condiciones de mercado, productos de venta por periodos, productos cuyas ventas aumentan o disminuyen a partir de la venta de otros productos, etcétera.
- **Reconocimiento:** identificar patrones específicos que permiten determinar la existencia de un evento o actividad. Por ejemplo, el estudio del ADN humano, para reconocer características de los genes que permitan identificar una enfermedad.
- **Optimización:** servir de soporte para la optimización de recursos en una organización. Por ejemplo, en un sistema de apoyo a la producción, las técnicas de *data mining* pueden ayudar a decidir la mejor planificación de producción en función de la demanda de artículos.
- **Predicción:** pronosticar cómo se comportarán determinados eventos en función de lo que sucedió en el pasado. En el estudio del clima, la minería de datos puede ser un soporte interesante comparando patrones de ocurrencia del pasado, y puede determinar cómo se comportará un evento en el futuro.

A modo de ejemplo, se hace referencia a una de las principales técnicas dentro de la minería de datos. Se trata de las reglas de asociación, que permiten establecer una correlación entre la presencia de un conjunto de elementos, con otro rango de valores para otro conjunto de variables. Por ejemplo, se puede encontrar un patrón que indique que quien compra cerveza en un supermercado también compra maníes.

Asociados con las reglas de asociación existen dos conceptos: confianza y soporte. El soporte indica el porcentaje de tuplas que mantienen la correlación. Por ejemplo, el porcentaje de ventas donde se llevaron cerveza y maníes. Un alto porcentaje significa que la mayoría de la gente que lleva cerveza lleva maníes, y, por consiguiente, la regla tiene un soporte adecuado. En su defecto, no existiría una evidencia concreta de que la acción ocurra.

Pero no solamente el soporte es importante. Suponga que en ciertas ventas realizadas se puede determinar que los clientes que llevaron tornillos también compraron agua mineral. El soporte de la regla es de 80%; esto significa que de cada 10 compras de tornillos, en ocho compras también se compró agua mineral. El dato puede ser muy representativo. Pero suponga ahora que solamente 0,1% de las ventas involucraron tornillos. Si bien la regla tiene un soporte importante, no tiene la confianza suficiente. La regla encontrada no es importante porque la venta de tornillos no es representativa con respecto al total de ventas. Esto lo establece el nivel de confianza.

Las reglas deben tolerar un umbral de soporte y confianza especificado por el usuario.

El problema a enfrentar es generar todos los conjuntos de elementos que tengan un soporte que exceda un umbral, y luego, para esas reglas, revisar un mínimo de confianza.

El procesamiento de la información pasa a ser el punto más crítico. Se genera un gran volumen de datos; con *M* elementos se pueden crear 2^m conjuntos diferentes para analizar.

Otras aplicaciones

Hay una serie de nuevas tendencias o aplicaciones en lo que a BD y SGBD se refiere. Aquí se presentan, a modo de ejemplo, tres aristas que pueden ser consideradas como ejemplos de estas tendencias.

En primer término, con el uso de la navegación satelital (sistemas de posicionamiento global), los sistemas de información geográfica se han difundido ampliamente. A esta difusión ha contribuido también la utilización para ayuda en catástrofes (inundaciones e incendios, entre otros eventos) o para proporcionar información catastral o económica (por ejemplo, predicción de cosechas y estudio de servicios públicos).

La generación de bibliotecas virtuales hace necesaria la administración de grandes volúmenes de información textual. Las BD textuales representan otro ejemplo de las nuevas tendencias en BD.

Por último, se discuten brevemente las BD móviles. La tendencia más reciente en BD es aquella donde los usuarios se conectan y desconectan automáticamente desde localizaciones móviles, utilizando para ello Internet y, generalmente, dispositivos móviles como *notebooks* o PDA. Este tipo de conexión presenta un nuevo desafío en el campo de las BDD.

Sistemas de información geográfica

Las BD geográficas básicamente están diseñadas para recoger, modelar, almacenar y procesar información cartográfica o topográfica.

Los mapas pueden proporcionar información muy variada: límites, ríos, lagos, carreteras, pero además información mucho más detallada con cada elemento; por ejemplo, la elevación de una montaña, el tipo de vegetación, las precipitaciones, la cantidad de habitantes de una ciudad, etcétera.

Un **GIS (Geographic Information System**, por sus siglas en inglés, **o Sistema de Información Geográfica)** se utiliza básicamente para aplicaciones vinculadas con el uso de mapas (cartografía), análisis digital del suelo y aplicaciones con objetos geográficos. Estas aplicaciones abarcan la representación de mapas ruteros, el estudio de riegos, el control de inundaciones, la distribución y utilización de servicios públicos y las redes de energía, entre otras.

Las aplicaciones de tipo cartográfico necesitan utilizar mapas de múltiples capas. Cada capa permite representar características diferentes del terreno. Por ejemplo, una capa representa el tipo de suelo; otra, la red vial; la siguiente, la red ferroviaria; etcétera.

Para modelar los datos en un GIS, existen dos modelos esenciales: el *raster* y el vectorial. El modelo *raster* consiste en un mapa de bits o píxeles en dos o más dimensiones. Cada píxel guarda una información en particular. Así, por ejemplo, en un plano se tiene representada una región de un mapa, en la cual los bits con valor 1(uno) indican los lugares por donde pasa un río, y el resto de los bits está en cero.

El modelo vectorial se genera a partir de objetos geométricos elementales: puntos, segmentos, triángulos y otros polígonos. Los datos de mapas suelen representarse de esta forma. Una corriente de agua se representa como segmentos, un lago se representa como un polígono.

El **GPS (Global Positioning System**, por sus siglas en inglés, o **Sistema de Posicionamiento Global**) utiliza la tecnología GIS, para la determinación del camino más corto o rápido desde un lugar hacia otro.

Bases de datos textuales

Las BD textuales surgen a partir de la necesidad de contar con bibliotecas digitalizadas. Los libros, revistas y publicaciones en general, y científicas en particular, necesitan estar digitalizados y disponibles para los usuarios de los catálogos digitales. Esto lleva a un nuevo problema en lo que a BD se refiere. El almacenamiento de grandes volúmenes de texto (un libro completo, los anales de un congreso o la colección de revistas de un tema) no representa en sí un problema. Se dispone en la actualidad de suficiente capacidad de almacenamiento para administrar esta información. Sin embargo, el acceso a dicha información debe ser rápido y fácil para cualquier usuario.

Las técnicas para la obtención (o digitalización) de la información no son triviales. Además, se debe tener en cuenta la gran diversidad de formatos existentes. Se debe comprender que no basta con escanear una hoja de un libro o una revista. La finalidad de las BD textuales no consiste en almacenar una copia de una imagen, sino en digitalizar el contenido de esta última.

Luego, debe considerarse la recuperación de la información. Las técnicas para buscar un dato específico en una biblioteca digital deben asumir que el usuario puede desear consultar aquellos textos donde aparezca una palabra específica.

Un ejemplo interesante para analizar es el producto desarrollado por la UNESCO (United Nations Educational, Scientific and Cultural Organization) denominado CDS/ISIS, consistente en un SGBD que permite el almacenamiento y la recuperación de información textual.

Bases de datos móviles

La utilización cada vez más difundida de equipos móviles como computadoras personales portátiles, teléfonos celulares y PDA, y el desarrollo de infraestructura de bajo costo de comunicación sin cables han permitido desarrollar, a su vez, una nueva tecnología en BD que va más allá de los sistemas centralizados y/o distribuidos. Son aquellos sistemas donde parte de la BD se encuentra almacenada en estos

dispositivos y, por ende, los datos allí contenidos no están disponibles todo el tiempo.

La gestión de la BD, el manejo de transacciones, la recuperación en caso de fallos presentan problemas diferentes de los discutidos oportunamente en este libro. Sin embargo, el tratamiento de estos problemas tiene un origen común con las BDD, pero con consideraciones especiales: ancho de banda limitado (generalmente producido por enlaces Wi-Fi), autonomía eléctrica limitada (la mayoría de estos dispositivos funcionan con baterías gran parte del tiempo) y localizaciones cambiantes (con conexión vía telefonía celular, 3G).

La arquitectura de soporte para BD móviles cuenta con servidores y clientes fijos (en una red de área local o una red de área extensa, con una BDD), y unidades móviles conectadas vía inalámbrica, en general distribuida geográficamente.

Desde la visión de la gestión de una BD móvil, hay similitudes con los entornos distribuidos. Puede ocurrir que la BD completa se distribuya entre los servidores fijos, o que parte de la distribución esté en un dispositivo móvil. En este último caso, el manejo transaccional se verá afectado. Los datos contenidos en el dispositivo móvil no se encuentran disponibles todo el tiempo, ya sea para consulta o actualización.

Una transacción en un ambiente móvil se ejecuta en serie. Puede no existir una coordinación con el resto de los sitios. Una transacción generada localmente en una computadora móvil sin conexión necesitará acceder a una red antes de actualizar la BD completa. Esto puede ocasionar problemas de consistencia en la información. En muchos casos, estos problemas solamente pueden ser resueltos con intervención humana.

Bibliografía

ABITEBOUL, SERGE; RICHARD HULL Y VICTOR VIANU, *Foundations of Databases*, Reading (Mass.), Addison-Wesley, 1995.

BATINI, CARLO; STEFANO CERI Y SHAMKANT NAVATHE, *Diseño conceptual de bases de datos: un enfoque de entidades-interrelaciones*, Wilmington (Del.), Addison-Wesley/Díaz de Santos, 1994.

CHEN, PETER, "The Entity-Relationship Mode. Toward a Unified View of Data", *ACM TODS*, 1:1, marzo de 1976.

CODD, EDGAR FRANK, "A Relational Model of Data for Large Shared Data Banks", *CACM*, 13:6, junio de 1970.

"A Data Base Sublanguage Founded on the Relational Calculus", *Proceedings of the ACM SIGFIT Workshop on Data Control*, noviembre de 1971.

"Further Normalization of the Data Base Relational Model", en Randall Rustin (ed.), *Data Base Systems (Courant Computer Science Symposia 6)*, Englewood Cliffs (NJ), Prentice Hall, 1972.

"Relational Database. A Practical Foundation for Productivity", *CACM*, 25:2, diciembre de 1982.

The Relational Model for Database Management, Menlo Park (Ca.), Addison-Wesley, 1990.

CONNOLLY, THOMAS Y CAROLYN BEGG, *Sistemas de bases de datos: un enfoque práctico para diseño, implementación y gestión*, Madrid, Pearson Educación, 2005.

DATE, CHRISTOPHER J., *Introducción a los sistemas de bases de datos*, México, Pearson Educación, 2001.

DATE, CHRISTOPHER J. Y HUGH DARWEN, *A Guide to SQL Standard*, Addison-Wesley, 1993.

DE GIUSTI, ARMANDO Y OTROS, *Algoritmos, datos y programas con aplicaciones en Pascal, Delphi y Visual Da Vinci*, Pearson Educación, 2005.

ELMASRI, RAMENZA Y SHAMKANT NAVATHE, *Fundamentos de sistemas de bases de datos*, Pearson Education, 2003.

Folk, Michael y Bill Zoellick, *Estructuras de archivos: un conjunto de herramientas conceptuales*, Wilmington (Del.), Addison-Wesley, 1992.

Kroenke, David, *Procesamiento de bases de datos. Fundamentos, diseño e implementación*, México, Pearson Educación, 2003.

Mannino, Michael, *Administración de bases de datos. Diseño y desarrollo de aplicaciones*, México, McGraw-Hill, 2007.

Morteo, Francisco; Nicolás Bocalandro; Cristian Cascon y Hernán Cascon, *Fundamentos de diseño y modelado de datos*, Cooperativas, 2007.

Ozsu, M. Tamer y Patrick Valduriez, *Principles of Distributed Database Systems*, Englewood Cliffs (NJ), Prentice Hall, 1999.

Piattini, Mario; Esperanza Marcos; Coral Calero y Belén Vela, *Tecnología y diseño de bases de datos*, México, Alfaomega Grupo Editor, 2007.

Pons, Olga; Nicolás Marín; Juan Miguel Medina; Silvia Acid y María Amparo Vila, *Introducción a las bases de datos. El modelo relacional*, Paraninfo, 2005.

Ramakrishnan, Raghu y Johannes Gehrke, *Sistemas de gestión de bases de datos*, Madrid, McGraw-Hill, 2007.

Silberschatz, Abraham; Henry Korth y S. Sudarshan, *Fundamentos de bases de datos*, Madrid, McGraw-Hill, 2006.

Smith, Peter D. y G. Michael Barnes, *Files & Databases. An Introduction*, Reading (Mass.), Addison-Wesley, 1987.

Índice temático

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V