UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
"ARAGON"

INGENIERIA EN COMPUTACION

FUNDAMENTOS, MODELOS Y TENDENCIAS DE BASES DE DATOS

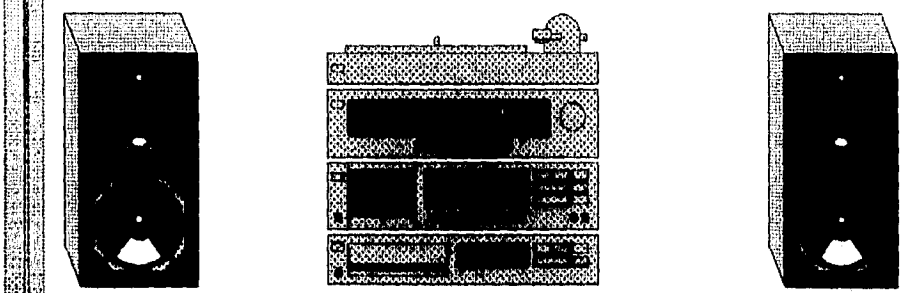
T E S I S P R O F E S I O N A L
QUE PARA OBTENER EL TITULO DE
INGENIERO EN COMPUTACION
P R E S E N T A :

GERARDO GUTIERREZ RESENDIZ

ASESOR DE TESIS
ING. ERNESTO PEÑALOZA ROMERO.

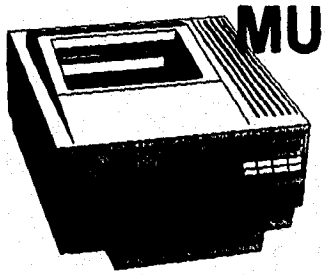
SAN JUAN DE ARAGON, EDO. DE MEXICO 1996

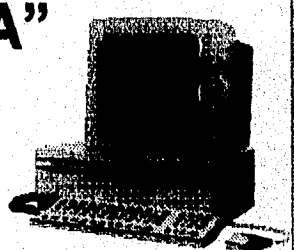
TESIS CON
FALLA DE ORIGEN



FUNDAMENTOS, MODELOS Y TENDENCIAS DE BASES DE DATOS

Y EL SISTEMA "CD VIRTUAL MULTIMEDIA"





Dedico esta Tésis:

A la memoria de mi Padre, Dimas Gutiérrez.

Con amor y agradecimiento a mi Madre, Ma Teresa Reséndiz.

A mi hermana, Ma Mayela Gutiérrez.

A mi futura esposa, María del Carmen Soto, te agradezco el que me hallas facilitado equipo y material para la realización de este trabajo, ya que sin él nada de esto se habría hecho realidad.

A los profesores, quienes tuvieron la dedicación y paciencia de revisar este trabajo y hacerme saber sus obervaciones y sugerencias.

A las siguientes personas, quienes de alguna u otra forma han tenido algo que ver con esta tésis:

Miguel Angel Morales
Eduardo Rueda
Francisco Ugalde
Alejandro Tapia

Al Creador del Universo.

PREFACIO

Esta tésis introduce los conceptos fundamentales necesarios para diseñar, usar, e implementar sistemas de bases de datos. Poniéndo énfasis en los fundamentos del modelado y diseño, los lenguajes y las facilidades proporcionadas por los DBMSs, así como las técnicas de implementación de sistemas.

Las bases de datos y su tecnología están teniendo un mayor impacto con el creciente uso de las computadoras. Es acertado decir que jugarán un papel crítico en casi todas las áreas donde se utilicen sistemas de cómputo, incluyendo negocios, ingeniería, medicina, leyes, educación, y bibliotecas de ciencia, por nombrar algunas.

La Primera Parte describe los conceptos básicos necesarios para un buen entendimiento del diseño e implementación de bases de datos. Se introducen las bases de datos, sus usuarios típicos, conceptos y arquitectura de DBMSs. Se presentan los conceptos del modelo Entidad-Relación (ER) y se usa a este para ilustrar el diseño conceptual de bases de datos. Se describen los métodos primarios de organización de archivos de registros en disco, incluyendo el corte estático y dinámico, las técnicas de indexado de archivos, etc.

La Segunda Parte describe el modelo de datos relacional. Se describe el modelo básico relacional, sus contensiones de integridad y operaciones de actualización, y las operaciones del álgebra relacional. También se incluye una sección que describe el diseño de un esquema relacional a partir de un diagrama conceptual ER.

La Tercera Parte presenta el así llamado legado de los sistemas de bases de datos, sistemas de red y jerárquicos. Estos han sido utilizados como base de muchas aplicaciones comerciales existentes, particularmente en grandes sistemas de transacción y procesamiento. Cada modelo se describe independientemente de DBMSs específicos. Una sección especial muestra cómo convertir el diseño conceptual de un esquema de bases de datos del modelo ER a un esquema de red o jerárquico.

La Cuarta Parte cubre varios temas relacionados al diseño de bases de datos. Primero, se cubren los formalismos, teoría, y algoritmos desarrollados para el diseño de bases de datos relacionales por normalización. Se presenta un panorama de las diferentes fases del proceso de diseño de bases de datos para aplicaciones de tamaño medio y grande, también se discuten temas físicos del diseño pertinentes a DBMS's relacionales, de red y jerárquicos.

La Quinta Parte define el papel del Catálogo del Sistema, el concepto de transacciones, los problemas del control de concurrencia así como las técnicas empleadas para evitar que dichas transacciones arrojen resultados erróneos, la Seguridad en las Bases de Datos la cual es crítica para mantener la integridad de la información y las técnicas de recuperación empleadas cuando ocurren fallas en la base de datos.

La Sexta Parte cubre un número de temas avanzados. Se da una introducción global a las bases de datos orientadas a objetos, dándose ejemplos de un sistema comercial. Se discuten las bases de datos distribuidas y la arquitectura cliente-servidor. Con poderosas estaciones de trabajo y redes de comunicaciones de alta velocidad, las verdaderas bases de datos distribuidas se vuelven viables. Se discuten las tendencias en la tecnología de bases de datos y sus aplicaciones. La siguiente generación tecnológica incluye bases de datos activas, temporales, espaciales, científicas y multimedia. Las aplicaciones emergentes incluyen diseño de ingeniería y manufactura, sistemas de soporte de oficina y toma de decisiones, y aplicaciones biológicas.

Finalmente, la Séptima Parte como aplicación a toda esta teoría, se ha desarrollado la Base de Datos Multimedia "Virtual CD Land System", la cual combina imágenes, música y datos, a través del DBMS Paradox for Windows, el cual puede catalogarse como uno de los DBMSs relacionales que está agregando características de OO.

FUNDAMENTOS, MODELOS Y TENDENCIAS DE BASES DE DATOS

C O N T E N I D O

PARTE II MODELO RELACIONAL, LENGUAJES Y SISTEMAS

PARTE III MODELOS Y SISTEMAS DE DATOS CONVENCIONALES

PARTE VII APLICACION

TESIS

COMPLETA

CAPITULO 1

Bases de Datos y Usuarios de Bases de Datos

1.1 Introducción

El mundo de las *bases de datos* es tan común que empezaremos por definir lo que es una base de datos. Nuestra definición inicial es algo general.

Una **base de datos** es una colección de información relacionada. Por **datos** queremos decir hechos conocidos que pueden ser registrados y que tienen un significado implícito. Por ejemplo, consideremos los nombres, números telefónicos, y direcciones de la gente que conocemos. Se puede haber registrado esta información en una agenda indexada, o quizá en un diskette, usando una PC y software como dBASE, Paradox o EXCEL. Esta es una colección de información relacionada con un significado implícito y por lo tanto es una base de datos.

Una base de datos tiene las siguientes propiedades implícitas:

- Representa algunos aspectos del mundo real, algunas veces llamado el **minimundo** o el **Universo de Disertación (UoD)**. Los cambios en el minimundo se reflejan en la base de datos.

- Es una colección coherente de datos con algún significado inherente. Una clasificación aleatoria de datos no puede ser correctamente referida como una base de datos.

- Se diseña, se construye, y popula con datos para un propósito en específico. Tiene un grupo inherente de usuarios y algunas aplicaciones preconcebidas en las cuales estos usuarios se interesan.

En otras palabras, una base de datos tiene alguna fuente de la cual se derivan los datos, algún grado de interacción con los eventos del mundo real, y una audiencia que está activamente interesada en su contenido.

Una base de datos puede ser de cualquier tamaño y variante complejidad. Por ejemplo, la lista de direcciones referida anteriormente puede consistir de algunos cientos de registros, cada uno con una simple estructura. Por otro lado, el catálogo de fichas de una biblioteca puede constar de medio millón de estas almacenadas bajo diferentes categorías -por el nombre del autor, por tema, por título- con cada categoría organizada por orden alfabético.

Una base de datos puede generarse y mantenerse manualmente o por máquina. El catálogo de la biblioteca es un ejemplo de base de datos que puede crearse y mantenerse a mano. Una base de datos computarizada puede crearse y mantenerse ya sea por un grupo de programas de aplicación escritos específicamente para esa tarea o por un manejador de bases de datos.

Un **sistema manejador de bases de datos (DBMS)** es una colección de programas que permite a los usuarios crear y mantener una base de datos. El DBMS es por lo tanto

software de *propósito general* que facilita el proceso de definir, construir, y manipular bases de datos para varias aplicaciones. **Definir** una base de datos involucra especificar los tipos de datos, estructuras, y argumentos de los datos que serán almacenados. **Construir** la base de datos es el proceso de almacenar los datos en algún medio que se controla por el DBMS. **Manipular** la base de datos incluye funciones tales como queries para la recuperación de información especifica, la actualización de la base refleja los cambios en el minimundo, y generar reportes de los datos.

Para implementar una base de datos computarizada no es necesario usar DBMS de propósito general. Podríamos escribir nuestros propios programas para crearla y mantenerla, en efecto crear nuestro propio DBMS de *propósito especial*. En cualquier caso, -si usamos un DBMS de propósito general o no- usualmente emplearemos una considerable cantidad de software para manipular la base de datos además de la base misma. A esta base de datos y software lo llamaremos **sistema de base de datos**. La Figura 1.1 ilustra estas ideas.

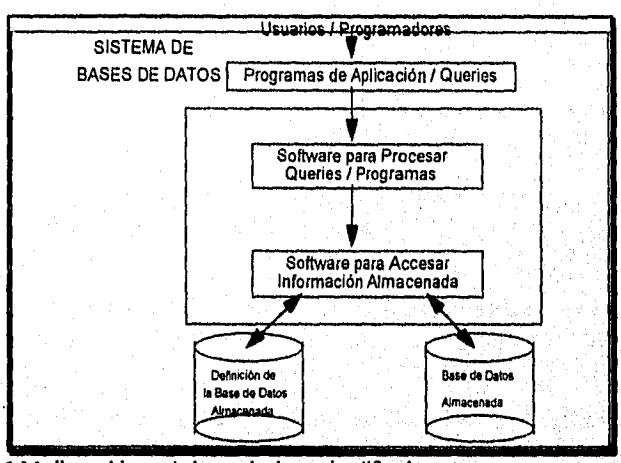


Figura 1.1 Medio ambiente de bases de datos simplificado.

1.2 Un Ejemplo

Consideremos el ejemplo de una base de datos de la UNIVERSIDAD para mantener información que tiene que ver con estudiantes, cursos, y calificaciones de un medio ambiente universitario. La Figura 1.2 muestra la estructura de la base de datos y unos cuantos ejemplos para dicha base. La base de datos se organiza como cinco archivos, cada uno de los cuales almacena registros del mismo tipo. El archivo ESTUDIANTE almacena la información de cada alumno; el archivo CURSO almacena información de cada uno; el archivo SECCION almacena los datos de la sección del curso; el archovo REPO_CAL almacena las calificaciones que los estudiantes reciben en las distintas secciones que han completado; y el archivo PRE-REQUISITO almacena los pre-requisitos de cada curso.

ESTUDIANTE	Nombre	NumEst	Clase	Mayor
	Smith	17	1	CC
	Brown	8	2	CC

CURSO	NomCurso	NumCurso	HrsCredito	Departamento
	Int a Ciencia Comp	CC1310	4	CC
	Estructuras de Datos	CC3320	4	CC
	Matemáticas Discreta	MAT2410	3	MAT
	Bases de Datos	CC3380	3	CC

SECCION	IdSección	NumCurso	Semestre	Año	Instructor
	85	MAT2410	Otoño	91	King
	92	CC1310	Otoño	91	Anderson
	102	CC3320	Verano	92	Knuth
	112	MAT2410	Otoño	92	Chang
	119	CC1310	Otoño	92	Anderson
	135	CC3380	Otoño	92	Stone

REP_CAL	NumEst	IdSección	Cal
	17	112	B
	17	119	C
	8	85	A
	8	92	A
	8	102	B
	8	135	A

PRE-REQUISITO	NumCurso	NumPre-requisito
	CC3380	CC3320
	CC3380	MAT2410
	CC3320	CC1310

Figura 1.2 Ejemplo de una base de datos.

Para *definir* esta base de datos, debemos especificar la estructura de los registros de cada archivo definiendo los diferentes tipos de **elementos de datos** a almacenarse en cada registro. En la Figura 1.2, cada registro de ESTUDIANTE incluye los datos que representan Nombre, NumEst, Clase, y Carrera(MAT, ciencias de la computación, ...); cada registro CURSO incluye datos para representar el NomCurso, NumCurso, HrsCrédito, y Departamento (el departamento que ofrece el curso); etc. También debemos especificar el **tipo de dato** para cada elemento dentro de un registro. Por ejemplo, podemos especificar que el nombre del estudiante es una cadena de caracteres alfabéticos, que el número de estudiante es entero, y que calificación es un caracter del conjunto {A, B, C, D, F, I}. También podemos usar un esquema de codificación para representar un elemento de información. Por ejemplo, en la Figura 1.2 representamos la clase de un estudiante 1 para novato, 2 para estudiante de segundo año, 3 para junior, 4 para semi-graduado, 5 para graduado.

Para *construir* la base UNIVERSIDAD, almacenamos los datos para representar cada estudiante, curso, sección, reporte de calificación, y el pre-requisito como un registro en el archivo apropiado. Nótese que los registros de los distintos archivos pueden relacionarse unos con otros. Por ejemplo, el registro de "Smith" en el archivo ESTUDIANTE se relaciona con dos registros en el archivo REP_CAL que especifica las calificaciones de Smith en dos secciones. Similarmente, cada registro en el archivo PRE-REQUISITO se relaciona con dos registros de CURSO: uno que representa el curso y otro que representa

el pre-requisito. La mayoría de las bases de datos medianas y grandes incluyen muchos tipos de registros y tienen muchas relaciones entre ellos.

La *manipulación* de la base de datos involucra queries y actualizaciones. Ejemplos de queries son "recuperar una lista de todas las calificaciones de Smith"; "listar los nombres de los estudiantes que tomaron la sección de Bases de Datos en el Otoño del 92 y sus calificaciones en esa sección"; y "cuáles son los pre-requisitos del curso de Bases de Datos". Ejemplos de actualizaciones son "cambiar la clase de Smith a segundo año"; "crear una nueva sección para el curso de Bases de Datos de este semestre"; e "ingresar una calificación de A para Smith en la sección de Bases de Datos del semestre anterior". Antes de poder procesarse estos queries y actualizaciones informales deben especificarse precisamente en el lenguaje del sistema de base de datos.

1.3 Características del Acercamiento de Base de Datos

En el **procesamiento de archivos** tradicional, cada usuario define e implementa los archivos necesarios para una aplicación en específico. Por ejemplo, un usuario, de la oficina de reporte de calificaciones, puede guardar en un archivo a los estudiantes y sus calificaciones. Los programas imprimen una copia e ingresan nuevas calificaciones al archivo. Un segundo usuario, de la oficina de contabilidad, puede registrar sus cuotas y pagos. A pesar de que ambos usuarios se interesan en información de estudiantes, cada uno mantiene archivos separados -y los programas mantienen a estos archivos- porque cada uno requiere de información no disponible en los archivos de otros usuarios. Esta redundancia en la definición y almacenamiento resulta en un desperdicio de espacio de almacenamiento y en esfuerzos redundantes para mantener los datos actualizados.

En el acercamiento de bases de datos, se guarda un sólo reposo de los datos en donde se almacenan una vez y luego se accesan a través de varios usuarios. Las principales características del acercamiento de bases de datos contra el de procesamiento de archivos son las siguientes:

1.3.1 Naturaleza Autodescriptiva de un Sistema de Bases de Datos

Una característica fundamental del acercamiento de bases de datos es que el sistema no sólo contiene a la base sino una definición completa o descripción de la misma. La definición se almacena en el **catálogo** del sistema, el cual contiene información tal como la estructura de cada archivo, el tipo y formato de almacenamiento de cada elemento, y varios argumentos en la información. A la información almacenada en el catálogo se le llama **meta-data**, y describe la estructura de la base de datos primaria (Figura 1.1).

El catálogo se utiliza por el software del DBMS y ocasionalmente por los usuarios de la base quienes necesitan información acerca de la estructura. El DBMS no está escrito para ninguna aplicación en específico, y por lo tanto debe referirse al catálogo para conocer la estructura de los archivos de una base en específico, tal como el tipo y formato de la información que se accesará. El software del DBMS debe trabajar igualmente bien con

cualquier número de aplicaciones de bases de datos -por ejemplo, una base de datos universitaria, de un banco, o de una compañía- mientras que la definición de la base se almacene en el catálogo.

En el procesamiento tradicional de archivos, la definición de datos es típicamente parte de los programas de la aplicación. Por tanto, estos programas están restringidos a trabajar con una sola *base de datos* en específico, cuya estructura se declara en los programas de aplicación. Por ejemplo, un programa en PASCAL puede tener registradas estructuras declaradas en él; un programa en C++ puede tener declaraciones "struct" o "class"; y un programa en COBOL tiene instrucciones en la Data Division para definir sus archivos. Cualquiera que sea el software procesador de archivos sólo puede accesar a bases de datos específicas, el software DBMS puede accesar a diversas bases de datos extrayendo las definiciones del catálogo y usando estas definiciones.

En nuestro ejemplo de la Figura 1.2, el DBMS almacena en el catálogo las definiciones de todos los archivos mostrados. Cada que se hace una requisión de acceso, digamos, el nombre de un registro ESTUDIANTE, el software del DBMS se refiere al catálogo para determinar la estructura del archivo ESTUDIANTE y la posición y tamaño del elemento Nombre dentro de un registro de ESTUDIANTE. Por el contrario, en una aplicación típica de procesamiento de archivos, la estructura del archivo y, en algunos casos, la localización exacta del nombre dentro de un registro ESTUDIANTE están ya codificados dentro de cada programa que accesa al elemento.

1.3.2 Aislamiento entre Programas y Datos, y Abstracción de Datos

En el procesamiento tradicional de archivos, la estructura de datos está contenida en los programas de acceso, así cualquier cambio a la estructura de un archivo puede requerir *cambiar todos los programas* que accesan a este archivo. Por el contrario, el DBMS accesa programas que están escritos independientemente de cualquier archivo específico. La estructura de los archivos de datos se almacena en el catálogo del DBMS separadamente de los programas de acceso. A esta propiedad se le llama **independencia de programas y datos**. Por ejemplo, un archivo puede accesar a un programa que pudiera estar escrito de tal forma que sólo pueda accesar al registro ESTUDIANTE de longitud de 42 caracteres (Figura 1.3). Si queremos agregar otro trozo de información a cada registro ESTUDIANTE, digamos la fecha de nacimiento, tal programa no trabajará más y deberá cambiarse. Por el contrario, en un medio ambiente DBMS, sólo necesitamos cambiar la descripción de los registros ESTUDIANTE en el catálogo; no se cambian programas. La próxima vez que un programa DBMS se refiera al catálogo, se accesará a la nueva estructura de los registros ESTUDIANTE.

Nombre Elemento Dato	Posición Inicial en Registro	Longitud en Caracteres (Bytes)
Nombre	1	30
NúmeroEstudiante	31	4
Clase	35	4
Mayor	39	4

Figura 1.3 Formato de almacenamiento de un registro ESTUDIANTE.

El reciente desarrollo de bases de datos orientadas a objetos y lenguajes de programación permiten a los usuarios definir operaciones sobre los datos como parte de la definición de la base de datos. Una **operación** (también llamada *función*) se especifica en dos partes. La *interface* (o *signatura*) de una operación incluye el nombre de la operación y los tipos de datos de sus argumentos (o parámetros). La *implementación* (o *método*) de la operación se especifica separadamente y puede cambiarse sin afectar la interface. Los programas de aplicación pueden operar sobre los datos invocando a estas operaciones a través de sus nombres y argumentos, sin importar cómo se implementan las operaciones. Esto puede llamarse **independencia de operación**.

A la característica que permite la independencia de programas de datos y de programas de operación se le llama **abstracción de datos**. Un DBMS proporciona a los usuarios una **representación conceptual** de la información que no incluye muchos de los detalles de cómo se almacena la información. Informalmente, un **modelo de datos** es un tipo de abstracción de datos que se usa para proporcionar esta representación conceptual. El modelo de datos usa conceptos lógicos, tales como objetos, sus propiedades, y sus interrelaciones, que pueden ser más fáciles de entender para la mayoría de los usuarios que los conceptos de almacenamiento en computadora. Por tanto, el modelo de datos *oculta* los detalles de almacenamiento que no son de interés para la mayoría de los usuarios de bases de datos.

En una aplicación de procesamiento de archivos, cada archivo puede definirse por la longitud de su registro -el número de caracteres (bytes) de cada registro- y cada elemento puede especificarse por su byte inicial dentro de un registro y su longitud en bytes. El registro ESTUDIANTE se representaría así como en la Figura 1.3. Pero a un típico usuario de bases de datos no le interesa dónde se encuentra cada elemento dentro del registro o cuál es su longitud; le preocupa que, cuando se haga referencia a un Nombre de ESTUDIANTE, se retorne el valor correcto. Una representación conceptual de los registros ESTUDIANTES se muestra en la Figura 1.2. Muchos otros detalles de la organización de almacenamiento de archivos -tales como las trayectorias de acceso especificadas en un archivo- pueden estar ocultas a los usuarios por el DBMS.

En el acercamiento de bases de datos, la estructura detallada y la organización de cada archivo se almacenan en el catálogo. Los usuarios se refieren a la representación conceptual de los archivos, y el DBMS extrae los detalles del almacenamiento del archivo del catálogo cuando estos son necesarios por el software del DBMS. Para proporcionar esta abstracción a los usuarios pueden usarse muchos modelos de datos.

Con la reciente tendencia hacia las bases de datos orientadas a objetos, la abstracción se lleva a cabo a un nivel superior para incluir no sólo a la estructura de datos, sino también a las *operaciones* sobre los datos. Estas operaciones proporcionan una abstracción de las actividades del minimundo comúnmente entendido por los usuarios. Por ejemplo, puede aplicarse la operación CALCULA_AVG a un objeto estudiante para calcular el promedio de calificaciones. Tal operación puede invocarse por el programa del usuario sin que conozca los detalles de cómo están implementados internamente. En ese sentido, una

abstracción de la actividad del minimundo se hace disponible al usuario como una **operación abstracta**.

1.3.3 Soporte de Múltiples Vistas de los Datos

Una base de datos típicamente tiene muchos usuarios, cada uno de los cuales puede requerir una diferente perspectiva o **vista**. Una vista puede ser un subconjunto de la base de datos o puede contener datos **virtuales** que se deriven de los archivos pero no se almacenan explícitamente. Algunos usuarios pueden no preocuparse de si la información que ellos necesitan está almacenada o se deriva. Un DBMS multiusuario cuyos usuarios tienen una variedad de aplicaciones deben proporcionar facilidades para definir múltiples vistas. Por ejemplo, un usuario de la base de datos de la Figura 1.2 puede interesarse sólo en la transcripción de cada estudiante; la vista para este usuario se muestra en la Figura 1.4(a). Un segundo usuario, quien sólo se interesa en checar que los estudiantes hayan tomado los pre-requisitos de cada curso al que se registran, puede requerir la vista mostrada en la Figura 1.4(b).

(a)

TRANSCRIBE	NombreEstud					
		NúmeroCurso	Cal	Semestre	Año	IdSección
	Smith	CC1310	C	Otoño	92	119
		MAT2410	B	Otoño	92	112
	Brown	MAT2410	A	Otoño	91	85
		CC1310	A	Otoño	91	92
		CC3320	B	Primavera	92	102
		CC3380	A	Otoño	92	135

(b)

PRE-REQUISITOS	NomCurso	NumCurso	Pre-requisitos
	Bases de Datos	3380	CC3320
			MAT2410
	Estructuras de Datos	3320	CC1310

Figura 1.4 Dos vistas (datos derivados) de la base de datos en la Figura 1.2. (a) La vista transcripción estudiante. (b) La vista curso pre-requisito.

1.4 Actores en Escena

Para una pequeña base de datos personal, tal como la lista de direcciones discutida en la sección 1.1, típicamente una sola persona define, construye y manipula la base de datos. Sin embargo, en las grandes bases de datos, muchas personas están involucradas en su diseño, uso y mantenimiento. En esta sección identificamos a la gente cuyo trabajo involucra el uso diario de una base de datos; los llamaremos "actores en escena" -aquéllos quienes trabajan para mantener el medio ambiente del sistema de base de datos, pero que no están interesados en la base misma.

1.4.1 Administradores de la Base de Datos

En cualquier organización donde muchas personas usan los mismos recursos, es necesario un administrador que observe y maneje estos recursos. En un medio ambiente de bases de datos, el recurso primario es la base misma y el recurso secundario es el DBMS y el software relacionado. Administrar estos recursos es responsabilidad del **administrador de base de datos (DBA)**. El DBA es responsable de autorizar acceso a la base, para coordinar y monitorear su uso, y de adquirir recursos de software y hardware cuando sean necesarios. El DBA es responsable de problemas tales como cuarteaduras de seguridad o tiempo de respuesta del sistema muy pobre. En las grandes organizaciones, el DBA es asistido por un staff que le ayuda a realizar estas funciones.

1.4.2 Diseñadores de Bases de Datos

Son responsables de identificar los datos a almacenar en la base y de escoger las estructuras apropiadas para representar y almacenarlos. Estas tareas se deben encargar antes de que la base sea implementada. Es responsabilidad de los diseñadores comunicarse con todos los prospectos de usuarios, para entender sus requerimientos, y realizar un diseño que cumpla estos requerimientos. En muchos casos, los diseñadores están dentro del staff del DBA y se les asignan otras responsabilidades después de que el diseño está completo. Los diseñadores frecuentemente interactúan con cada grupo potencial de usuarios y desarrollan una **vista** de la base de datos que cumple con los requerimientos de este grupo. Estas vistas entonces se analizan y se integran con las vistas de otros usuarios. El diseño final debe ser capáz de soportar los requerimientos de todos los grupos de usuarios.

1.4.3 Usuarios Finales

Estas son las personas cuyas tareas requieren acceso a la base de datos para consultar, actualizar y generar reportes. Existen varias categorías de usuarios finales:

- **Usuarios casuales** ocasionalmente accesan a la base de datos, pero pueden necesitar información diferente cada vez. Usan un sofisticado lenguaje de query para especificar sus requisiciones y son típicamente gerentes de medio y nivel superior u otros ocasionales.

- **Usuarios ingenuos o paramétricos** conforman una porción medible de usuarios finales. Su tarea principal radica en hacer queries y actualizaciones a la base de datos constantemente; usando tipos estandar de queries y actualizaciones -llamados **transacciones enlatadas**- que han sido cuidadosamente programadas y probadas. Todos estamos acostumbrados a tratar con tales usuarios. En los bancos se chechan balances y depósitos. En líneas aéreas, hoteles, y compañías de renta de autos checan la disponibilidad de una requisición dada y hacen reservaciones. Los despachadores en estaciones de servicio para paquetería haciendo identificaciones vía código de barras e

información descriptiva a través de botones para actualizar una base central de paquetes recibidos y en tránsito.

- **Usuarios sofisticados** incluyen ingenieros, científicos, analistas, y otros quienes se familiarizan con las facilidades del DBMS para cumplir con sus complejos requerimientos.

- **Usuarios independientes** mantienen bases personales usando programas ya hechos que proporcionan menús fáciles de usar o interfaces gráficas. Un ejemplo de ello es el usuario de un paquete de impuestos que almacena una variedad de información financiera personal para propósitos de impuestos.

Un DBMS típico proporciona múltiples facilidades para accesar a una base de datos. Los usuarios finales ingenuos necesitan aprender muy poco acerca de las facilidades proporcionadas por el DBMS; sólo tienen que entender los tipos de transacciones estandar diseñadas e implementadas para su uso. Los usuarios casuales aprenden sólo algunas facilidades que puedan usar repetidamente. Los usuarios sofisticados intentan aprender la mayoría de las facilidades del DBMS para cumplir con sus requerimientos complejos. Los usuarios independientes típicamente se vuelven muy eficientes usando paquetes en específico.

1.4.4 Analistas de Sistemas y Programadores de Aplicaciones

Los **analistas de sistemas** determinan los requerimientos de los usuarios finales, especialmente de los usuarios ingenuos, y desarrollan especificaciones para transacciones enlatadas que cumplan estos requerimientos. Los **programadores de aplicaciones** implementan estas especificaciones como programas; luego prueban, depuran, documentan y mantienen estas transacciones enlatadas. Tanto analistas como programadores deben estar familiarizados con la capacidad total proporcionada por el DBMS para cumplir sus tareas.

1.5 Trabajadores Detrás de Escena

Además de aquéllos que diseñan, usan y administran una base de datos, otros están asociados con el diseño, desarrollo y operación del *software y medio ambiente* del DBMS. Estas personas no se interesan en la base de datos.

1.5.1 Diseñadores e Implementadores del DBMS

Son personas que diseñan e implementan los módulos e interfaces del DBMS como un paquete de software. UN DBMS es un sistema complejo de software de muchos componentes o **módulos**, incluyéndose aquéllos para implementar el catálogo, lenguaje query, procesadores de interface, acceso a datos, y seguridad. El DBMS debe interfazarse con otro sistema, tal como el sistema operativo y compiladores de varios lenguajes de programación.

1.5.2 Desarrolladores de Herramientas

Las **herramientas** son paquetes de software que facilitan el diseño y uso del sistema de base de datos, y ayudan a mejorar el desempeño. Las herramientas son paquetes opcionales que se compran separadamente. Incluyen paquetes para diseño de bases de datos, monitoreo de desempeño, lenguaje natural o interfaces gráficas, prototipo, simulación, y generación de datos de prueba.

1.5.3 Operadores y Personal de Mantenimiento

Estos son el personal de administración del sistema que es responsable de que el medio ambiente de hardware y software se encuentre operando adecuadamente.

A pesar de que las categorías anteriores de trabajadores detrás de la escena son instrumental en hacer el sistema disponible a los usuarios finales, típicamente no usan la base de datos para sus propios propósitos.

1.6 Usos Proyectados de un DBMS

Aquí discutiremos los usos proyectados de un DBMS y las capacidades que un buen DBMS debe poseer. El DBA debe utilizar estas capacidades para completar una variedadd de objetivos relacionados al diseño, administración y uso de una gran base de datos multiusuario.

1.6.1 Control de la Redundancia

En el tradicional desarrollo de software que usa el procesamiento de archivos, cada grupo de usuarios mantiene sus propios archivos para manejar sus aplicaciones de procesamiento de información. Por ejemplo, consideremos el ejemplo de la base de datos UNIVERSIDAD; aquí, dos grupos de usuarios podrían ser el personal de registro y la oficina de contabilidad. En el acercamiento tradicional, cada grupo mantiene independientemente los archivos de estudiantes. Mucha de la información se almacena dos veces: una en los archivos de cada grupo de usuarios. Grupos adicionales de usuarios pueden duplicar algunos o todos los datos en sus propios archivos.

Esta **redundancia** de almacenar la misma información varias veces conduce a varios problemas. Primero, hay necesidad de realizar una actualización lógica -tal como ingresar la información de un nuevo estudiante- varias veces: una vez para cada archivo donde se almacene la información de estudiantes. Esto conduce a *duplicación de esfuerzo*. Segundo, *el espacio de almacenamiento se desperdicia* cuando la misma información se almacena repetidas veces, este problema puede ser muy serio en grandes bases de datos. Tercero, los archivos que representan la misma información se vuelven *inconsistentes*. Esto puede suceder debido a una actualización que se aplica a algunos de los archivos y no a todos. Aún si una actualización -tal como agregar un nuevo estudiante- se aplica a todos

los archivos apropiados, la información concerniente al estudiante puede seguir siendo **inconsistente** debido a que las actualizaciones se aplican independientemente por cada grupo de usuarios. Por ejemplo, un grupo de usuarios puede ingresar la fecha de nacimiento erróneamente como ENE-19-1974, mientras que los demás grupos de usuarios ingresan el valor correcto de ENE-29-1974.

En la aproximación de bases de datos, las vistas de diferentes grupos de usuarios se integran durante el diseño de la base de datos. Por consistencia, debemos tener un diseño que almacene cada elemento lógico -tal como el nombre o la fecha de nacimiento- en *un sólo lugar* de la base de datos. Esto no permite ninguna inconsistencia, y ahorra espacio de almacenamiento. En algunos casos, la **redundancia controlada** puede ser útil. Por ejemplo, podemos almacenar Nom Estud y NumCurso redundantemente en un archivo REP_CAL (Figura 1.5(a)), porque, cada que recuperamos un registro REP_CAL, queremos recuperar el nombre del estudiante y el número de curso junto con la calificación, el número de estudiante, y el identificador de sección. Colocando toda la información junta, no tenemos que buscar en varios archivos para colectarla. En tales casos, el DBMS debe tener la capacidad de **controlar** esta redundancia para prohibir inconsistencias entre los archivos. Esto puede lograrse automáticamente checando que los valores de NomEstud y NumEstud en cualquier registro de REP_CAL en la Figura 1.5(a) coincida con uno de los valores de nombe y número de estudiante de un registro ESTUDIANTE (Figura 1.2). Similarmente, los valores de IdSección y Numcurso en REP_CAL pueden checarse contra los registros SECCION. Tales chequeos pueden especificarse al DBMS cada vez que se actualice el archivo REP_CAL. La Figura 1.5(b) muestra un registro REP_CAL que es inconsistente con el archivo ESTUDIANTE, el cual puede ingresar erróneamente si no se controla la concurrencia.

(a)

REP_CAL	NumEstud	NomEstud	IdSección	NumCurso	Cal
	17	Smith	112	MAT2410	B
	17	Smith	119	CC1310	C
	8	Brown	85	MAT2410	A
	8	Brown	92	CC1310	A
	8	Brown	102	CC3320	B
	8	Brown	135	CC3380	A

(b)

REP_CAL	NumEstud	NomEstud	IdSección	NumCurso	Cal
	17	Brown	112	MAT2410	B

Figura 1.5 Almacenamiento redundante de elementos de datos entre archivos. (a) redundancia controlada: Incluyendo NomEstud y NumCurso en el archivo REP_CAL. (b) Redundancia sin control: Un registro REP_CAL que es consistente con los registros ESTUDIANTE de la Figura 1.2 (el nombre del estudiante número 17 es Smith, no Brown).

1.6.2 Restricción de Acceso No Autorizado

Cuando varios usuarios comparten una base de datos, es como si algunos usuarios no estuvieran autorizados para accesar toda la información en la base de datos. Por ejemplo, la información financiera es considerada confidencial, y por lo tanto sólo las personas

autorizadas tienen acceso a esta información. Además, a algunos usuarios sólo se les permite recuperar registros, mientras que a otros se les permite recuperar y actualizar. Por lo tanto, también debe controlarse la operación de tipo de acceso -recuperación o actualización. Típicamente, a los usuarios o grupos de usuarios se les dan números de cuenta protegidos por passwords, los cuales pueden usar para tener acceso a bases de datos. Un DBMS debe proporcionar un subsistema de **seguridad y autorización**, el cual el DBA usa para crear cuentas y especificar las restricciones. El DBMS debe entonces forzar estas restricciones automáticamente. Nótese que se pueden aplicar controles similares al software del DBMS. Por ejemplo, sólo al satff del DBA se le permite usar cierto software **privilegiado**, tal como el software para crear nuevas cuentas. Similarmente, a los usuarios paramétricos se les permite accesar a la base de datos sólo a través de transacciones enlatadas desarrolladas para su uso.

1.6.3 Almacenamiento Persistente para Objetos y Estructuras de Datos

Esta es una de las principales razones de la aparición de los DBMS **orientados a objetos**. Los lenguajes de programación típicamente tienen complejas estructuras de datos, tales como los record de PASCAL o las definiciones de clase en C++. Los valores de las variables se descartan una vez que el programa finaliza, a menos que el programador explícitamente las guarde en archivos permanentes, lo cual involucra convertir estas complejas estructuras a un formato aconsejable de almacenamiento en archivo. Cuando hay necesidad de leer estos datos una vez más, el programador debe convertir de formato archivo a estructura de variables. Los sistemas de bases de datos orientados a objetos son compatibles con lenguajes de programación tales como C++ y SMALLTALK, y el software del DBMS automáticamente realiza cualquier conversión necesaria. Por lo tanto, un objeto complejo en C++ puede almacenarse permanentemente en un DBMS orientado a objetos, tal como ObjectStore. Tal objeto se dice ser **persistente**, porque sobrevive a la terminación de la ejecución del programa y después puede recuperarse directamente por otro programa en C++.

El almacenamiento persistente de objetos y estructuras de datos es una función importante de los sistemas de bases de datos. Los sistemas de bases de datos tradicionales sufrían del así llamado *problema de impedimiento de correspondencia*, debido a que las estructuras de datos proporcionadas por el DBMS eran incompatibles con las estructuras de datos de los lenguajes de programación. Los sistemas de bases de datos orientados a objetos ofrecen *compatibilidad* de estructuras de datos con uno o más lenguajes de programación orientados a objetos.

1.6.4 Inferencia de Bases de Datos usando Reglas de Deducción

Otra aplicación reciente de los sistemas de bases de datos es proporcionar capacidades de definir *reglas de deducción* para *inferenciar* nueva información a partir de los términos almacenados en la base de datos. Tales sistemas son llamados sistemas de bases de datos **deductivos**. Por ejemplo, puede haber reglas complejas en el minimundo para detrminar cuando un estudiante está aprobado. Estas pueden especificarse *declarativamente* como

reglas de deducción, las cuales al ejecutarse pueden determinar todos los estudiantes aprobados. En un DBMS tradicional, un código explícito de *programa de procedimientos* tendría que escribirse para soportar tales aplicaciones. Pero si las reglas del minimundo cambian, es generalmenta más conveniente cambiar las reglas de deducción declaradas que recodificar programas de procedimientos.

1.6.5 Proporción de Múltiples Interfaces de Usuario

Debido a que muchos tipos de usuarios, con varios niveles de conocimientos técnicos, usan una base de datos, un DBMS debe proporcionar una variedad de interfaces con el usuario. Estas incluyen lenguajes query para usuarios casuales, interfaces de lenguajes de programación para programadores de aplicaciones, formas y códigos de comandos para usuarios paramétricos, e interfaces manejadas con menús y de lenguaje natural para usuarios independientes.

1.6.6 Representación de Relaciones Complejas entre Datos

Una base de datos puede incluir numerosas variedades de información que se inter-relacionen de muchas formas. Consideremos el ejemplo de la Figura 1.2. El registro de Brown en el archivo ESTUDIANTES se relaciona con cuatro registros en el archivo REP_CAL. De manera similar, cada registro de SECCION se relaciona a uno de CURSO así como a un número de registros de REP_CAL -uno para cada estudiante que haya completado esa sección. Un DBMS debe tener la capacidad de representar una variedad de relaciones complejas entre los datos así como de recuperar y actualizar la información relacionada de manera fácil y eficiente.

1.6.7 Reforzamiento de los Argumentos de Integridad

La mayoría de las aplicaciones de bases de datos tienen ciertos **argumentos de integridad** que deben retenerse para los datos. Un DBMS debe proporcionar capacidades de definir y reforzar estos argumentos. El tipo más simple de argumento de integridad involucra especificar un tipo de dato para cada elemento de información. Por ejemplo, en la Figura 1.2, podemos especificar que el valor del elemento clase dentro de cada registro ESTUDIANTE debe ser un entero entre 1 y 5 y que el valor de Nombre debe ser una cadena no mayor a 30 caracteres alfabéticos. Un tipo más complejo de argumento que ocurre frecuentemente involucra especificar que un registro en un archivo debe relacionarse con registros de otros archivos. Por ejemplo, en la Figura 1.2 podemos especificar que "cada registro SECCION debe relacionarse a uno de CURSO". Otro tipo de argumento especifica la unicidad de valores de los elementos de datos, tales como "cada registro CURSO debe tener un valor único de Numcurso". Estos argumentos se derivan del significado o **semántica** de los datos y del minimundo que representan. Es responsabilidad de los diseñadores de la base identificar los argumentos de integridad durante el diseño de la misma. Algunos argumentos pueden especificarse y reforzarse automáticamente al DBMS. Otros argumentos pueden tener que checarse por los programas de actualización o al momento de la entrada de datos.

Un elemento de información puede ingresarse erróneamente y seguir satisfaciendo los argumentos de integridad especificadas. Por ejemplo, si un estudiante recibe una calificación de A pero se ingresa una calificación de C a la base de datos, el DBMS *no puede* descubrir este error automáticamente, porque C es un valor válido para el tipo de dato Cal. Tales errores de entrada de datos pueden descubrirse manualmente (cuando el estudiante recibe la calificación y se queja) y corregirse posteriormente actualizando la base de datos. Sin embargo, una calificación de X puede rechazarse automáticamente por el DBMS, porque X no es un valor válido para el tipo de dato Cal.

1.6.8 Proporcionando Respaldo y Recuperación

Un DBMS debe proporcionar facilidades para recuperar fallas en el hardware o software. Por ejemplo, si el sistema falla a la mitad de un programa complejo de actualización, el subsistema de recuperación es responsable de asegurarse que la base de datos se restaure al estado en el que estaba antes de que el programa empezara a ejecutarse. Alternativamente, el subsistema de recuperación podría asegurarse de que el programa sea continuado desde el punto en el cual fue interrumpido de tal forma que su efecto completo se registre en la base de datos.

1.7 Implicaciones del Acercamiento de Bases de Datos

Además de los temas discutidos en la sección previa, otras implicaciones de utilizar bases de datos pueden beneficiar a la mayoría de las organizaciones.

1.7.1 Reforzamiento Potencial de Estándares

El acercamiento de bases de datos permite al DBA definir y reforzar los estándares entre usuarios de la base en una gran organización. Esto facilita la comunicación y la cooperación entre varios departamentos, proyectos, y usuarios dentro de la organización. Pueden definirse estandares para nombres y formatos de elementos de datos, formatos de despliegue, estructuras de reportes, terminología, y otros. El DBA puede reforzar estándares en un medio ambiente centralizado más fácilmente que en un medio donde cada grupo de usuarios tiene control de sus propios archivos y software.

1.7.2 Reducción de Tiempo en el Desarrollo de Aplicaciones

Una primer característica de venta del acercamiento de bases de datos es que el desarrollo de una nueva aplicación -tal como la recuperación de cierta información de la base para imprimir un nuevo reporte- toma muy poco tiempo. El diseño e implementación de una nueva base desde cero puede tomar más tiempo que escribir una aplicación de archivo especializada. Sin embargo, una vez que la base de datos está creada y corriendo, se requiere substancialmente menor tiempo para crear nuevas aplicaciones usando las facilidades del DBMS. El tiempo de desarrollo usando un DBMS es estimado de un sexto a un cuarto de lo que toma en un sistema tradicional de archivos.

1.7.3 Flexibilidad

Puede ser necesario cambiar la estructura de una base de datos como requerimiento. Por ejemplo, puede emerger un nuevo grupo de usuarios que necesite información adicional y no disponible actualmente en la base de datos. En respuesta, podemos necesitar agregar un nuevo archivo a la base o extender los elementos de un archivo existente. Algunos DBMS permiten tales cambios a la estructura sin afectar la información almacenada y a los programas de aplicaciones existentes.

1.7.4 Disponibilidad de Información Hasta la Fecha

Un DBMS hace disponible la base de datos a todos los usuarios. Tan pronto como un usuario actualiza a la base de datos, todos los usuarios pueden ver inmediatamente esta actualización. Esta disponibilidad de información hasta la fecha es esencial para muchas aplicaciones de procesamiento de transacciones, tales como sistemas de reservación o bases bancarias, y se hace posible por el control de concurrencia y subsistemas de recuperación de un DBMS.

1.7.5 Economía de Escala

El acercamiento DBMS permite la consolidación de datos y aplicaciones, reduciendo así la cantidad de traslapes desperdiciados entre actividades de personal de procesamiento en diferentes proyectos o departamentos. Esto permite que la organización completa invierta en procesadores más poderosos, dispositivos de almacenamiento, o equipo de comunicaciones, en vez de que cada departamento compre su propio equipo separadamente. Esto reduce los costos operacionales y administrativos de manera global.

1.8 Cuando No Usar un DBMS

A pesar de estas ventajas, existen algunas situaciones donde usar un DBMS puede incurrir en costos innecesarios comparados con el procesamiento de archivos tradicional. Los costos de usar un DBMS se deben a lo siguiente:

- Alta inversión inicial en hardware, software y entrenamiento.
- Un DBMS proporciona generalidad en la definición y procesamiento de datos.
- Proporciona seguridad, control de concurrencia, recuperación, y funciones de integridad.

Si los diseñadores y DBA de la base no la diseñan apropiadamente o si las aplicaciones no son implementadas apropiadamente pueden derivarse problemas adicionales. Debido a los costos de usar un DBMS y los problemas potenciales de administración inapropiada, puede ser más deseable usar archivos regulares bajo las siguientes circunstancias:

- La base de datos y las aplicaciones son sencillas, bien definidas, y no esperan cambios.
- Hay severas restricciones de requerimientos en tiempo real de algunos programas que no pueden cumplirse por el DBMS.
- No se requiere acceso multiusuario a la información.

CAPITULO 2

Conceptos y Arquitectura del Sistema de Base de Datos

2.1 Modelos de Datos, Esquemas, e Instancias

Una característica fundamental del acercamiento de bases de datos es que proporciona algún nivel de abstracción ocultando detalles de almacenamiento que no son necesarios para la mayoría de los usuarios. Un **modelo de datos** es un conjunto de conceptos que pueden usarse para describir la estructura de una base de datos. Por *estructura de una base de datos*, queremos decir los tipos de datos, relaciones, y argumentos que debe retener la información. La mayoría de los modelos también incluyen un conjunto de **operaciones básicas** para especificar recuperaciones y actualizaciones de la base de datos. Se está volviendo común la práctica de incluir conceptos en el modelo de datos para especificar **comportamientos**; esto se refiere a especificar en la base de datos un conjunto de **operaciones válidas definidas por el usuario** además de las operaciones básicas proporcionadas por el modelo de datos. Un ejemplo de operación definida por el usuario es CALCULA_CAL, la cual puede aplicarse a un objeto ESTUDIANTE. Por otro lado, las operaciones genéricas de insertar, borrar, modificar, o recuperar un objeto se incluyen dentro de las operaciones básicas del modelo de datos.

2.1.1 Categorías de los Modelos de Datos

Muchos modelos de datos han sido propuestos. Podemos categorizar modelos de datos basados en los tipos de conceptos que ellos proporcionan para describir la estructura de la base de datos. Los modelos de datos de **alto nivel o conceptual** proporcionan conceptos que están cerca de la forma en que muchos usuarios perciben los datos, mientras que los de **bajo nivel o físicos** proporcionan conceptos que describen los detalles del almacenamiento de los datos en la computadora. Los conceptos proporcionados por los modelos de bajo nivel son generalmente empleados por especialistas en computadoras, no por usuarios. Entre estos dos extremos está una clase de modelos **representacionales** (o **de implementación**), los cuales proporcionan conceptos que pueden entenderse por los usuarios finales pero que no están muy alejados de la forma en que la información se organiza dentro de la computadora. Los modelos representacionales ocultan algunos detalles de almacenamiento pero pueden implementarse de manera directa en un sistema de cómputo.

Los modelos de alto nivel usan conceptos tales como entidades, atributos y relaciones. Una **entidad** representa un objeto o concepto del mundo real, tal como un empleado o proyecto, que se almacena en la base de datos. Un **atributo** representa alguna propiedad de interés que describe más a una entidad, tal como el nombre o salario de un empleado. Una **relación** entre dos o más entidades representa una interacción entre entidades; por ejemplo, una relación de trabajo entre un empleado y un proyecto.

Los modelos de datos representacional o de implementación son los más utilizados frecuentemente en DBMSs comerciales, e incluyen los tres modelos de datos más ampliamente usados: relacional, de red y jerárquico. Representan datos usando estructuras de registros y por lo tanto son a veces llamados modelos **basados en registros.** Podemos

contemplar a los modelos **orientados a objetos** como una nueva familia de implementación de nivel superior que está más cercano a los modelos conceptuales.

Los modelos físicos describen cómo se almacenan los datos representando información tal como formatos de registro, ordenamientos, y trayectorias de acceso. Una **trayectoria de acceso** es una estructura que hace eficiente la búsqueda de registros en una base de datos.

2.1.1 Esquemas e Instancias

En cualquier modelo es importante distinguir entre la descripción de la base y la *base en si*. A la descripción de la base se le llama **esquema** (o **meta-data**). Durante el diseño se especifica un esquema y se espera que no cambie con frecuencia. La mayoría de los modelos tienen ciertas convensiones para desplegar diagramalmente los esquemas. A un esquema desplegado se le llama **diagrama esquema**. La Figura 2.1 muestra un diagrama esquema para la base de la Figura 1.2; el diagrama despliega la estructura de cada tipo de registro pero no las instancias actuales de los registros. A cada objeto del esquema -tal como ESTUDIANTE o CURSO- se le llama **constructor de esquema**.

Un diagrama esquema despliega sólo *algunos aspectos* del mismo, tales como los nombres los tipos de registros y elementos de datos, y algunos tipos de argumentos. Otros aspectos no se especifican en el diagrama esquema; por ejemplo, la Figura 2.1 no muestra ni el tipo de dato de cada elemento ni las relaciones entre los archivos. Muchos tipos de argumentos no se representan en diagramas esquema; por ejemplo, es difícil representar un argumento tal como "los estudiantes de la carrera ciencias de la computación deben tomar CC1310 antes de finalizar su segundo año".

La información dentro de una base de datos puede cambiar con frecuencia; por ejemplo, la base de datos de la Figura 1.2 cambia cada que se agrega un nuevo estudiante o una nueva calificación. A los datos de la base en un momento en particular se le llama **estado de la base** (o conjunto de **ocurrencias** o **instancias**). En un estado dado, cada esquema tiene su propio *conjunto actual* de instancias; por ejemplo, ESTUDIANTES contendrá como sus instancias al conjunto individual de estudiantes (registros). Cada vez que se inserta o se borra un registro, o cambia de valor algún elemento, cambiamos de un estado a otro.

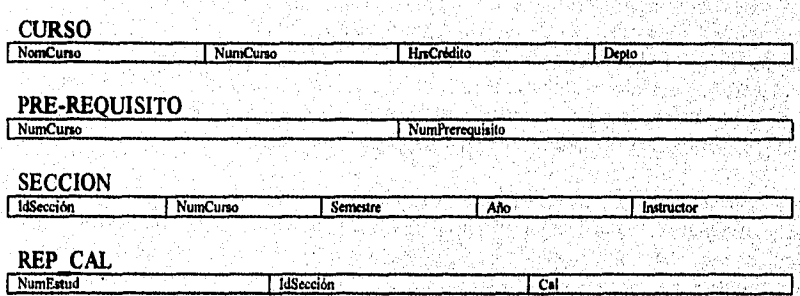


Figura 2.1 Esquema para la base de datos de la Figura 1.2

La distinción entre esquema y estado es muy importante. Cuando se **define** una nueva base de datos, sólo especificamos su esquema al DBMS. En este momento, el estado correspondiente de la base es el "estado vacío" sin datos. Cuando se **carga** información por primera vez obtenemos el "estado inicial" de la base. De ahí en adelante, cada que se aplica una operación de actualización a la base, obtenemos otro estado. El DBMS es en parte responsable de asegurar que *cada* estado de la base sea un **estado válido** -esto es, uno que satisfaga la estructura y argumentos especificados en el esquema. El DBMS almacena el esquema en el catálogo de tal forma que el software pueda referirse a él cada vez que lo necesite. Al esquema algunas veces se le llama la **intensión**, y al estado de la base se le llama una **extensión** del esquema.

2.2 Arquitectura del DBMS e Independencia de Datos

Tres características importantes del acercamiento de bases de datos son: (a) el aislamiento de programas y datos (independencia programa-datos y programa operación); (b) soporte de múltiples vistas; y (c) uso de un catálogo para almacenar la descripción de la base de datos (esquema). En esta sección especificamos la **arquitectura de los tres esquemas**, que fue propuesta para ayudar a cumplir estas tres características.

2.2.1 Arquitectura de los Tres Esquemas

La meta de la arquitectura de los tres esquemas, ilustrado en la Figura 2.2, es separar las aplicaciones y la base de datos física de los usuarios. En esta arquitectura, los esquemas pueden definirse en los tres niveles siguientes:

1. El **nivel interno** tiene un **esquema interno**, el cual describe los detalles completos de almacenamiento y trayectorias de acceso a la base de datos.

2. El **nivel conceptual** tiene un **esquema conceptual**, el cual oculta los detalles de almacenamiento físico y se concentra en describir entidades, tipos de datos, relaciones, operaciones del usuario, y argumentos. A este nivel puede usarse un modelo de alto nivel o de implementación.

3. El **nivel externo** o **de vista** incluye un **número de esquemas externos** o **vistas de usuarios**. Cada esquema externo describe la parte de la base de datos en que se interesa un grupo de usuarios en particular y oculta el resto de la base de datos de ese grupo. Aquí puede usarse un modelo de alto nivel o de implementación.

La mayoría de los DBMSs no separan completamente los tres niveles, pero varios de ellos soportan la arquitectura de los tres esquemas. Algunos DBMSs incluyen detalles a nivel físico en el esquema conceptual. En la mayoría de los DBMSs que soportan vistas de usuarios, los esquemas externos se especifican en el mismo modelo que describe la información a nivel conceptual. Algunos DBMSs permiten que se usen diferentes modelos a niveles conceptuales y externos.

Notemos que los tres esquemas son sólo descripciones de datos; el único dato que *actualmente* existe es a nivel físico. En un DBMS basado en la arquitectura de los tres esquemas, cada grupo de usuarios se refiere únicamente a su propio esquema externo. Por lo tanto, el DBMS debe transformar una requisición especificada en un esquema externo en una requisición contra el esquema conceptual, y posteriormente en una requisición en el esquema interno para procesarse sobre la base almacenada. Si la requisición es una recuperación de la base de datos, la información extraida de la base almacenada debe reformatearse para cumplir con la vista externa del usuario. Al proceso de transformar requisiciones y resultados entre niveles se le llama **mapeo**. Estos mapeos pueden ser consumidores de tiempo, así algunos DBMSs -especialmente aquéllos diseñados para soportar pequeñas bases de datos- no soportan vistas externas. Aún en tales sistemas, sin embargo, es necesaria un poco de mapeo para transformar requisiciones entre los niveles conceptuales e internos.

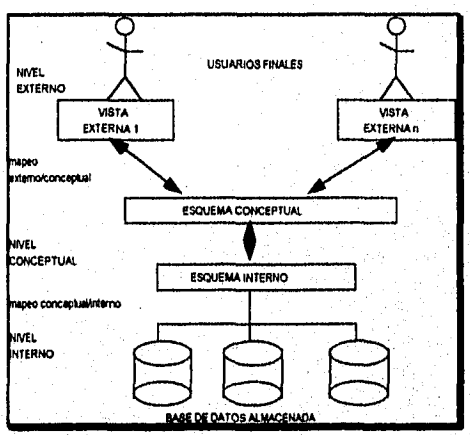


Figura 2.2 Arquitectura de los tres esquemas.

2.2.2 Independencia de Datos

La arquitectura de los tres esquemas puede utilizarse para explicar el concepto de **independencia de datos**, el cual puede definirse como la capacidad para cambiar el esquema a nivel de sistema de base de datos sin tener que cambiar el esquema al siguiente nivel superior. Podemos definir dos tipos de independencia de datos:

1. La **independencia lógica de datos** es la capacidad de cambiar el esquema conceptual sin tener que cambiar los esquemas externos o programas de aplicación. Podemos cambiar el esquema conceptual para expander la base de datos (agregando un nuevo tipo de registro o elemento de datos). En el más reciente caso, no deben afectarse los esquemas externos que se refieren sólo a la información restante. En un DBMS que soporta independencia lógica de datos sólo necesitan cambiarse las definiciones de vistas y los mapeos . Los programas de aplicación referencian a los esquemas de

construcciones externas tanto antes como después de que el esquema conceptual haya sido sometido a una reorganización lógica. También pueden aplicarse los cambios de argumentos al esquema conceptual sin afectar los esquemas externos.

2. La **independencia física de datos** es la capacidad de cambiar el esquema interno sin tener que cambiar los esquemas conceptuales (o externos). Pueden necesitarse cambios al esquema interno debido a que algunos archivos físicos tuvieran que reorganizarse -mejorar el desempeño de recuperación o actualización. Si la misma información que la anterior permanece en la base de datos, no se tiene que cambiar el esquema conceptual. Por ejemplo, proporcionar una trayectoria de acceso para mejorar la recuperación de registros SECCION (Figura 1.2) por semestre y año no debe requerir un query tal como "listar todas las secciones ofrecidas en Otoño del 91" para cambiarse, a pesar de que el query puede ejecutarse más eficientemente por el DBMS usando una nueva trayectoria de acceso. Debido a que la *independencia física de datos* se refiere únicamente al aislamiento de una aplicación respecto a las estructuras físicas de almacenamiento, es más fácil de adquirir que la independencia lógica de datos.

Cada vez que se tiene un DBMS multinivel, su catálogo debe expandirse para incluir información sobre cómo mapear requisiciones y datos entre los distintos niveles. El DBMS utiliza software adicional para acompletar estos mapeos referiendo a la información de mapeo en el catálogo. La independencia de datos se completa porque, cuando se cambia el esquema en algún nivel, el esquema del siguiente nivel permanece sin cambio; sólo cambia el mapeo entre los dos niveles. Los programas de aplicación que se refieren al esquema de más alto nivel no necesitan cambiarse. Por lo tanto, la arquitectura de los tres esquemas puede facilitar el logro de la verdadera independencia de datos, física y lógica. Sin embargo, los dos niveles de mapeo crean un problema durante la compilación o ejecución de un query o programa, llevando a ineficiencias en el DBMS. Debido a esto, pocos DBMS's han implementado completamente la arquitectura de los tres esquemas.

2.3 Lenguajes e Interfaces de Bases de Datos

El DBMS debe proporcionar lenguajes e interfaces apropiados para cada categoría de usuarios. En esta sección discutiremos los tipos de lenguajes e interfaces proporcionadas por un DBMS y las categorías de usuarios destinados a cada interface.

2.3.1 Lenguajes DBMS

Una vez que se completa el diseño de una base de datos y se elige un DBMS para implementarla, el primer paso es especificar los esquemas conceptual e interno de la base de datos y cualquier mapeo entre ambos. En muchos DBMSs donde no se mantiene separación estricta de niveles, se usa el **lenguaje de definición de datos (DDL),** para definir ambos esquemas. El DBMS tendrá un compilador DDL cuya función es procesar instrucciones DDL para identificar y almacenar la descripción del esquema en el catálogo del DBMS.

En DBMS's donde se mantiene una clara separación entre los niveles conceptuales e internos, el DDL se usa para especificar el esquema conceptual únicamente. Para especificar el esquema interno se usa el **lenguaje de definición de almacenamiento (SDL)**. Los mapeos entre los dos esquemas pueden especificarse en cualquiera de estos lenguajes. Para una verdadera arquitectura de tres esquemas, necesitaríamos un tercer lenguaje, el **lenguaje de definición de vistas (VDL)**, para especificar vistas a los usuarios y sus mapeos al esquema conceptual. Una vez que se compilan los esquemas y se popula a la base, los usuarios deben tener algunos medios para manipular la base de datos. Las manipulaciones típicas incluyen recuperación, inserción, borrado, y modificación de los datos. Para estos propósitos el DBMS proporciona un **lenguaje de manipulación de datos (DML)**.

Es común en los DBMS's actuales no identificar los tipos de lenguajes precedentes como distintos; en vez de ello, puede usarse un lenguaje global integrado que incluye construcciones para la definición del esquema conceptual, definición de vistas, manipulación de datos, y definición de almacenamiento. Un ejemplo típico es el lenguaje de bases de datos relacional SQL, el cual representa una combinación de DDL, VDL, MDL y SDL, a pesar de que el componente SDL está siendo eliminado del lenguaje.

Existen dos tipos principales de DML's. Un DML de **alto nivel** o **no procedural** puede usarse para especificar de manera concisa operaciones complejas de bases de datos. Muchos DBMS's permiten instrucciones DML de alto nivel ya sea para ser ingresadas interactivamente desde una terminal o para ser contenidas en un lenguaje de propósito general de alto nivel. En el último caso, deben identificarse instrucciones DML dentro del programa de tal forma que puedan procesarse por el DBMS. Un DML de **bajo nivel** o **procedural** debe estar contenido en un lenguaje de programación de propósito general. Este tipo de DML típicamente recupera registros individuales de la base y los procesa separadamente. Por lo tanto, necesita hacer uso de las construcciones del lenguaje de programación, tales como los ciclos, para recuperar y procesar cada registro individual de un conjunto de registros. Debido a esta propiedad a los DML de bajo nivel también se les llaman DML's de **registro a la vez**. Los DML's de alto nivel, tales como SQL, pueden especificar y recuperar muchos registros en una sola instrucción DML y son llamados DML's de **conjunto a la vez** u **orientados a objetos**. Un query en un DML de alto nivel especifica a menudo *qué datos* van a recuperarse en vez de *cómo* recuperar la información; por lo tanto, tales lenguajes son llamados **declarativos**.

Siempre que los comandos DML, ya sea de alto o bajo nivel, están contenidos en un lenguaje de programación de propósito general, al lenguaje se le llama **lenguaje huésped** y al DML se le llama **sublenguaje de datos**. En los DBMS's más recientes, tales como los sistemas orientados a objetos, el huésped y los sublenguajes de datos típicamente forman un lenguaje integrado tal como C++. Por otro lado, a un DML de alto nivel usado de manera independiente se le llama **lenguaje query**. En general, los comandos de recuperación y actualización de un DML de alto nivel pueden usarse interactivamente y se consideran por lo tanto parte del lenguaje query.

Los usuarios finales casuales típicamente usan un lenguaje query de alto nivel para especificar sus requisiciones, toda vez que los programadores usan al DML en su forma básica. Para usuarios parámetricos, existen usualmente **interfaces amigables** para interactuar con la base de datos; estas también pueden usarse por usuarios casuales u otros quienes no quieran aprender los detalles de un lenguaje query de alto nivel.

2.3.2 Interfaces DBMS

Dentro de las interfaces amigables al usuario proporcionadas por un DBMS se pueden incluir las siguientes:

Interfaces basadas en menús. Estas interfaces presentan al usuario listas de opciones, llamadas **menús**, que llevan al usuario a través de la formulación de una requisición. Los menús deben estar lejos de la necesidad de memorizar los comandos específicos y la sintáxis de un lenguaje query; en cambio, el query se compone paso a paso escogiendo opciones de un menú que se despliega por el sistema. Los menús escalonados se están volviendo muy populares en interfaces basadas en ventanas. Se usan con frecuencia en **interfaces de barrido**, las cuales permiten que un usuario busque a través del contenido de la base de datos de manera no estructurada.

Interfaces Gráficas. Una interface gráfica típicamente despliega un esquema al usuario en forma de diagrama. El usuario entonces puede especificar un query manipulando el diagrama. En muchos casos, las interfaces gráficas se combinan con menús. La mayoría de las interfaces gráficas usan un **dispositivo apuntador**, tal como un mouse, para seleccionar ciertas partes del esquema seleccionado.

Interfaces Basadas en Formas. Despliegan una **forma** a cada usuario. Los usuarios pueden llenarlas para insertar nuevos datos, o llenar sólo ciertas entradas, en cuyo caso el DBMS recuperará la información correspondiente de las entradas restantes. Las formas usualmente se diseñan y programan para usuarios inexpertos como interfaces a transacciones enlatadas. Muchos DBMS's tienen lenguajes especiales, llamados *lenguajes de especificación de formas*, que ayudan a los programadores a su especificación. Algunos sistemas tienen utilerías que definen una forma dejando al usuario final construir interactivamente una muestra en la pantalla.

Interfaces de Lenguaje Natural. Estas interfaces aceptan requisiciones escritas en inglés o algún otro lenguaje e intentan "entenderlas". Una interface de lenguaje natural usualmente tiene su propio "esquema", el cual es similar a la base de datos conceptual. La interface refiere a las palabras en su esquema, así como a un conjunto de palabras estándares, en la interpretación de la requisición. Si la interpretación es exitosa, la interface genera un query de alto nivel correspondiente a la requisición en lenguaje natural y lo somete al DBMS para procesamiento; de otro modo, se inicia un diálogo con el que el usuario aclara la requisición.

Interfaces para usuarios paramétricos. Los usuarios paramétricos tienen con frecuencia un pequeño conjunto de operaciones que deben realizar repetidamente. Los analistas y programadores diseñan e implementan una interface especial para una clase conocida de usuarios ingenuos. Usualmente, se incluye un pequeño conjunto de comandos abreviados, con la finalidad de minimizar las teclas de cada requisición. Por ejemplo, pueden programarse las teclas de funciones en una terminal para iniciar varios comandos. Esto permite al usuario paramétrico a proceder con un número mínimo de teclas.

Interfaces para el DBA. La mayoría de los sistemas de bases de datos contienen comandos privilegiados que pueden usarse por el staff del DBA. Estos incluyen comandos para la creación de cuentas, colocación de parámetros del sistema, conceder autorización a cuentas, cambios al sistema, y reorganización de la estructura de almacenamiento en la base de datos.

2.4 Medio Ambiente del Sistema de Bases de Datos

Un DBMS es un sistema complejo de software. En esta sección discutimos los tipos de software que constituyen a un DBMS y los tipos de sistemas de cómputo con los cuales interactúa el DBMS.

2.4.1 Módulos Componentes del DBMS

La Figura 2.3 ilustra los componéntes típicos de un DBMS. La base de datos y el catálogo del DBMS se almacenan usualmente en disco. El acceso a disco se controla principalmete por el *sistema operativo* (SO), el cual calendariza la I/O del disco. Un módulo **administrador de información almacenada** de más alto nivel controla el acceso del DBMS a la información almacenada, ya sea parte de la base de datos o del catálogo. Las líneas punteadas A, B, C, D y E ilustran los accesos que están bajo el control de este administrador. El administrador puede usar los servicios básicos del SO para llevar a cabo la transferencia de información a bajo nivel entre el disco y el medio de almacenamiento principal, pero controla otros aspectos de transferencia de datos, tales como el manejo de buffers en memoria principal. Una vez que los datos están en los buffers de la memoria principal, pueden ser procesados por otros módulos del DBMS.

El **compilador DDL** procesa definiciones de esquemas, especificados en el DDL, y almacena la descripción de los esquemas (meta-data) en el catálogo del DBMS. El catálogo incluye información tal como nombres de los archivos, elementos de información, detalles de almacenamiento de cada archivo, información de mapeo entre esquemas, y argumentos. Los módulos de software del DBMS que necesitan buscar esta información deben accesar al catálogo.

El **runtime de la base de datos** maneja el acceso a la misma, al momento de la corrida; recibe operaciones de recuperación o actualización y las lleva a cabo en la base de datos. El acceso a disco va a través del administrador de información almacenada. El **compilador de queries** maneja queries de alto nivel que están siendo ingresados

interactivamente, los parsea y analiza, y posteriormente genera llamadas al procesador para ejecutar la requisición.

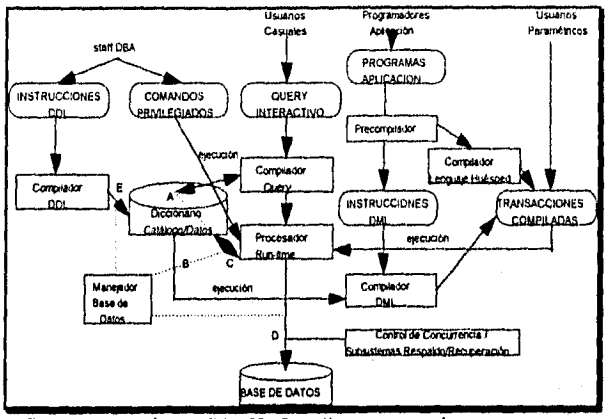


Figura 2.3 Componentes de un DBMS. Las líneas punteadas muestran acceso bajo el control del manejador de datos almacenados.

El **precompilador** extrae comandos DML de una aplicación escrita en un lenguaje de programación huésped. Estos comandos se envían al **compilador DML** para compilarse a código objeto para acceso a la base de datos. El resto del programa se envía al compilador del lenguaje huésped. Los códigos objeto de los comandos DDL y el resto del programa se enlazan, formando una transacción enlatada cuyo código ejecutable incluye llamadas al procesador del run-time.

La Figura 2.3 no intenta describir a un DBMS en específico; sino que, lo usamos para ilustrar módulos típicos de un DBMS. El DBMS interactúa con el sistema operativo cuando es necesario el acceso a disco -a la base de datos o al catálogo-. Si el sistema de cómputo es compartido por muchos usuarios, el SO calendarizará las requisiciones de acceso y el procesamiento junto con otros procesos. El DBMS también se interfaza con compiladores para lenguajes de programación huésped de alto nivel. Las interfaces amigables con el usuario pueden proporcionarse para ayudar a cualquiera de los tipos de usuarios mostrados especificando sus requisiciones.

2.4.2 Utilerías de Sistemas de Bases de Datos

Además de poseer los módulos de software recién descritos, la mayoría de los DBMS's tienen **utilerías** que ayudan al DBA a correr el sistema de bases de datos. Las utilerías comúnes tienen los siguientes tipos de funciones:

Carga: Se usa para cargar a la base de datos archivos de datos existentes -tales como archivos de texto o archivos secuenciales. Usualmente, se especifica a la utilería el formato

actual (fuente) del archivo de datos y la estructura del archivo deseado (objetivo), la cual entonces automáticamente reformatea los datos y los almacena en la base. Con la proliferación de los DBMSs, la transferencia de datos de un DBMS a otro se está volviendo común en varias organizaciones. Algunos vendedores ofrecen productos que generan los programas apropiados de carga, dado el archivo fuente existente y la descripción de almacenamiento de la base objetivo (esquema interno). A tales herramientas también se les llama **herramientas de conversión**.

Respaldo: Crea una copia de respaldo de la base de datos, usualmente descargando la base completa a cinta. La copia de respaldo puede usarse entonces para restaurar a la base de datos en caso de alguna falla catastrófica.

Organización de archivos: esta utilería puede usarse para reorganizar una base de datos en una organización de archivos diferente para mejorar su desempeño.

Monitoreo de desempeño: Tales utilerías monitorean el uso de la base de datos y proporcionan estadísticas al DBA. El DBA usa las estadísticas para tomar decisiones tales como si o no reorganizar los archivos para mejorar el desempeño.

Pueden haber otras utilerías disponibles para sortear archivos, manejar compresión de datos, monitorear acceso a usuarios, y desempeñar otras funciones. Otra utilería que puede ser útil en grandes organizaciones es un **sistema de diccionario de datos** expandido. Además de almacenar la información del catálogo acerca de esquemas y argumentos, el diccionario de datos almacena otra información tal como decisiones de diseño, uso de estándares, descripción de programas de aplicación, e información del usuario. Esta información puede accesarse *directamente* por los usuarios o el DBA cuando sea necesario. Un diccionario de datos es similar al catálogo del DBMS, pero incluye una variedad más amplia de información y es accesada principalmente por usuarios en lugar del software del DBMS. A un diccionario combinado catálogo/datos, el cual puede accesarse por usuarios y software del DBMS, se le llama **directorio de datos** o diccionario de datos **activo**. A un diccionario de datos que puede accesarse por usuarios y el DBA pero no por el software del DBMS se le llama **pasivo**.

2.4.3 Medios de Comunicaciones

El DBMS también necesita interfazarse con **software de comunicaciones**, cuya función es permitir a los usuarios en localidades remotas del sistema de base de datos acceso a la misma a través de terminales de cómputo, estaciones de trabajo, o sus mini o microcomputadoras locales. Estas se conectan al sitio de la base de datos a través de hardware de comunicaciones tales como líneas telefónicas, grandes redes de comunicaciones, o dispositivos de comunicación satelital. Muchos sistemas de bases de datos comerciales tienen paquetes de comunicaciones que trabajan con el DBMS. Al sistema DBMS y de comunicaciones integrado se le llama sistema **DB/DC**.

Algunos DBMS´s distribuidos están físicamente esparcidos sobre múltiples máquinas. En este caso, son necesarias redes de comunicaciones para conectar las máquinas. Estas son a menudo *redes de área local* (LAN), pero también pueden ser otro tipo de redes. El término **arquitectura cliente-servidor**, se usa junto con un DBMS si la aplicación corre físicamente en una máquina, llamada el **cliente**, y el almacenamiento y acceso es manejado por otra máquina, llamada el **servidor**. Pueden ofrecerse varias combinaciones de clientes y servidores -por ejemplo, un servidor para varios clientes.

2.5 Clasificación de Sistemas Manejadores de Bases de Datos

El principal criterio usado para clasificar un DBMS es el **modelo de datos** en el cual se basan. Los modelos de datos usados con más frecuencia en los DBMS´s actuales son lo relacionales, de red y jerárquicos. Algunos DBMS´s recientes están basados en orientación a objetos o modelos conceptuales. Categorizaremos a los DBMS's como **relacionales, de red, jerárquicos, orientados a objetos, y otros**.

Otro criterio usado para clasificar DBMS´s es el **número de usuarios** soportados por el sistema. Los **sistemas monousuarios** soportan a un solo usuario a la vez y se usan casi siempre en PC´s. Los **sistemas multiusuario**, los cuales incluyen a la mayoría de los DBMS´s, soportan a muchos usuarios de manera concurrente.

Un tercer criterio es el **número de sitios** sobre los cuales se distribuye la base de datos. La mayoría de los DBMS´s son **centralizados**, significa que sus datos se almacenan en un sólo sitio. Un DBMS centralizado puede soportar a múltiples usuarios, pero los DBMSs y las mismas bases residen totalmente en un sólo lugar. Un **DBMS distribuido (DDBMS)** puede tener a la base y al software del DBMS distribuidos sobre varios lugares, conectados por una red de computadoras. Los **DDBMSs homogéneos** usan el mismo software del DBMS en distintos lugares. Una tendencia reciente es desarrollar software para accesar a varias bases autónomas pre-existentes almacenadas bajo DBMSs **heterogéneos**. Esto conduce a un **DBMS federado** (o sistema **multibase**), donde los DBMSs participantes están débilmente acoplados y tienen un grado de autonomía local. Muchos DBMSs usan arquitectura cliente-servidor.

Un cuarto criterio es el **costo** del DBMS. La mayoría de los paquetes DBMSs cuestan entre $10,000 y $100,000 US Dollars. Los sistemas monousuario que trabajan con microcomputadoras cuestan entre $100 y $3000 US Dollars. Por otro lado, sólo unos cuantos elaboran paquetes que cuestan más de $100,000 US Dollars.

También se puede clasificar a un DBMS en base a los **tipos de trayectorias de acceso** disponibles para almacenar archivos. Finalmente, un DBMS puede ser de **propósito general** o **propósito especial**. Cuando el desempeño es una consideración primaria, puede diseñarse y construirse un DBMS de propósito especial para una aplicación en específico; dicho sistema no puede emplearse por otras aplicaciones. Muchos sistemas de reservaciones aéreas y directorios telefónicos son DBMS's de propósito especial. Estos caen dentro de la categoría de los **sistemas de procesamiento de transacción en línea**

(OLTP), los cuales deben soportar un gran número de transacciones concurrentes sin imponer retrasos excesivos.

Discutamos brevemente el criterio principal de clasificación de los DBMS's: el modelo de datos. El modelo **relacional** representa a una base de datos como una colección de tablas, donde cada tabla puede almacenarse como un archivo separado. La base de la Figura 1.2 se muestra de manera muy similar a la representación relacional. La mayoría de las bases relacionales tienen lenguajes query de alto nivel y soportan una forma limitada de vistas de usuarios.

El modelo de **red** representa los datos como tipos de registro y también representa un tipo limitado de 1:N relaciones, llamado conjunto de tipo. La Figura 2.4 muestra un diagrama de esquema de red para la base de datos de la Figura 1.2, donde los tipos de registro se muestran como rectángulos y los conjuntos de tipo se muestran como flechas dirigidas etiquetadas. El modelo de red, también conocido como modelo *CODASYL DBTG*, tiene un lenguaje asociado de registro a la vez que debe estar contenido en un lenguaje de programación huésped.

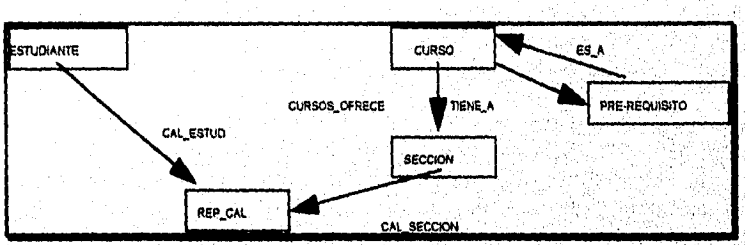


Figura 2.4 Un esquema de red.

El modelo **jerárquico** representa a la información como árboles de estructuras jerárquicas. Cada jerarquía representa un número de registros relacionados. No hay lenguaje estándar para el modelo jerárquico, a pesar de que la mayoría de ellos tienen lenguajes de registro a la vez.

El modelo **orientado a objetos** define a una base de datos en términos de objetos, sus propiedades, y sus operaciones. Los objetos con la misma estructura y comportamiento pertenecen a la **clase**, y las clases se organizan en jerarquías o gráficas acíclicas. Las operaciones de cada clase se especifican en términos de procedimientos predefinidos llamados **métodos**. Se han vuelto disponibles un número de sistemas comerciales basados en el paradigma de la orientación a objetos. Además, los DBMS's relacionales han estado extendiendo sus modelos para incorporar los conceptos de orientación a objetos y sus capacidades; a estos se les refiere como **sistemas relacionales extendidos**.

CAPITULO 3

Modelado de Datos
Usando el Modelo Entidad-Relación

3.1 Uso de Modelos Conceptuales de Alto Nivel para el Diseño de Bases de Datos

La Figura 3.1 muestra una descripción simplificada del proceso de diseño de bases de datos. El primer paso mostrado es la **colección y análisis de requerimientos.** Durante éste, los diseñadores entrevistan a usuarios prospecto para entender y documentar sus requerimientos de información. El resultado es un escrito conciso de los requerimientos de los usuarios. Estos requerimientos deben especificarse tan detallados y completos como sea posible. En paralelo a la especificación de requerimientos, es util especificar los *requerimientos funcionales* de la aplicación. Estos constan de operaciones definidas por el usuario (o **transacciones**) que se aplicarán a la base de datos, y que incluyan recuperaciones y actualizaciones. Para especificar requerimientos funcionales es común usar técnicas tales como *diagramas de flujo de datos.*

Una vez que se han recolectado los requerimientos, el siguiente paso es crear un **esquema conceptual** de la base de datos, usando un modelo conceptual de alto nivel. A este paso se le llama **diseño conceptual de la base de datos**. El esquema conceptual es una descripción concisa de los requerimientos de los usuarios e incluye descripciones detalladas de los tipos de datos, relaciones, y argumentos; estas se expresan usando los conceptos proporcionados por el modelo de alto nivel. El esquema conceptual de alto nivel puede usarse también como referencia para asegurar que los requerimientos de información de todos los usuarios se cumplan y que los requerimientos no incluyan ningun conflicto. Este acercamiento permite a los diseñadores concentrarse en la especificación de propiedades de los datos, sin preocuparse en los detalles de almacenamiento.

Después de que ha sido creado el diseño conceptual, pueden usarse las operaciones básicas del modelo para especificar transacciones de alto nivel correspondientes a las operaciones definidas por el usuario identificadas durante el análisis funcional. Esto también sirve para confirmar que el esquema conceptual cumple con todos los requerimientos funcionales identificados. Si algunos requerimientos no pudieron especificarse en el esquema inicial pueden introducirse modificaciones al modelo conceptual.

El siguiente paso es la implementación de la base, utilizando un DBMS comercial. La mayoría de los DBMS's comerciales usan un modelo de implementación, así el esquema conceptual se transforma de modelo de alto nivel en modelo de implementación. A esto se le llama el **diseño lógico de la base de datos** o **mapeo del modelo de datos**, y su resultado es un esquema de implementación en el modelo del DBMS.

Finalmente, el último paso es la fase del **diseño físico**, durante el cual se especifican las estructuras de almacenamiento interno y organización de archivos de la base. En paralelo a estas actividades, se diseñan e implementan los programas de aplicación como transacciones correspondientes a las especificaciones de transacción de alto nivel.

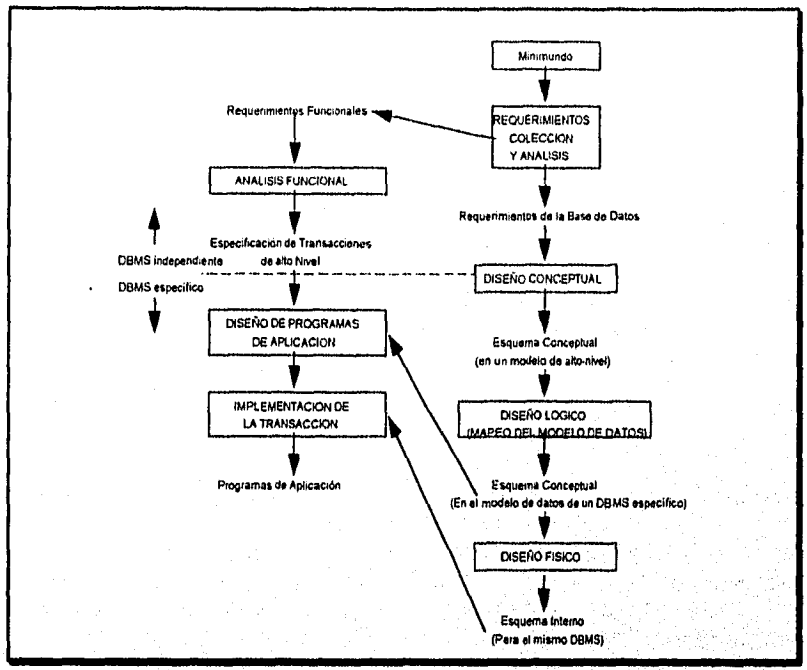


Figura 3.1 Fases del diseño de bases de datos.

3.2 Un Ejemplo

En esta sección describiremos un ejemplo de base de datos, llamada COMPAÑIA, que sirve pará ilustrar los conceptos del modelo ER y su uso en el diseño de esquemas. Enlistaremos los requerimientos de datos para esta base, conforme introducimos los conceptos del modelo ER crearemos paso a paso su esquema conceptual. La base de datos COMPAÑIA registra a los empleados, departamentos, y proyectos de la misma. Supongamos que, después de la fase de requerimientos y análisis, los diseñadores establecieron la siguiente descripción del "minimundo" -la parte de la compañía a representar en la base de datos:

1. La compañía está organizada en departamentos. Cada departamento tiene un nombre, y número único, y un empleado en particular que maneja al departamento. Registraremos la fecha inicial en que el empleado inició en el manejo del departamento. Un departamento puede tener diferentes localidades.

2. Un departamento controla cierto número de proyectos, cada uno de los cuales tiene un nombre y número únicos, y una sola localidad.

3. Almacenaremos cada nombre, número de seguro social, dirección, salario, sexo y fecha de nacimiento. Un empleado se asigna a un departamento pero puede trabajar en diferentes proyectos, los cuales no necesariamente son controlados por el mismo departamento. Guardaremos las horas trabajadas a la semana en cada proyecto por empleado. También registraremos al supervisor de cada empleado.

4. Queremos registrar a los dependientes de cada empleado. Guardamos el nombre, sexo, fecha de nacimiento y relación con el empleado de cada dependiente.

La Figura 3.2 muestra cómo puede desplegarse el esquema para esta aplicación de base de datos a través de la notación gráfica conocida como **diagramas ER**. Conforme introduzcamos los conceptos del modelo ER describiremos el proceso de derivar este esquema a partir los requerimientos iniciales -y explicar la notación diagramática ER.

3.3 Conceptos del Modelo ER

El modelo ER describe los datos como entidades, relaciones y atributos. En la siguiente sección introduciremos los conceptos de entidades y sus atributos. Discutiremos los tipos de entidades y atributos llave, los tipos de relaciones y sus argumentos estructurales. Discutiremos los tipos de entidades débiles, y después demostraremos el uso de conceptos de ER en el diseño de la base de datos COMPAÑIA.

3.3.1 Entidades y Atributos

El objeto básico que el modelo ER representa es una **entidad**, la cual es "algo" del mundo real con una existencia independiente. Una entidad puede ser un objeto con existencia física -una persona, carro, casa, o empleado- o puede ser un objeto con existencia conceptual -una compañía, un trabajo, o curso universitario. Cada entidad tiene propiedades particulares que la describen, llamadas **atributos**. Por ejemplo, una entidad empleado puede describirse por el nombre, edad, dirección, salario y trabajo del empleado. Una entidad en particular tendrá un **valor** para cada uno de sus atributos. Los valores de los atributos que describen a cada entidad se vuelven una parte mayor de los datos almacenados en la base.

La Figura 3.3 muestra dos entidades y los valores de sus atributos. La entidad empleado tiene cuatro atributos: Nombre, Dirección, Edad y TelCasa; sus valores son "John Smith", "2311 Kirby, Houston, Texas", "55" y "713-749-2630", respectivamente.

La entidad compañía c1 tiene tres atributos: Nombre, Oficina Central, y Presidente; sus valores son: "Sunco Oil", "Houston" y "John Smith", respectivamente.

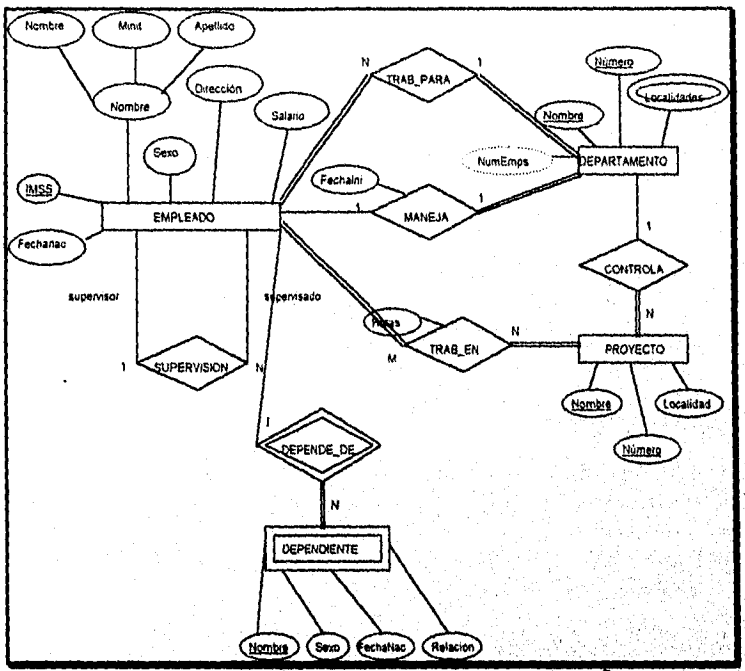


Figura 3.2 Esquema de un diagrama ER para la base de datos COMPAÑIA.

Tipos de Atributos. En el modelo ER existen diferentes tipos de atributos: *sencillos* contra *compuestos*; *un solo valor* y *multivaluados*; y *almacenados* contra *derivados*. Primero definiremos estos tipos de atributos e ilustraremos su uso a través de ejemplos. Después introduciremos el concepto de *valor nulo* de un atributo.

Atributos compuestos. Pueden dividirse en subpartes más pequeñas, las cuales representan atributos básicos con significados independientes de si mismos. Por ejemplo, el atributo dirección de la entidad empleado mostrado en la Figura 3.3 puede subdividirse en Calle, Ciudad, Estado y CP, con los valores "2311 Kirby", "Houston", "Texas", y "77001". Los atributos que no son divisibles se llaman **sencillos** o **atómicos**. Como se muestra en la Figura 3.4 los atributos compuestos pueden formar jerarquías, por ejemplo, CalleDirección puede subdividirse en tres atributos sencillos, Número, Calle y NumDepto. El valor de un atributo compuesto es la concatenación de los valores de sus atributos sencillos.

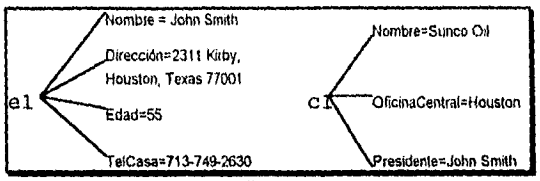


Figura 3.3 Dos entidades y sus valores atributos

Los atributos compuestos son útiles en situaciones de modelaje en las cuales el usuario se refiere algunas veces al atributo compuesto como una unidad pero en otras lo hace específicamente a sus componentes. Si el atributo compuesto es referenciado como un todo, no hay necesidad de subdividirlo en componentes. Por ejemplo, si no hay necesidad de referirse a los componentes individuales de una dirección (CP, Calle, y otros), entonces la dirección completa se designa como un atributo sencillo.

La mayoría de los atributos tienen un valor sencillo para una entidad particular; a tales atributos se les llama de **valor sencillo**. Por ejemplo, la Edad de una persona es un atributo de un sólo valor. En algunos casos un atributo puede tener un conjunto de valores para la misma entidad -por ejemplo, un atributo Colores para un carro, o CalifEscuela de una persona. Los carros con un color tienen un sólo valor; del mismo modo los carros de dos tonos tienen dos valores de Colores. Similarmente, una persona puede no tener ninguna calificación, otra puede tener una, y una tercera puede tener dos o más; así diferentes personas pueden tener diferente *número de valores* para el atributo CalifEscuela. A tales atributos se les llama **multivaluados**. Un atributo multivaluado puede tener límites inferiores y superiores en el número de valores para una entidad individual. Por ejemplo, el atributo Colores de un carro puede estar entre uno y cinco, si suponemos que un carro puede tener como máximo cinco colores.

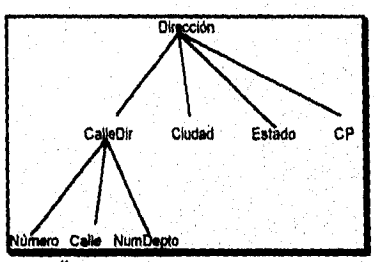


Figura 3.4 Jerarquía de un atributo compuesto

En algunos casos se relacionan dos (o más) valores de atributos -por ejemplo, los atributos Edad y FechaNac de una persona. Para una entidad de persona en particular, el valor de Edad puede determinarse a partir de la fecha actual (hoy) y el valor de FechaNac. El atributo Edad es por lo tanto un **atributo derivado** y se dice ser **derivable del** atributo FechaNac, al cual se le llama **atributo almacenado**. Algunos valores de atributos pueden derivarse de *entidades relacionadas*; por ejemplo, la entidad NumEmpleados de

departamento puede derivarse contando el número de empleados relacionados a (trabajando para) ese departamento.

En algunos casos una entidad particular puede no tener ningún valor aplicable de un atributo. Por ejemplo, el atributo NumDepto de una dirección se aplica sólo a direcciones que están en apartamentos y no para otros tipos de residencias tales como casas propias. Similarmente, un atributo CalifEscuela se aplica sólo a personas con calificaciones. En tales situaciones, se crea un valor especial llamado **nulo**. Una dirección con casa propia tendría nulo en el atributo Numdepto, y una persona sin calificaciones tendría nulo en CalifEscuela. El nulo también puede usarse si no conocemos el valor de un atributo para una entidad en particular -por ejemplo, en la Figura 3.3, si no conocemos el número telefónico de "John Smith". El significado anterior del tipo nulo *no es aplicable*, aunque el significado del último sea *desconocido*. La categoría desconocida de nulo puede clasificarse en dos casos. El primero cuando se conoce que el valor del atributo existe pero es *equivocado* -por ejemplo, si el atributo *Peso* es listado nulo. El segundo cuando *no se conoce* si el valor del atributo existe -por ejemplo, si el atributo TelCasa de una persona es nulo.

3.3.2 Tipos de Entidades, Conjuntos de Valores y Atributos Llave

Una base de datos usualmente contiene grupos de entidades que son similares. Por ejemplo, una compañía que emplea a cientos de personas puede desear almacenar información similar concerniente a cada uno de los empleados. Las entidades empleados comparten los mismos atributos, pero cada entidad tiene sus propios valores para cada atributo. Un **tipo de entidad** define un conjunto de entidades que tienen los mismos atributos. Cada tipo de entidad en la base de datos se describe por un nombre y una lista de atributos. La Figura 3.5 muestra dos tipos de entidades, llamadas EMPLEADO y COMPAÑIA, y una lista de atributos de cada una. También se ilustran algunas entidades individuales de cada una, junto con los valores de sus atributos.

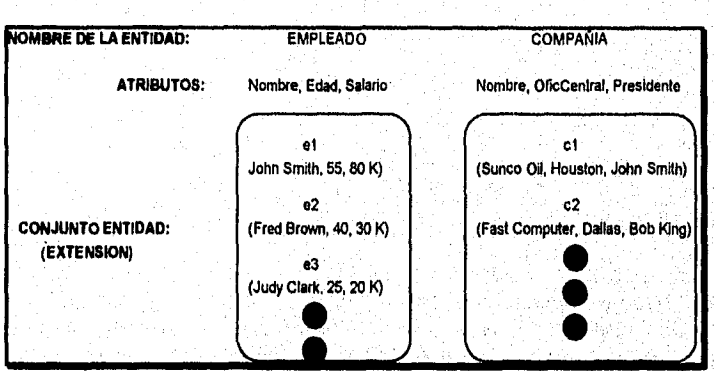


Figura 3.5 Dos tipos de entidades y algunos de los miembros de cada entidad.

En diagramas ER un tipo de entidad se representa (como en la Figura 3.2) como una caja rectangular encerrando el nombre del tipo de entidad. Los nombres de los atributos se encierran en ovalos y se conectan a su tipo de entidad mediante líneas rectas. Los atributos compuestos se conectan a sus atributos componentes mediante líneas rectas. Los atributos multivaluados se despliegan en óvalos dobles.

Un tipo de entidad describe el **esquema** o **intensión** de un *conjunto de entidades* que comparten la misma estructura. Las entidades individuales de un tipo en particular se agrupan en una **colección** o **conjunto de entidades**, a las cuales también se les llama **extensión** del tipo de entidad.

Atributos Llave de un Tipo de Entidad. Un argumento importante en las entidades de un tipo es la **llave** o **unicidad** en los atributos. Una entidad usualmente tiene un atributo cuyos valores son distintos en cada entidad individual. A tal atributo se le llama **atributo llave**, y sus valores pueden usarse para identificar a cada entidad de manera única. En la Figura 3.5 el atributo NOMBRE es una llave de la entidad COMPAÑIA, porque a ninguna compañía se le permite tener el mismo nombre que otra. Para el tipo de entidad PERSONA, una llave típica es el NúmeroIMSS. Algunas veces, varios atributos en conjunto forman una llave, significando que la *combinación* de los valores de los atributos debe ser distinta en cada entidad individual. Un conjunto de atributos que posee esta propiedad puede agruparse en un atributo compuesto, el cual se vuelve un atributo llave del tipo de entidad. Como se ilustra en la Figura 3.2, en notación ER, cada atributo llave tiene su nombre **subrayado** dentro del óvalo.

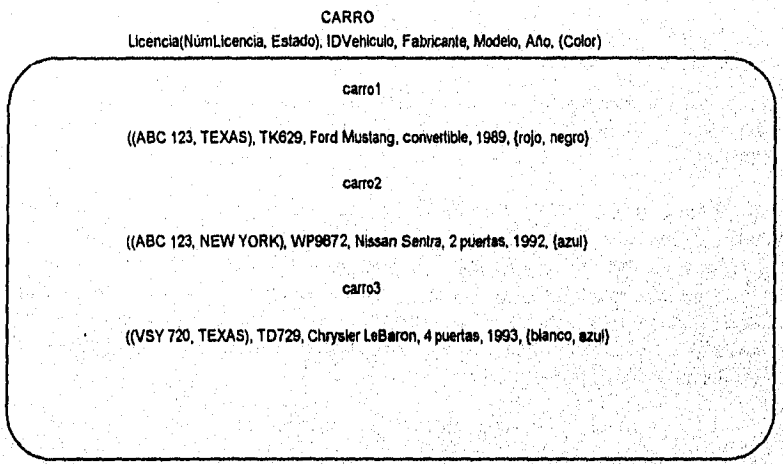


Figura 3.6 La entidad tipo CARRO. Los atributos multivaluados se muestran entre corchetes {}. Los componentes de un atributo compuesto se muestran entre paréntesis.

Especificar que un atributo es una llave de un tipo de entidad significa que la propiedad de unicidad precedente debe retenerse en *cada extensión* del tipo de entidad. Por lo tanto, es

un argumento que prohibe a cualesquiera dos entidades de tener el mismo valor al mismo tiempo para el atributo llave. No es propiedad de una extensión en particular; por el contrario, es un argumento en *todas la extensiones* del tipo de entidad. Este argumento de llave se deriva de las propiedades del minimundo que representa la base de datos.

Algunos tipos de entidades tienen *más de un* atributo llave. Por ejemplo, cada uno de los atributos IDVehúculo y Licencia del tipo de entidad CARRO (Figura 3.6) es una llave. El atributo Licencia es un ejemplo de una llave compuesta formada por dos atributos simples, NúmeroLicencia y Estado, ninguno de los cuales es llave por si mismo.

Conjuntos de Valores (Dominios) de Atributos. Cada atributo de un tipo de entidad se asocia con un **conjunto de valores** (o **dominio**), el cual especifica al conjunto de valores que puede asignarse a ese atributo para cada entidad individual. En la Figura 3.5, si el rango de edades permitido para empleados está entre 16 y 70, podemos especificar el conjunto de valores del atributo Edad de EMPLEADO como el conjunto de enteros entre 16 y 70. Igualmente, podemos especificar el conjunto de valores para el atributo Nombre como el conjunto de caracteres alfabéticos separados por caracteres en blanco; etc. Los conjuntos de valores no se despliegan en diagramas ER.

Matemáticamente, un atributo A de entidad tipo E cuyo valor es V puede definirse como una **función** de E al conjunto poder V:

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

{TelCasa({Tel(CodArea, NumTel)},
Dirección(CalleDir(NumCalle, NumDepto),
Ciudad, Estado, CP)}

Figura 3.7 Atributo de valor compuesto. TelDirección con componentes multivaluados y compuestos.

Nos referiremos al valor de un atributo A para una entidad e como A(e). La definición anterior cubre a atributos univaluados y multivaluados, así como a los nulos. Un valor nulo se representa por el conjunto vacío. Para atributos univaluados, A(e) se restringe a ser un elemento único para cada entidad e en E mientras que no hay restricción en atributos multivaluados. Para un atributo compuesto A, el conjunto de valores V es el producto Cartesiano de P(V1), P(V2), ..., P(Vn), donde V1, V2, ..., Vn son el conjunto de valores del componente simple de atributos que forman A:

$$V=(PV1) X (PV2) X...X(PVn)$$

nótese que los atributos compuestos y multivaluados pueden anidarse de manera arbitraria. Podemos representar el anidamiento arbitrario agrupando componentes de un atributo compuesto entre paréntesis () y separando a los componentes con comas, y desplegando atributos multivaluados entre corchetes {}. Por ejemplo, si una persona puede tener más

de una residencia y cada una tiene varios teléfonos, puede especificarse un atributo TelDir para un tipo de entidad PERSONA como se muestra en la Figura 3.7.

Diseño Conceptual Inicial de la Base de Datos COMPAÑIA. Ahora podemos definir los tipos de entidades para la base de datos COMPAÑIA descrita en la Sección 3.2. Después de definir varios tipos de entidades y sus atributos, *refinaremos* nuestro diseño en la siguiente sección (después de introducir el concepto de relación). De acuerdo a los requerimientos listados en la Sección 3.2, podemos identificar cuatro tipos de entidades - una correspondiente a acada uno de los cuatro elementos en la especificación (véase Figura 3.8):

1. Una entidad tipo DEPTO con atributos Nombre, Número, Localidades, Gerente y FechaIniGte. Localidades es el único atributo multivaluado. Podemos especificar que cada Nombre y Número es un atributo llave, porque cada uno fue especificado como único.

2. Una entidad tipo PROYECTO con atributos Nombre, Número, Localidad, y DeptoControl. Cada nombre y Número son atributos llave.

3. Una entidad tipo EMPLEADO con atributos Nombre, IMSS, Sexo, Dirección, Salario, Fechanac, Depto, y Supervisor. Nombre y Dirección pueden ser atributos compuestos; sin embargo, esto no se especificó en los requerimientos. Debemos regresar a los usuarios para ver si cualquiera de ellos se referirá a los componentes individuales de Nombre -Nombre, Apellido Paterno y Materno- o de Dirección.

4. Una entidad tipo **DEPENDIENTE** con atributos Empleado, NomDependiente, Sexo, FechaNac, y Relación (al empleado).

Hasta ahora, no hemos representado el hecho de que un empleado pueda trabajar en varios proyectos, ni hemos representado el número de horas a la semana que un empleado trabaja en cada proyecto. Esto puede representarse por un atributo compuesto multivaluado de **EMPLEADO** llamado **TrabajaEn** con componentes sencillos (Proyecto, Horas). Alternativamente, puede representarse como un atributo compuesto multivaluado de **PROYECTO** llamado **Trabajadores**, con componentes sencillos (Empleado, Horas). Seleccionemos la primer alternativa de la Figura 3.8, la cual muestra cada una de las entidades descritas anteriormente. El atributo Nombre de **EMPLEADO** se muestra como un atributo compuesto, presumiblemente después de la consulta con los usuarios.

En la Figura 3.8 hay varias *relaciones implícitas* entre los distintos tipos de entidades. En efecto, cada vez que un atributo de un tipo de entidad se refiere a otro tipo de entidad, existen algunas relaciones. Por ejemplo, el atributo Gerente de **DEPARTAMENTO** se refiere a un empleado que maneja el departamento; el atributo DeptoControl de **PROYECTO** se refiere al departamento que controla el proyecto; el atributo Supervisor de **EMPLEADO** se refiere al departamento para el cual trabaja el empleado; etc. En el modelo ER, estas referencias no deben representarse como atributos pero sí como

relaciones, las cuales se definen en la siguiente sección. El esquema de la base de datos COMPAÑIA se redefine en la Sección 3.3.5 para representar explícitamente las relaciones. En el diseño inicial de tipos de entidades, las relaciones se capturan en forma de atributos. Cuando se refina el diseño, estos atributos se convierten en relaciones entre tipos de entidades.

3.3.3 Relaciones, Roles y Argumentos Estructurales

Tipos de Relaciones e Instancias. Un **tipo de relación** R entre n tipos de entidades E1, E2, ..., En define un conjunto de asociaciones entre entidades de estos tipos. Matemáticamente, R es un conjunto de **instancias de relaciones** ri, donde cada ri asocia n entidades (e1, e2, ..., en), y cada entidad ej en ri es miembro de una entidad tipo Ej, $1 \leq j \leq n$. Por lo tanto, un tipo de relación es una relación matemática en E1, E2, ..., En o alternativamente puede definirse como un subconjunto del producto Cartesiano E1 X E2 X ... X En. Cada una de los tipos de entidades E1, E2, ..., En se dice **participar** en la relación tipo R, y similarmente cada una de las entidades individuales e1, e2, ..., en se dice participar en la instancia de relación ri=(e1, e2, ..., en).

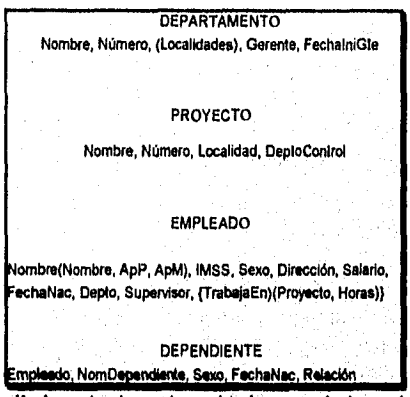


Figura 3.8 Diseño preliminar de tipos de entidades para la base de datos descrita en la Sección 3.2. Los atributos multivaluados se muestran entre llaves {}. Los componentes de un atributo compuesto se muestran entre paréntesis ().

Informalmente, cada instancia de relación ri en R es una asociación de entidades, donde la asociación incluye exactamente una entidad de cada tipo de entidad participante. Cada instancia de relación ri representa el hecho de que las entidades que participan en ri se relacionan de alguna manera en la situación correspondiente del minimundo.

Por ejemplo, consideremos un tipo de relación TRABAJA_PARA entre los dos tipos de entidades EMPLEADO y DEPARTAMENTO, el cual asocia cada empleado con el departamento en que trabaja cada empleado. Cada instancia de relación en TRABAJA_PARA asocia una entidad empleado y una entidad departamento. La Figura

3.9 ilustra este ejemplo, donde cada instancia de relación ri se muestra conectada a las entidades empleado y departamento que participan en ri. En el minimundo representado en la Figura 3.9, los empleados e1, e3 y e6 trabajan para el departamento d1; e2 y e4 trabajan para d2; y e5 y e7 trabajan para d3.

En diagramas ER, los tipos de relaciones se despliegan como cajas en forma de diamante, las cuales se conectan con líneas rectas a las cajas rectangulares representando los tipos de entidad participante. El nombre de la relación se despliega en la caja con forma de diamante.

Grado de un Tipo de Relación. Es el número de tipos de entidades participantes. Por lo tanto, el tipo de relación TRABAJA_PARA es de grado dos. Un tipo de relación de grado dos se llama **binaria**, y una de grado tres es **ternaria**. Un ejemplo de relación ternaria es ALMACEN, mostrada en la Figura 3.10, donde cada instancia de relación ri asocia a tres entidades -un proveedor s, una parte p, y un proyecto j- entendiéndose que s provee una parte p para el proyecto j. Las relaciones pueden ser de cualquier grado, pero las que ocurren con mayor frecuencia son las binarias.

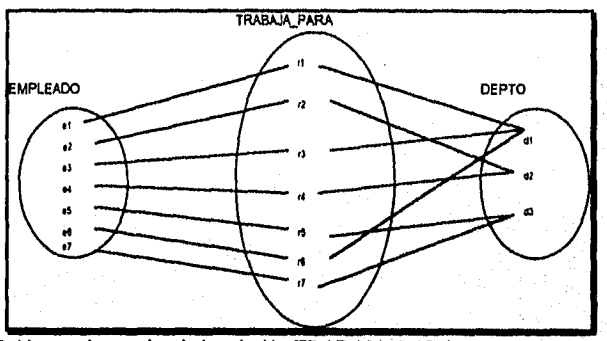


Figura 3.9 Algunas instancias de la relación TRABAJA_PARA.

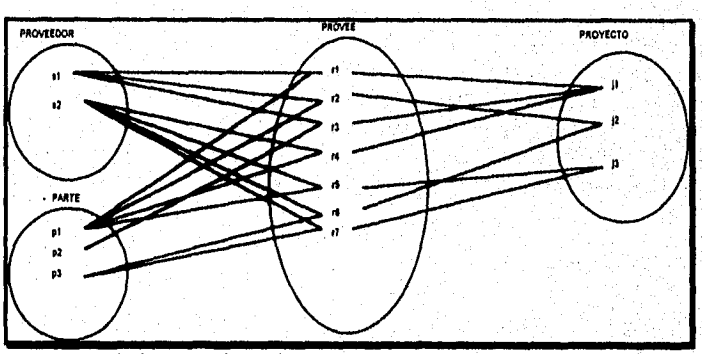


Figura 3.10 La relación ternaria PROVEE.

Relaciones como Atributos. A veces es conveniente pensar en las relaciones como un tipo de relación en términos de atributos. Consideremos la relación TRABAJA_PARA de la Figura 3.9. Podemos pensar de un atributo llamado Departamento de la entidad EMPLEADO cuyo valor para cada entidad empleado es la *entidad departamento* para la que el empleado trabaja. Por lo tanto, el conjunto de valores de este atributo Departamento es el *conjunto de todas las entidades* DEPARTAMENTO. Esto es lo que se hizo en la Figura 3.8 cuando especificamos el diseño inicial de la entidad tipo EMPLEADO para la base de datos COMPAÑIA. Sin embargo, cuando pensamos de una relación binaria como un atributo, siempre tenemos dos opciones. En este ejemplo, la alternativa es pensar en un atributo multivaluado de Empleados de la entidad tipo DEPARTAMENTO cuyos valores para cada entidad departamento es el *conjunto de entidades empleados* que trabajan para ese departamento. El conjunto de valores de este atributo Empleados es el conjunto de la entidad EMPLEADO. Cualquiera de estos dos atributos -Departamento de EMPLEADO o Empleados de DEPARTAMENTO- pueden representar la relación de tipo TRABAJA_PARA. Si ambas se representan, están restringidas a ser una a la inversa de la otra.

Nombres de Rol y Relaciones Recursivas. Cada tipo de entidad que participa en un tipo de relación juega un **rol** particular en la relación. El **nombre del rol** significa el papel que una entidad participante del tipo de entidad juega en cada instancia de relación. Por ejemplo, en el tipo de relación TRABAJA_PARA, EMPLEADO juega el papel de *empleado* o *trabajador* y DEPARTAMENTO juega el papel de *departamento* o *empleador*.

Los nombres de rol no son necesarios en tipos de relaciones donde todos los tipos de entidades participantes son distintos, debido a que cada nombre de tipo de entidad puede usarse como el nombre del rol. Sin embargo, en algunos casos el *mismo* tipo de entidad participa más de una vez en un tipo de relación en *diferentes roles*. En tales casos el nombre del rol se vuelve esencial para distinguir el significado de cada participación. A tales tipos de relación se les llama **recursivos**, la Figura 3.11 muestra un ejemplo. El tipo de relación SUPERVISION relaciona un empleado a supervisor, donde ambas entidades son miembros del mismo tipo de entidad EMPLEADO. Por lo tanto, el tipo de entidad EMPLEADO *participa dos veces* en SUPERVISION: una en el *rol de supervisor* (o jefe), y otra en el *rol de supervisado* (o subordinado). Cada instancia de relación ri en SUPERVISION asocia dos entidades empleados ek y ek, uno de los cuales juega el papel de supervisor y el otro de supervisado. En la Figura 3.11, las líneas marcadas "1" representan el rol supervisor, y aquéllas marcadas con "2" representan el rol supervisado; por lo tanto, e1 supervisa a e2 y e3; e4 supervisa a e6 y e7; y e5 supervisa a e1 y e4.

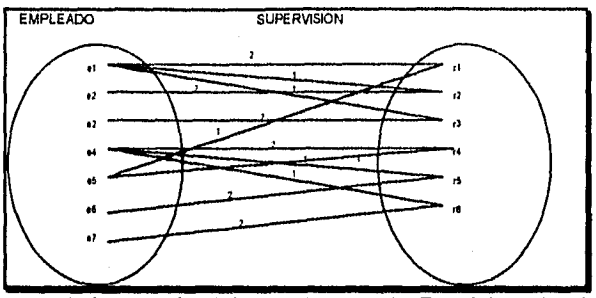


Figura 3.11 La relación recursiva SUPERVISION: EMPLEADO juega los dos roles (1) de supervisor y (2) de supervisado.

Argumentos en Tipos de Relación. Los tipos de relación usualmente tienen ciertos argumentos que limitan las posibles combinaciones de entidades que pueden participar en las instancias de relación. Estos argumentos se determinan de la situación del mininundo al que representa la relación. Por ejemplo, en la Figura 3.9, si la compañía tuviera la política de que un trabajador sólo puede trabajar para un departamento, podríamos describir este argumento en el esquema. Podemos distinguir dos tipos principales de argumentos de relación: el radio de cardinalidad y la participación.

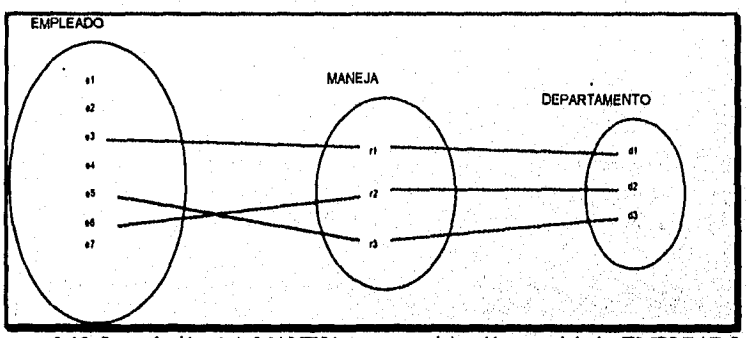


Figura 3.12 La relación 1:1 MANEJA, con participación parcial de EMPLEADO y participación total de DEPARTAMENTO.

El **radio de cardinalidad** especifica el número de instancias de relación en la que una entidad puede participar. La relación binaria TRABAJA_PARA tipo DEPARTAMENTO:EMPLEADO es de radio de cardinalidad 1:N, significando que cada departamento puede relacionarse con numerosos empleados, pero un empleado puede relacionarse con (trabajar para) un sólo departamento. Los radios de cardinalidad más comunes en relaciones binarias son de 1:1, 1:N y M:N.

Un ejemplo de relación binaria 1:1 es el tipo MANEJA (Figura 3.12), la cual relaciona a una entidad departamento con el empleado que maneja el departamento. Esto representa

los argumentos del minimundo de que un empleado puede sólo manejar un departamento y que un departamento tiene un sólo gerente. El tipo de relación TRABAJA_PARA (Figura 3.13) es de radio de cardinalidad M:N, si el rol es que un empleado puede trabajar en diferentes proyectos y que diferentes empleados pueden trabajar en un proyecto.

El **argumento de participación** especifica si la existencia de una entidad depende en si está relacionada a otra entidad vía el tipo de relación. Hay dos tipos de argumentos de participación -total y parcial- los cuales ilustraremos. Si la politica de una compañia establece que *cada* empleado debe trabajar para un departamento, entonces una entidad empleado sólo puede existir si participa en la instancia de relación TRABAJA_PARA (Figura 3.9). A la participación de EMPLEADO en TRABAJA_PARA se le llama **total**, y significa que cada entidad en "el conjunto total" de entidades empleado debe relacionarse a una entidad departamento vía TRABAJA_PARA. La participación total es llamada a veces **dependencia de existencia**. En la Figura 3.12 no se espera que cada empleado maneje un departamento, así la participación de EMPLEADO en la relación de tipo MANEJA es **parcial**, significa que *algo* o "parte de el conjunto de" las entidades empleado se relaciona a una entidad departamento vía MANEJA, pero no necesariamente todos. Nos referiremos al radio de cardinalidad y argumentos de participación, en conjunto, como los **argumentos estructurales** de un tipo de relación.

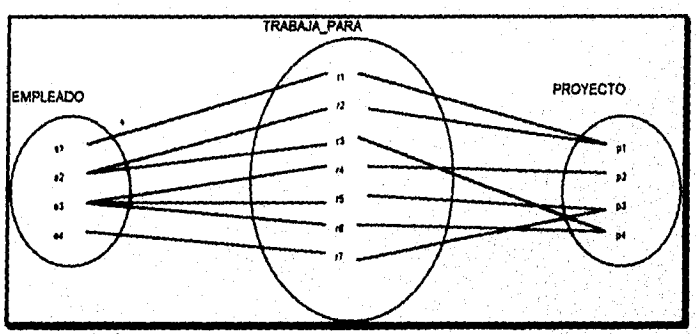


Figura 3.13 Relación M:N TRABAJA_PARA.

Los radios de cardinalidad de una relación binaria se despliegan en diagramas ER mostrando 1, M y N como se muestra en la Figura 3.2. La participación total se muestra como una línea doble que conecta la participación del tipo de entidad con la relación, mientras que la participación parcial se representa con una sola línea.

Atributos de Tipos de Relación. Los tipos de relación también pueden tener atributos, similar a los tipos de entidad. Por ejemplo, registrar el número de horas a la semana que un empleado trabaja en un proyecto, podemos incluir un atributo Horas para el tipo de relación TRABAJA_PARA de la Figura 3.13. Otro ejemplo es incluir la fecha en la cual un gerente inició en un departamento a través de un atributo FechaIni para el tipo de relación MANEJA de la Figura 3.12.

Notemos que los atributos de los tipos de relación 1:1 o 1:N pueden emigrar a uno de los tipos de entidades participantes. Por ejemplo, el atributo FechaIni para la relación MANEJA puede ser un atributo de EMPLEADO o DEPARTAMENTO -a pesar de que conceptualmente pertenece a MANEJA. Esto se debe a que MANEJA es una relación 1:1, así cada departamento o empleado participa en al menos una instancia de relación. Por lo tanto, el valor del atributo FechaIni puede determinarse independientemente; ya sea por la entidad de departamento o empleado (gerente).

Para un tipo de relación 1:N, un atributo de relación puede emigrar *sólo* al tipo de entidad del lado N de la relación. Por ejemplo, en la Figura 3.9, si la relación TRABAJA_PARA también tiene un atributo FechaIni que indica cuando un empleado empezó a trabajar para un departamento, este atributo puede incluirse como un atributo de EMPLEADO. Esto es porque la relación es 1:N, así cada entidad empleado participa en al menos una instancia de relación en TRABAJA_PARA. En ambos tipos de relación 1:1 y 1:N, se determina subjetivamente la decisión de dónde debe colocarse el atributo de relación -como atributo de tipo de relación o como atributo de tipo de entidad participante- por el esquema del diseñador.

Para los tipos de relaciones M:N, algunos atributos pueden determinarse por la *combinación de entidades participantes* en una instancia de relación, y no por una sola. Tales atributos deben especificarse como atributos de relación. Un ejemplo es el atributo Horas de la relación M:N TRABAJA_PARA (Figura 3.13); el número de horas que un empleado trabaja en un proyecto se determina por una combinación empleado-proyecto y no separadamente por cualquier entidad.

3.3.4 Tipos de Entidades Débiles

Algunos tipos de entidad pueden no tener ningún atributo llave en sí mismos; a estos se les llama **tipos de entidades débiles**. Las entidades que pertenecen a un tipo de entidad débil se identifican por estar relacionados a entidades específicas de otro tipo en combinación con algunos de los valores de sus atributos. A este otro tipo de entidad lo llamamos **identificación de propietario**, y a la relación que asocia a un tipo de entidad débil con su propietario se le llama **relación de identificación** del tipo de entidad débil. Un tipo de entidad débil siempre tiene un argumento de participación *total* (existencia-dependencia) con respecto a su relación de identificación, debido a que una entidad débil no puede identificarse sin una entidad propietaria. Sin embargo, no toda existencia de dependencia resulta en un tipo de entidad débil. Por ejemplo, una entidad LICENCIA_CONDUCIR no puede existir a menos que se relacione a una entidad PERSONA, a pesar de que tenga su propia llave (NUM_LICENCIA), por lo tanto no es una entidad débil.

Consideremos el tipo de entidad DEPENDIENTE, relacionado a EMPLEADO, el cual se usa para rastrear a los dependientes de cada empleado a través de la relación 1:N. Los atributos de DEPENDIENTE son NombreDep, FechaNac, Sexo, y Relación (con el empleado). Dos dependientes de distintos empleados pueden tener los mismos valores para NombreDep, FechaNac, Sexo, y Relación, pero ser entidades distintas. Se identifican

como entidades distintas sólo después de determinar la entidad empleado a la cual se relacionan. Cada entidad empleado se dice **poseer** las entidades dependientes que se relacionan a ella.

Una entidad de tipo débil normalmente tiene una **llave parcial**, la cual es el conjunto de atributos que pueden identificar de manera única entidades débiles relacionadas a la *misma entidad propietaria*. En nuestro ejemplo, si asumimos que ningun par de dependientes del mismo empleado tendrán el mismo nombre, los atributos NombreDepto y NombreDependiente de DEPENDIENTE serán la llave parcial.

En diagramas ER, una entidad de tipo débil y su relación de identificación se distinguen rodeando sus cajas con líneas dobles (véase figura 3.2). El atributo llave parcial está subrayado con una línea punteada.

Los tipos de entidades débiles pueden representarse como atributos compuestos, multivaluados. En el ejemplo anterior, podríamos especificar un atributo multivaluado compuesto Dependientes para EMPLEADO, compuesto de los atributos NombreDependiente, FechaNac, Sexo y Relación. La elección de cuál representación usar se hace por el diseñador de la base de datos. Un acercamiento es seleccionar el tipo de entidad débil si tiene muchos atributos y participa independientemente en tipos de relación diferentes a los que identifican el tipo de relación. En general, pueden definirse cualquier número de niveles de tipos de entidad débil; un tipo de entidad propietario puede por sí misma ser un tipo de entidad débil. Además, un tipo de entidad débil puede tener más de un tipo de identificador de entidad y una relación de identificación de grado mayor a dos.

3.3.5 Refinación del Diseño ER para la Base de Datos Compañía

Ahora podemos refinar el diseño de la Figura 3.8 cambiando los atributos que representan relaciones en tipos de relaciones. El radio de cardinalidad y argumento de participación de cada tipo de relación se determinan de los requerimientos listados en la Sección 3.2. Si algun radio de cardinalidad o dependencia no puede determinarse de los requerimientos, debe cuestionarse a los usuarios para determinar estas propiedades estructurales.

En nuestro ejemplo, especificamos los siguientes tipos de relación:

1. MANEJA, relación de tipo 1:1 entre EMPLEADO y DEPARTAMENTO. La participación de EMPLEADO es parcial. La participación de DEPARTAMENTO no es clara de los requerimientos. Preguntemos a los usuarios, quienes dirán que un departamento debe tener un gerente todo el tiempo, lo cual implica participación total. El atributo FechaIni se asigna a este tipo de relación.

2. TRABAJA_PARA, relación de tipo 1:N entre DEPARTAMENTO y EMPLEADO. Ambas particiapaciones son totales.

3. CONTROLA, relación 1:N entre DEPARTAMENTO y PROYECTO. La participación de PROYECTO es total, mientras que DEPARTAMENTO se determina ser parcial, después de consultar a los usuarios.

4. SUPERVISION, relación de tipo 1:N entre EMPLEADO (en el rol de supervisor) y EMPLEADO (en el rol de supervisado). Ambas participaciones son parciales, después de que los usuarios indican que no todo empleado es un supervisor y no todo empleado tiene un supervisor.

5. TRABAJA_PARA, determinada a ser una relación tipo M:N con atributo Horas, después de que los usuarios indican que un proyecto puede tener varios empleados trabajando en él. Ambas participaciones se determinan ser totales.

6. DEPENDE_DE, relación tipo 1:N entre EMPLEADO y DEPENDIENTE, el cual es también la identificación de relación para la entidad débil tipo DEPENDIENTE. La participación de EMPLEADO es parcial, mientras que la de DEPENDIENTE es total.

Después de especificar los seis tipos de relaciones anteriores, eliminemos de los tipos de entidades en la Figura 3.8 todos los atributos que han sido refinados en relaciones. Estas incluyen Gerente y FechaIniGte de DEPARTAMENTO; Deptocontrol de PROYECTO; Departamento, Supervisor, y TrabajaEn de EMPLEADO; y Empleado de DEPENDIENTE. Cuando diseñamos el esquema conceptual de una base de datos es importante tener la menor redundancia posible. Si se desea alguna redundancia a nivel de almacenamiento o a nivel de vista del usuario, puede introducirse posteriormente.

3.4 Notación de Diagramas Entidad-Relación (ER)

Las Figuras 3.9 a 3.13 ilustran los tipos de entidades y relaciones desplegando sus extensiones -las entidades individuales e instancias de relación. En los diagramas ER se representa el énfasis en esquemas en vez de instancias. Esto es más útil debido a que un esquema de base de datos cambia raramente, mientras que la extensión cambia frecuentemente. Además, el esquema es usualmente más fácil de desplegar que la extensión de una base de datos, debido a que es mucho más pequeño.

La Figura 3.2 despliega el **esquema de la base de datos ER** COMPAÑIA como un diagrama ER. Los tipos de entidades tales como EMPLEADO, DEPARTAMENTO, y PROYECTO se muestran en cajas rectangulares. Los tipos de relación tales como TRABAJA_PARA, MANEJA, CONTROLA, Y TRABAJA_EN se muestran en cajas con forma de diamante unidos a las entidades participantes con líneas rectas. Los atributos se muestran en óvalos, y cada atributo está unido a su tipo de entidad o relación por una línea recta. Los atributos componentes de un atributo compuesto se unen a los óvalos que representan el atributo compuesto, como se mostró por el atributo nombre de EMPLEADO. Los atributos multivaluados se muestran en óvalos dobles, como se mostró por el atributo Localidades de DEPARTAMENTO. Los atributos llave tienen sus nombres

subrayados. Los atributos derivados se muestran en óvalos punteados, como se muestra por el atributo NumEmpleados de DEPARTAMENTO.

Los tipos de entidades débiles se distinguen por colocarse en rectángulos dobles y teniendo su identificación de relación colocada en diamantes dobles, como se ilustró por la entidad DEPENDIENTE y el tipo de relación DEPENDEN_DE. La llave parcial del tipo de entidad débil está subrayada con *líneas punteadas.*

En la Figura 3.2 el radio de cardinalidad de cada relación de tipo *binario* se especifica colocando un 1, M o N en cada esquina participante. El radio de cardinalidad de DEPARTAMENTO:EMPLEADO en MANEJA es 1:1, mientras que es 1:N para DEPARTAMENTOS:EMPLEADO en TRABAJA_PARA, y es M:N para TRABAJAEN. El argumento de participación se especifica por una línea sencilla para participación parcial y línea doble para participación total (existencia dependencia).

En la Figura 3.2 mostramos los nombres rol para la relación tipo SUPERVISION porque el tipo de entidad EMPLEADO juega ambos roles en la relación. Notemos que la cardinalidad es 1:N de supervisor a supervisado porque, por otro lado, cada empleado en el rol de supervisado tiene al menos un supervisor directo, nientras que un empleado en el rol de supervisor puede supervisar a cero o más empleados.

Una alternativa de la notación ER para especificar argumentos estructurales involucra asociar un par de enteros (min, max) con cada *participación* de una entidad tipo E en una relación tipo R, donde $0 \leq min \leq max$ y $max \geq 1$. Los números significan que, por cada entidad e en E, debe participar en al menos min y máximo max instancias de relación en R *en todo momento.* En este método, min=0 implica participación parcial, mientras que min > 0 implica participación total. La Figura 3.14 despliega el esquema COMPAÑIA, usando esta notación. Este método es más preciso, y podemos usarlo fácilmente para especificar argumentos estructurales para tipos de relaciones *de cualquier grado.*

La Figura 3.14 también despliega todos los nombres de los roles para el esquema de la base de datos COMPAÑIA. La Figura 3.15 resume las convensiones para diagramas ER.

3.5 Nombramiento Apropiado de los Constructores del Esquema

La elección de nombres para tipos de entidad, atributos, tipos de relaciones, y (particularmente) roles no siempre es directo. Se deben elegir nombres que convengan, tanto como sea posible, el significado unido a los diferentes constructores en el esquema. Elegimos usar *nombres similares* para los tipos de entidades, en vez de plurales, debido a que el nombre del tipo de entidad se aplica a cada entidad individual perteneciente a ese tipo de entidad. En nuestros diagramas ER, usaremos la convensión de que los nombres del tipo de entidad y relaciones están en mayúsculas, los nombres de atributos inician con mayúscula, y los nombres de roles en minúsculas.

Como práctica general, dada una descripción narrativa de los requerimientos de la base de datos, los *nombres* que aparecerán en la narración tenderán a dar lugar a nombres de tipos de entidad, y los *verbos* tenderán a indicar nombres de tipos de relación. Los nombres de atributos generalmente provienen de nombres adicionales que describen los nombres correspondientes a tipos de entidades. Otra consideración de nombramiento involucra el elegir nombres de relaciones que hagan legible al diagrama ER de izquierda a derecha y de arriba a abajo.

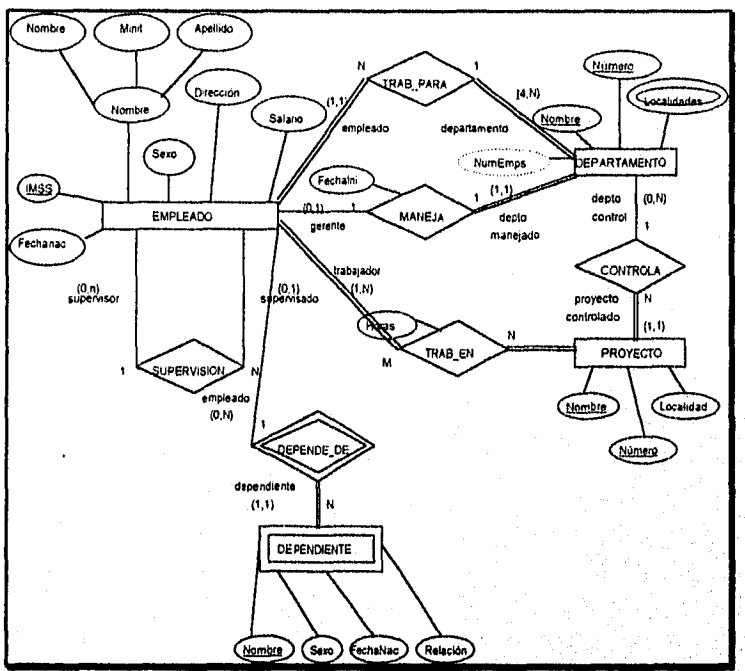


Figura 3.14 Diagrama ER para el esquema COMPAÑIA, con todos los nombres rol incluidos y los argumentos estructurales de relación especificadas usando notación alterna (min, max).

3.6 Tipos de Relaciones de Grado Mayor a Dos

En general, en un diagrama ER un tipo de relación R de grado n tendrá n esquinas, conectando una R a cada tipo de entidad participante.

La Figura 3.16(b) muestra un diagrama ER para los tres tipos de relaciones binarias PUEDE_PROPORCIONAR, USA y PROPORCIONA. En general, una relación ternaria representa más información que las tres relaciones de tipo binario. Consideremos las tres relaciones binarias. Supongamos que PUEDE_PROPORCIONAR, entre proveedor y

parte, incluye una instancia (s,p) cada que un proveedor *s* puede *proporcionar* una parte *p* (a cualquier proyecto); USA, entre PROYECTO y PARTE incluye una instancia (j,p) cada que el proyecto *usa* una parte p; y PROPORCIONA, entre PROVEEDOR y PROYECTO, incluye una instancia (s,j) cada que el proveedor s *proporciona alguna parte* al proyecto j. La existencia de las tres relaciones no necesariamente implica que una instancia (s,j,p) y (s,j) existan en la relación ternaria PROPORCIONA. A menudo es engañoso decidir si una relación en particular debe representarse como un tipo de relación de grado n o debe romperse en varios tipos de relación de menor grado. El diseñador debe basar esta decisión en la semántica o significado de la situación en particular que está siendo representada. La solución típica es incluir la relación ternaria *más* una o más de las relaciones binarias, cuando sea necesario.

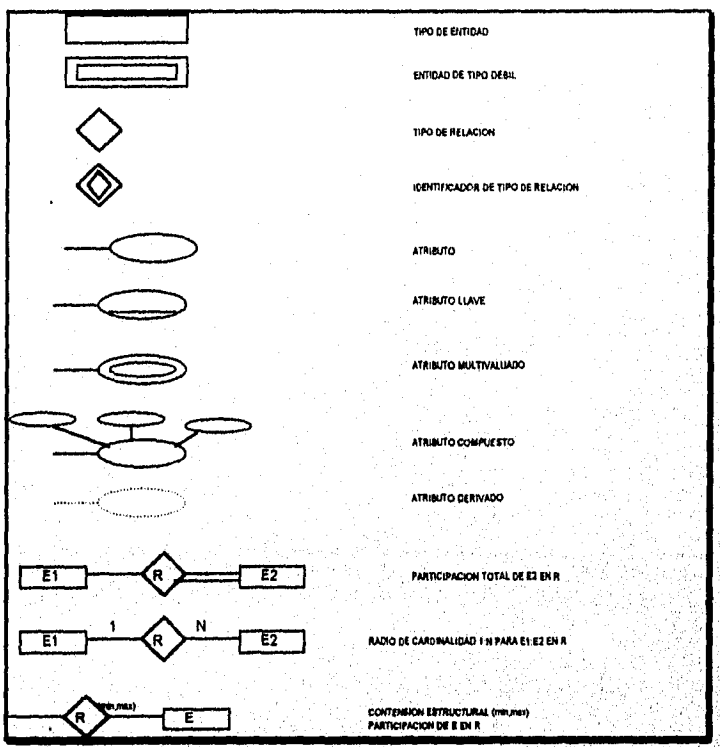


Figura 3.15 Resumen de notación de diagramas ER.

Algunas herramientas de diseño de bases de datos se basan en variantes del modelo ER que permiten sólo relaciones binarias. En este caso, una relación ternaria tal como PROPORCIONA debe representarse como una entidad de tipo débil; sin llave parcial y con tres relaciones de identificación. Los tres tipos de entidades participantes forman

juntas el tipo de entidad poseedora. Por lo tanto, mediante la combinación de sus tres entidades propietarias se identifica a una entidad de tipo débil.

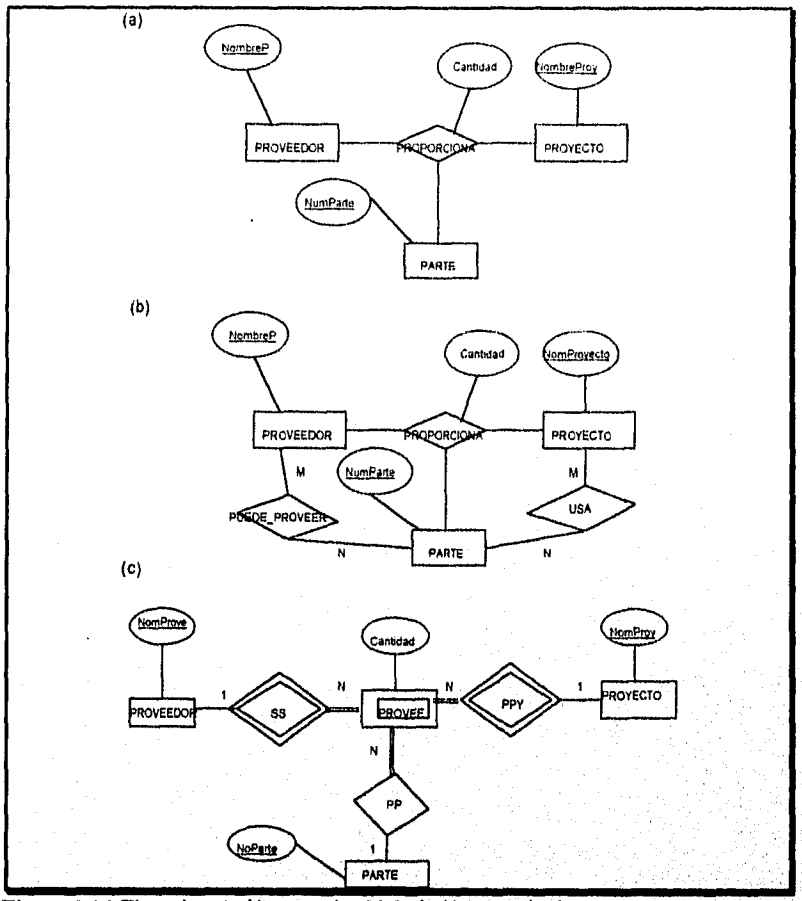
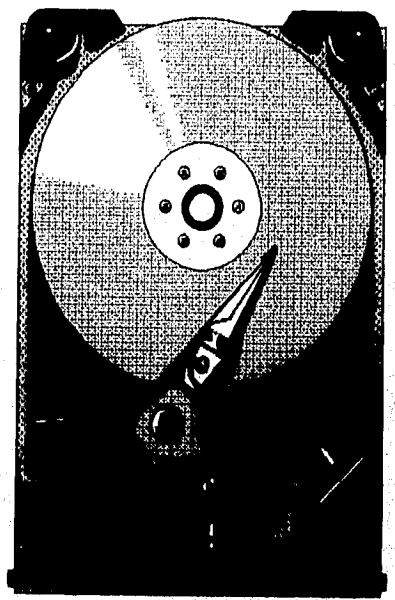


Figura 3.16 Tipos de relación ternaria. (a) Relación ternaria tipo **PROPORCIONA**. (b) Tres tipos de relación ternaria que no son equivalentes a la relación ternaria tipo **PROPORCIONA**. (c) **PROPORCIONA** representada como una entidad de tipo débil.

CAPITULO 4



Almacenamiento de Registros y Organización Primaria de Archivos

4.1 Introducción

La colección de información que conforma una base de datos computarizada debe de almacenarse físicamente en algún **medio de almacenamiento**. El software del DBMS puede entonces recuperar, actualizar y procesar esta información cuando sea necesario. La computadora almacena a través de una *jerarquía de almacenamiento* que incluye dos categorías principales:

- Almacenamiento primario: Esta categoría incluye medios de almacenamiento que pueden operarse directamente por el CPU de la computadora, tales como la memoria principal y memorias caches. El almacenamiento primario usualmente proporciona acceso a los datos pero está limitado a la capacidad de almacenamiento.

- Almacenamiento secundario: Incluye a discos magnéticos, discos ópticos, cintas y tambores que usualmente son de gran capacidad, menor costo, y proporcionan acceso más lento a los datos que los dispositivos primarios. Los datos en almacenamiento secundario no pueden procesarse directamente por el CPU; deben copiarse primero a almacenamiento primario.

Las bases de datos típicamente almacenan grandes cantidades de información que debe persistir por largos periodos de tiempo. Durante este periodo los datos se accesan y procesan repetidamente. Esto contrasta con la noción de estructuras de datos que persisten por un tiempo limitado durante la ejecución del programa, lo cual es común en los lenguajes de programación. La mayoría de las bases de datos se almacenan permanentemente en **discos magnéticos**, por las siguientes razones:

- Generalmente, las bases de datos son muy grandes para alojarse en la memoria principal.

- Las circunstancias que ocasionan pérdida permanente de información almacenada es menos frecuente en discos que en almacenamiento primario. Por lo tanto, nos referiremos al disco -y a otros dispositivos de almacenamiento secundario- como **almacenamiento no volátil**, mientras que a la memoria principal se le llama **almacenamiento volátil**.

- El costo de almacenamiento por unidad de datos es de magnitud menor en disco que en almacenamiento primario.

Están emergiendo nuevas tecnologías y hardware de propósito especial orientado a bases de datos, incluyendo el almacenamiento óptico, memorias más baratas y más grandes. Estas tecnologías pueden proporcionar alternativas viables al uso de discos magnéticos en el futuro. Por ahora, sin embargo, es importante estudiar y entender las propiedades y características de los discos magnéticos y la forma en que los archivos de datos se organizan en el disco para diseñar bases de datos efectivas con desempeño aceptable.

Las cintas mágnéticas se usan frecuentemente como medio de respaldo de las bases de datos debido a que una cinta cuesta menos que un disco. Sin embargo, el acceso a datos en cinta es lento. La información almacenada en cinta está **fuera de línea**; esto es, es necesaria la intervención de un operador (o dispositivo de carga) para cargar una cinta antes de que esta información esté disponible. En contraste, los discos son dispositivos **en línea** que pueden accesarse directamente en cualquier momento.

En este capítulo describiremos las técnicas usadas para almacenar grandes cantidades de información estructurada en disco. Estas técnicas son importantes para los diseñadores de bases de datos, el DBA e implementadores de DBMSs. Los diseñadores y el DBA deben conocer las ventajas y desventajas de cada técnica de almacenamiento cuando diseñan, implementan y operan una base de datos en un DBMS específico. Usualmente, el DBMS tendrá varias opciones disponibles para organizar la información, y el proceso del **diseño físico de la base** involucra elegir de entre las opciones disponibles las técnicas de orgnización que mejor se apeguen a los requerimientos de la aplicación. Los implementadores de sistemas DBMS deben estudiar las técnicas de organización de la información de tal forma que puedan implementarlas eficientemente y así proporcionar suficientes opciones al DBA y usuarios del DBMS.

Las aplicaciones típicas de bases de datos necesitan sólo una pequeña porción de la base al momento de su procesamiento. Cada vez que es necesaria una cierta porción de datos, debe de localizarse en el disco, copiarse a la memoria principal para procesarse, y luego reescribirla al disco si la información cambió. Los datos almacenados en disco se organizan como **archivos** o **registros**. Cada registro es una colección de valores de datos que pueden interpretarse como hechos acerca de entidades, sus atributos, y sus relaciones. Los registros deben almacenarse en disco de manera que sea posible localizarlos eficientemente.

Existen varias **organizaciones primarias de archivos** los cuales determinan cómo están *colocados físicamente en el disco* los registros de un archivo. Un *archivo pila* (o *archivo desordenado*) coloca los registros en disco sin un orden en particular, mientras que un *archivo sorteado* (o *archivo secuencial*) guarda los registros almacenados por el valor de un campo en particular. Un *archivo seccionado* utiliza una función de *seccionado* para determinar la colocación del registro en disco. Otras organizaciones primarias de archivos, tales como B-trees, usan estructuras de árbol.

4.2 Dispositivos de Almacenamiento Secundario

En esta sección describiremos algunas características de discos y cintas magnéticas.

4.2.1 Descripción del Hardware de Dispositivos de Disco

Los discos magnéticos se utilizan para almacenar grandes cantidades de información. La unidad básica de información en el disco es el **bit**. Magnetizando de cierta forma un área del disco, se puede hacer presente un valor de 0 o 1. Para codificar información, los bits se

agrupan en **bytes** (o **caracteres**). Los tamaños del byte son de 4 a 8 bits, dependiendo de la computadora y el dispositivo. Se asume que un caracter se almacena en un byte, y usamos los términos *byte* y *caracter* de forma intercambiable. La **capacidad** de un disco es el número de bytes que puede almacenar, la cual es usualmente muy grande. Nos referimos a las capacidades en disco en kilobytes (Kbyte), megabytes (Mbyte), y gigabytes (Gbyte). Los pequeños floppies usados con PC típicamente almacenan de 400 Kbytes a 1.2 Mbytes; los discos duros de micros típicamente almacenan de 30 a 250 Mbytes; y los grandes discos usados en microcomputadoras y mainframe tienen capacidades que llegan a unos cuantos Gbytes. Conforme mejora la tecnología las capacidades en disco continúan creciendo.

Cualquiera que sea la capacidad, todos los discos están hechos de material magnético formando un delgado disco circular (Figura 4.1(a)) y protegido por una funda plástica o acrílica. Un disco es de **un lado** si almacena información en una sola de sus superficies y de **dos lados** si se usan ambas. Para incrementar la capacidad de almacenamiento, los discos se ensamblan en **paquetes** (Figura 4.1(b)), los cuales pueden incluir tantas como 30 superficies. La información se almacena en la superficie del disco en círculos concéntricos de **pequeña amplitud**, cada uno de distinto diámetro. A cada círculo se le llama **pista**. En los paquetes de discos, a las pistas con el mismo diámetro en las distintas superficies se les llama **cilindro** debido a la forma que generarían si se conectaran en el espacio. El concepto de cilindro es importante porque la información almacenada en el mismo cilindro puede recuperarse mucho más rápido que si estuviera distribuída entre los distintos cilindros.

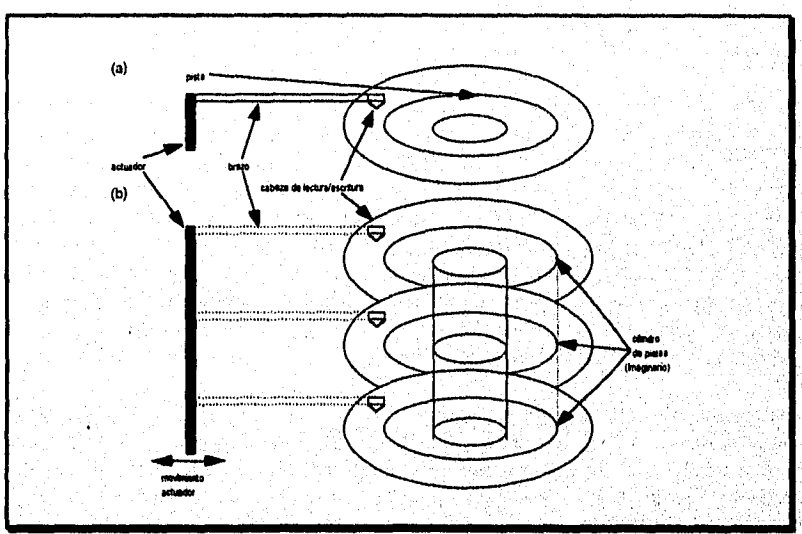


Figura 4.1 (a) Disco de un sólo lado con hardware de lectura/escritura. (b) Paquete de discos con hardware de lectura/escritura.

Cada círculo concéntrico almacena casi siempre la misma cantidad de información, así los bits se empacan de manera más densa en las pistas de menor diámetro. El número de pistas en un disco llega hasta las 800, y la capacidad de cada pista va de los 4 a los 50 Kbytes. Debido a que usualmente una pista contiene gran cantidad de información, se divide en bloques más pequeños o sectores. La división de una pista en **sectores** está codificada en el hardware del disco y no puede cambiarse. Los sectores subtienden un ángulo fijo en el centro (Figura 4.2), y no todos los discos tienen sus pistas divididas en sectores. La división de una pista en **bloques** o *páginas* ecualizadas se coloca por el sistema operativo durante el **formateo** del disco (o **inicialización**). El tamaño del bloque es fijo y no puede cambiarse dinámicamente. Los tamaños típicos de bloques van de 512 a 4096 bytes. Un disco con sectores codificados por hardware a menudo tiene los sectores más ampliamente subdivididos en bloques durante la inicialización. Los bloques se separan mediante **huecos** de tamaño fijo, los cuales incluyen información de control especialmente codificada escrita durante la inicialización. Esto se usa para determinar qué bloque de la pista sigue a cada bloque.

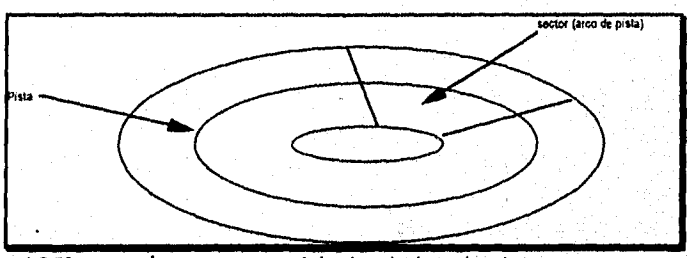


Figura 4.2 Un grupo de sectores que subtienden el mismo ángulo.

A un disco se le llama dispositivo direccionable de *acceso aleatorio*. La transferencia de datos entre memoria principal y disco tiene lugar en unidades de bloques. La **dirección de hardware** de un bloque -una combinación de número de superficie, número de pista (dentro de la superficie), y número de bloque (dentro de la pista)- se proporciona al hardware de I/O del disco, así como la dirección de un **buffer** -un área reservada contigua en almacenamiento principal que guarda un bloque. Para un comando **read**, se copia el bloque de disco a buffer; mientras que para un comando **write**, el contenido del buffer se copia al bloque del disco. Algunas veces varios bloques contiguos, llamados **clusters**, pueden transferirse como una unidad. En este caso el tamaño del buffer se ajusta para que corresponda con el número de bytes en el cluster.

El mecanismo de hardware que lee o escribe un bloque en el disco es la **cabeza de lectura/escritura**, la cual es parte del sistema llamado **drive**. Un disco o paquete de discos se monta en el drive, el cual incluye un motor que los hace girar. Una cabeza de lectura/escritura que incluye un componente electrónico conectado al **brazo mecánico**. Los paquetes de discos con varias superficies se controlan mediante varias cabezas de lectura/escritura -una para cada superficie (véase Figura 4.1(b)). Todos los brazos se conectan a un **actuador** conectado a otro motor eléctrico, el cual mueve las cabezas de

lectura/escritura al unísono y las posiciona precisamente sobre los cilindros de pistas especificados en un bloque de direcciones.

Los drives para disco duro giran el paquete continuamente a velocidad constante. Para un floppy, el drive empieza a girar al disco cada vez que se requiere de una escritura o lectura y la rotación cesa una vez que la transferencia es completada. Una vez que la cabeza de lectura/escritura se posiciona en la pista correcta y el bloque especificado en el bloque de direcciones, se activa el componente electrónico de la cabeza para transferir los datos. Algunas unidades de disco tienen cabezas fijas, con tantas cabezas como pistas. A estos se les llaman discos de **cabeza fija**, aunque a las unidades de disco con actuador se les llaman discos de **cabeza movible**. Para discos de cabeza fija, en vez de movimiento mecánico se selecciona electrónicamente una pista o cilindro switcheando a la cabeza apropiada; consecuentemente, es mucho más sencillo. Sin embargo, el costo de cabezas adicionales es alto, así los discos fijos no son muy utilizados.

Para transferir un bloque de disco, dada su dirección, el drive debe posicionar mecánicamente a la cabeza en la pista correcta. El tiempo requerido para lograrlo es el **tiempo de búsqueda**. Después de eso, hay otro retardo -llamado el **retardo rotacional** o **tardanza**- mientras el inicio del bloque deseado gira en posición bajo la cabeza. Finalmente, es necesario algún tiempo adicional para transferir los datos; a esto se le llama **tiempo de transferencia de bloque**. Por lo tanto, el tiempo total necesario para localizar y transferir un bloque arbitrario, dada su dirección, es la suma del tiempo de búsqueda, el retraso rotacional, y el de transferencia de bloque. El tiempo de búsqueda y el retraso rotacional son mucho más largos que el de transferencia de bloque. Para hacer más eficiente la transferencia de bloques, es común transferir varios bloques consecutivos en la misma pista o cilindro. Esto elimina el tiempo de búsqueda y el retraso rotacional para todo el primer bloque y puede resultar en un ahorro substancial de tiempo cuando se transfieren numerosos bloques contiguos. Usualmente, el fabricante de discos proporciona un **factor de transferencia de volúmen** para calcular el tiempo requerido para transferir bloques consecutivos.

El tiempo necesario para localizar y transferir un bloque de disco está en el orden de los milisegundos, usualmente de 15 a 60 mseg, pero la transferencia de bloques subsecuentes puede sólo tomar 1 o 2 mseg. Muchas técnicas de búsqueda sacan ventaja de los bloques de recuperación consecutivos. En cualquier caso, un tiempo de transferencia en el orden de los milisegundos se considera un poco alto comparado al mismo proceso requerido en memoria principal por los CPU's actuales. Por lo tanto, en aplicaciones de bases de datos la localización en disco es un *cuello de botella*.

4.2.2 Dispositivos de Almacenamiento de Cinta Magnética

Los discos son dispositivos de almacenamiento secundario de *acceso aleatorio*, debido a que una vez que se especifica su dirección puede accesarse un bloque arbitrario "aleatorio". Las cintas magnéticas son dispositivos de **acceso secuencial**; para accesar el enésimo bloque en la cinta, debemos leer primero los n-1 bloques precedentes. La

información se almacena en carretes de cinta magnética de alta capacidad, algo similar a las cintas de audio y video. Se requiere de un **drive de cinta** para leer información de o escribir a una **cinta**. Usualmente, cada grupo de bits que forman un byte se almacenan a lo ancho de la cinta, y los bytes se almacenan consecutivamente en la cinta.

Se usa una cabeza de lectura/escritura para leer o escribir datos a la cinta. Los registros también se almacenan en bloques -a pesar de que los bloques pueden ser substancialmente más grandes que los de los discos, y los huecos intermedios son también algo grandes. Con densidades de cinta típicas de 1600 a 6250 bytes por pulgada, un hueco de 0.6 pulgadas corresponde de 960 a 3750 bytes de espacio desperdiciado. Para mejor utilización del espacio es habitual agrupar muchos registros en un bloque.

La principal característica de una cinta es su requerimiento de accesar los datos en **orden secuencial**. Para llegar a un bloque a la mitad de la cinta, se debe montar la cinta y luego leerla hasta llegar al bloque requerido bajo la cabeza de lectura/escritura. Por esta razón, el acceso a cinta es lento y no se usan para almacenar información en línea, excepto por algunas aplicaciones especializadas. Sin embargo, las cintas sirven a una función muy importante -**respaldar** a la base de datos. Una razón de respaldar es guardar copias de seguridad de archivos en disco en caso de que la información se pierda debido a falla del disco, lo cual puede suceder si la cabeza toca a la superficie debido a una falla en el mecanismo. Por esta razón, los archivos en disco son periódicamente copiados a cinta. Las cintas también se usan para almacenar archivos excesivamente grandes. Finalmente, los archivos de bases de datos que raramente se usan pero se requieren para históricos se guardan en cintas **archivadas**. Recientemente, cintas magnéticas más pequeñas que las de 8 mm (similares a las usadas en cámaras) y CD ROM's, se están volviendo medios populares para respaldar archivos de datos de estaciones de trabajo y computadoras personales y almacenar imágenes y sistemas de bibliotecas.

4.3 Buffereo de Bloques

Cuando se necesita transferir varios bloques de disco a memoria principal y se conocen de antemano todas las direcciones, pueden reservarse varios buffers en memoria principal para acelerar la transferencia. Mientras un buffer está siendo leído o escrito, el CPU puede procesar datos de otro buffer. Esto es posible debido a que usualmente existe un procesador independiente de I/O, puede proceder a transferir un bloque de datos entre memoria y disco, independientemente y en paralelo al CPU.

La Figura 4.3 ilustra cómo puede efectuarse el proceso en paralelo. Los procesos A y B están corriendo **concurrentemente** de forma **intercalada**, mientras que los procesos C y D están corriendo **concurrentemente** de manera **simultánea**. Cuando un sólo CPU controla múltiples procesos, no es posible la ejecución simultánea. Sin embargo, el proceso puede seguir corriendo concurrentemente de manera intercalada. El buffereo es más útil cuando los procesos pueden correr concurrentemente de manera simultánea, ya sea a través de un procesador de I/O independiente, o a través de varios procesadores.

La Figura 4.4 ilustra cómo puede efectuarse la lectura y el procesamiento en paralelo cuando el tiempo requerido para procesar un bloque de disco en memoria es menor al tiempo requerido para leer el siguiente bloque y llenar un buffer. El CPU puede empezar a procesar un bloque una vez que se completa su transferencia a memoria principal; al mismo tiempo el procesador de I/O de disco puede leer y transferir el siguiente bloque a un buffer diferente. A esta técnica se le llama **doble buffereo** y puede utilizarse para escribir un flujo continuo de bloques de memoria a disco. El doble buffereo permite lectura continua o escritura de datos en bloques consecutivos del disco, lo cual elimina el tiempo de búsqueda y el retraso rotacional para todo el primer bloque de transferencia. Sin embargo, los datos se mantienen listos para procesarse, reduciendo así el tiempo de espera en los programas.

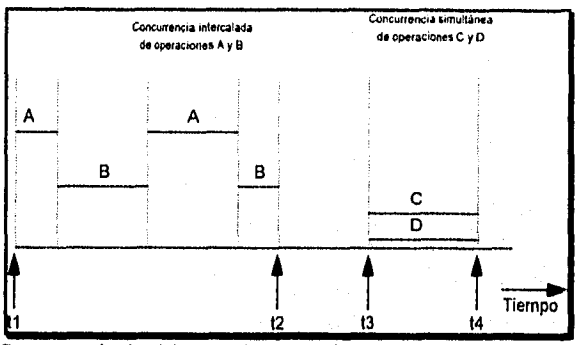


Figura 4.3 Concurrencia simultánea contra intercalada.

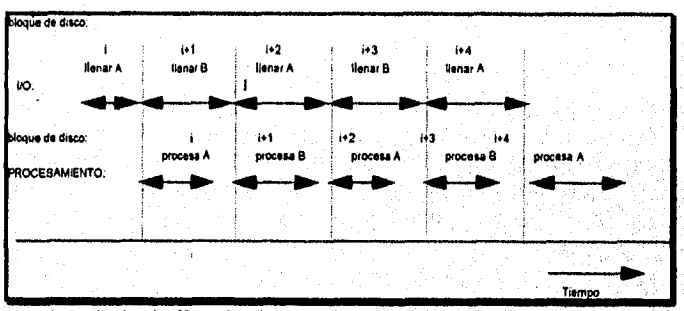


Figura 4.4 Uso de dos buffers, A y B, para lectura en disco.

4.4 Colocación de Archivos de Registros en Disco

En esta sección definiremos los conceptos de registros, tipos de registros, y archivos. Después discutiremos las diferentes técnicas de colocación de archivos de registros en disco.

4.4.1 Tipos de Registros

Los datos se almacenan usualmente en forma de **registros**. Cada registro consta de una colección de **valores** de información relacionada, donde cada valor se forma de uno o más bytes y corresponde a un **campo** particular del registro. Los registros usualmente describen entidades y sus atributos. Por ejemplo, un registro EMPLEADO representa la entidad de un empleado, y cada valor de campo en el registro especifica algún atributo de ese empleado, tal como el NOMBRE, FECHANAC, SALARIO, o SUPERVISOR. Una colección de nombres de campos y su correspondiente tipo de información constituye una definición de **tipo registro** o **formato de registro**. Un **tipo de dato**, asociado con cada campo, especifica el tipo de valores que un campo puede tomar.

El tipo de dato de un campo es usualmente uno de los tipos de datos estándar usados en programación. Estos incluyen numéricos (integer, long-integer, o real), cadena de caracteres (de longitud fija o variable), Booleano (sólo valores de 0 o 1 o FALSO y VERDADERO), y algunas veces tipos de datos especialmente codificados **tiempo** y **fecha**. Para un sistema de cómputo dado el número de bytes requeridos para cada tipo de dato es fijo. Un entero puede requerir 4 bytes, un long integer 8, un real 4 bytes, un Boolean 1 byte, una fecha 4 bytes (para codificar la fecha a entero), y una cadena de longitud fija de k caracteres k bytes. Las cadenas de longitud variable pueden requerir tantos bytes como caracteres en cada valor de campo. Por ejemplo, un registro tipo EMPLEADO puede definirse -usando notación PASCAL- como sigue:

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

En recientes aplicaciones de bases de datos, puede surgir la necesidad de almacenar elementos de datos que consistan de grandes objetos no estructurados, que representen imágenes, video digitalizado o flujo de video, o texto libre. Estos son referidos como **BLOBs** (objetos binarios largos). Normamente, un elemento de datos BLOB se almacena separadamente de su registro en una alberca de bloques de disco, e incluye en el registro un apuntador al BLOB.

4.4.2 Archivos, Registros de Longitud Fija y Variable

Un **archivo** es una secuencia de registros. En muchos casos, todos los registros de un archivo son del mismo tipo. Si cada registro del archivo tiene exactamente el mismo tamaño (en bytes), el archivo se dice estar compuesto de registros de **longitud fija**. Si los diferentes registros del archivo tienen diferentes tamaños, el archivo se dice estar formado por registros de **longitud variable**. Un archivo puede tener registros de longitud variable por varias razones:

- Los registros son del mismo tipo, pero uno o más de los campos pueden tener distintos valores para registros individuales; a tal campo se le llama **campo de repetición** y al grupo de valores del campo se le llama **grupo de repetición.**

- Los registros son del mismo tipo, pero uno o más de los campos son **opcionales**; esto es, pueden tener los valores de algunos pero no de todos los registros (**campos opcionales**).

- El archivo contiene registros de *diferentes tipos* y por lo tanto de tamaño variable (**archivo mixto**). Esto ocurriría si registros relacionados de diferentes tipos fueran colocados juntos en bloques de disco; por ejemplo, los registros de REP_CAL de un estudiante en particular pueden colocarse después del registro ESTUDIANTE.

Los registros de longitud fija EMPLEADO en la Figura 4.5(a) tienen un tamaño de 71 bytes. Cada registro tiene los mismos campos, y las longitudes son fijas, de tal forma que se puede identificar la posición inicial de cada campo relativo a la posición inicial del registro. Esto facilita el localizar valores de campos por programas que accesan a tales archivos. Notemos que es posible representar un archivo que lógicamente tiene registros de longitud variable como un archivo de registros de longitud fija. Por ejemplo, en el caso de campos opcionales podríamos tener *cada campo* incluido en *cada registro del archivo* pero almacenando un valor especial nulo si no hay valor para el campo. Para un campo repetido, podríamos colocar tantos espacios en cada registro como el valor máximo que el campo puede tomar. En cualquier caso, se desperdicia espacio cuando ciertos registros no tienen valores para todos los espacios físicos proporcionados en cada registro. Ahora consideremos otras opciones para formatear registros de longitud variable.

Para *campos de longitud variable*, cada registro tiene un valor de cada campo, pero no se conoce la longitud exacta de algunos valores. Para determinar los bytes dentro de un registro particular que representa cada campo, se pueden usar caracteres especiales **separadores** -los cuales no aparecen en ningún valor de campo- (tal como ? o % o $)- para determinar los campos de longitud variable (Figura 4.5(b)), o se puede almacenar la longitud de cada campo en el registro.

Un archivo de registros con *campos opcionales* pude formatearse de diferentes maneras. Si el número total de campos para el tipo de registro es grande pero el número de campos que actualmente aparecen en un registro típico es pequeño, podemos incluir en cada registro una secuencia de parejas <nombre del campo, valor del campo> en lugar de sólo los valores de los campos. En la Figura 4.5(c) se usan tres tipos de caracteres separadores, a pesar de que se podría usar el mismo caracter para los primeros dos propósitos - separando el nombre del valor del campo y separando a un campo del siguiente. Una opción más práctica es asignar un **tipo de campo** short -digamos, un integer- para cada campo e incluir en cada registro una secuencia de parejas <tipo de campo, valor del campo> en lugatr de <nombre del campo, valor del campo>.

Un *campo repetible* necesita de un separador para separar los valores repetibles del campo y otro separador para indicar la terminación del campo. Fianlmente, para un archivo que incluye *registros de diferentes tipos*, cada registro se procesa por un indicador de tipo de **registro**. Por lógica, los programas que procesan archivos con registros de longitud variable necesitan ser más complejos que aquéllos de longitud fija, donde la posición inicial y el tamaño de cada campo se conocen y son fijos.

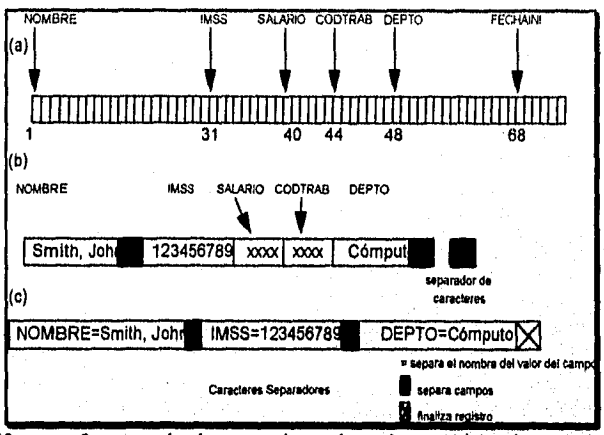


Fig 4.5 Diferentes formatos de almacenamiento de registros. (a) Registro de longitud fija con seis campos, y 71 bytes. (b) Registro con dos campos de longitud variable y tres campos de longitud fija. (c) Registro de longitud variable con tres tipos de separadores de caracteres.

4.4.3 Bloqueo y Puenteo de Registros contra No Puenteados

Los registros de un archivo deben localizarse en bloques de disco porque son la *unidad de transferencia de información* entre disco y memoria. Cuando el tamaño del bloque es mayor que el del registro, cada bloque contendrá numerosos registros. Algunos archivos pueden tener inusualmente registros de gran tamaño que no caben dentro de un bloque. Supongamos que el tamaño del bloque es de B bytes. Para un archivo de registros de longitud fijo de tamaño R bytes, con B >= R, pueden caber (B/R) registros por bloque, donde la (*función inferior*) (x) redondea el valor x al siguiente entero inferior. Al valor *bfr* se le llama **factor de bloqueo** del archivo. En general, R no puede dividir exactamente a B, así podemos tener espacio no utilizado en cada bloque igual a

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

Para usar este espacio no utilizado, podemos almacenar parte del registro en un bloque y el resto en otro. Un **apuntador** al final del primer bloque apunta al bloque contenedor del resto del registro en caso de no ser el siguiente bloque consecutivo en disco. A esta organización se le llama **puenteo**, debido a que los registros pueden puentearse en más de

un bloque. Cada vez que un registro es más grande que un bloque, *debemos* utilizar organización puenteada. La organización se llama **no puenteada** si a los registros no se les permite cruzar los límites del bloque. Esto se usa con registros de longitud fija teniendo B ≥ R porque hace que cada registro empiece en una localidad conocida del bloque, simplificando el procesamiento del registro. Para registros de longitud variable, puede usarse organización puenteada o no puenteada. Si el tamaño promedio del registro es grande, es ventajoso usar el puenteo para reducir el espacio perdido en cada bloque. La Figura 4.6 ilustra la organización puenteada contra la no puenteada.

Para registros de longitud variable se usa la organización puenteada, cada bloque puede almacenar diferente número de registros. En este caso, el factor de bloqueo bfr representa el número de registros *promedio* por bloque para el archivo. Podemos usar bfr para calcular el número de bloques b necesarios para un archivo de r registros

$$b = (r/bfr) \text{ bloques}$$

donde la (*función superior*) (x) redondea el valor al siguiente entero.

4.4.4 Localización de Bloques de Archivos en Disco

Existen varias técnicas estándar para localizar los bloques de un archivo en disco. En **localización contigua** los bloques del archivo se colocan en bloques consecutivos del disco. Esto hace la lectura del archivo muy rápida usando doble buffereo, pero dificulta su expansión. En la **localización enlazada** cada bloque tiene un apuntador al siguiente bloque. Esto facilita la expansión del archivo pero hace lenta la lectura. A los clusters algunas veces se les llaman **segmentos** o **extensiones**. Otra posibilidad es usar la **localización indexada**, donde uno o más **bloques índice** contienen apuntadores a los bloques actuales de archivo. También es común usar combinaciones de estas técnicas.

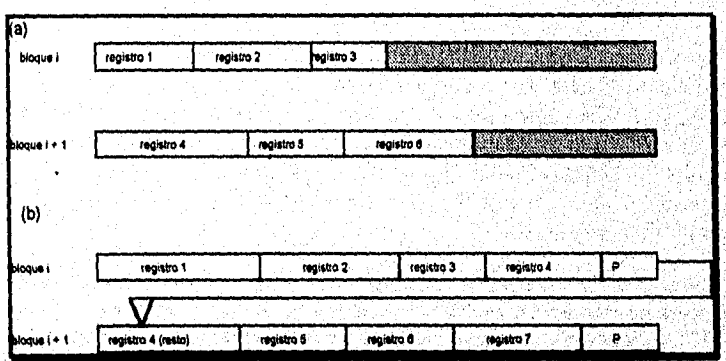


Figura 4.6 Tipos de organización de registros. (a) No puenteados. (b) Puenteados.

4.4.5 Encabezados de Archivos

Un **encabezado** o **descriptor** contiene información acerca de un archivo la cual es necesaria para los programas que accesan los archivos de registros. El encabezado incluye información para determinar las direcciones en disco de los bloques asi como las descripciones de los formatos de registros, los cuales pueden incluir la longitud de los campos y el orden de estos dentro del registro para los registros no puenteados de longitud fija y códigos del tipo de campo, caracteres separadores, y códigos de tipo de registro para registros de longitud variable.

Para buscar un registro en disco, se copian uno o más bloques a buffers de memoria principal. Entonces buscan el registro o registros deseados dentro de los buffers, utilizando la información en el encabezado. Si no se conoce la dirección del bloque que contiene el registro deseado, los programas de búsqueda deben hacer una **búsqueda lineal** a través de los bloques de archivos. Cada bloque se copia a un buffer y se busca hasta que se encuentre un registro o hayan sido buscados todos los bloques inexitosamente. Para un archivo grande esto puede ser muy consumidor de tiempo. La meta de una buena organización de archivo es localizar el bloque que contiene un registro deseado dentro de un mínimo de transferencias de bloques.

4.5 Operaciones Sobre Archivos

Usualmente se agrupan en operaciones de **recuperación** y **actualización**. La operación no cambia ningún dato en el archivo, sólo localiza ciertos registros de tal forma que puedan examinarse y procesarse sus valores de campos. El último cambia al archivo insertando o borrando registros o modificando valores de campos. En cualquier caso, podemos **seleccionar** uno o más registros para recuperación, borrado, o modificación basada en una **condición de selección**, la cual especifica los criterios que deben satisfacer el registro o registros deseados.

Consideremos un archivo EMPLEADO con campos NOMBRE, IMSS, SALARIO, CODTRAB y DEPTO. Una **condición sencilla de selección** puede involucrar una comparasión de igualdad en algún valor de campo -por ejemplo, (IMSS = '123456789') o (DEPTO = 'Investigación'). Condiciones más complejas pueden involucrar otros tipos de operadores de comparasión, tales como > o ≥; un ejemplo es (SALARIO ≥ 30000). El caso general es tener una expresión Booleana arbitraria sobre los campos del archivo como condición de selección.

Las operaciones de búsqueda en sistemas de archivos se basan generalmente en condiciones de selección sencilla. Una condición compleja debe descomponerse por el DBMS (o el programador) para extraer una condición sencilla que pueda usarse para localizar los registros en el disco. Cada registro localizado se checa para determinar si satisface la condición de selección completa. Por ejemplo, podemos extraer la condición sencilla (DEPTO = 'Investigación') de la condición compleja ((SALARIO ≥ 30000) AND

(DEPTO = 'Investigación')), se localiza a cada registro que satisface (DEPTO = 'Investigación') y luego se prueba para ver si también satisface (SALARIO ≥ 30000).

Cuando varios registros satisfacen la condición de búsqueda, sólo se localiza al *primer* registro -con respecto a la secuencia física de los registros. Para localizar los demás registros que satisfagan la condición, son necesarias operaciones adicionales. Al registro más recientemente localizado se le designa como el **registro actual**. Las operaciones subsecuentes de búsqueda comienzan a partir de este registro y localizan al *siguiente* que satisfaga la condición.

Las operaciones de localización y acceso varían de sistema a sistema. A continuación, presentamos un conjunto de operaciones representativas. Típicamente, programas de alto nivel, tales como los programas del DBMS, accesan los registros usando estos comandos, así algunas veces nos referimos a **variables de programa** con la siguiente descripción:

- *Find* (o Locate): Busca al primer registro que satisfaga la condición de búsqueda. Transfiere los bloques que contienen a ese registro a un buffer de memoria principal (si es que no está ahí). Se localiza al registro en el buffer y se vuelve el registro actual. Algunas veces, se usan diferentes verbos para indicar si el registro localizado va a ser recuperado. (Find) o (Locate).

- *Read* (o Get): Copia el registro actual del buffer a una variable de programa o área de trabajo en el programa del usuario. Este comando puede también avanzar el apuntador del registro actual al siguiente, lo cual puede necesitar de lectura del siguiente bloque de archivo del disco.

- *FindNext*: Busca al siguiente registro en el archivo que satisfaga la condición de búsqueda. Transfiere a los bloques que contienen a ese registro a un buffer de memoria principal (si es que no está ahí). Se localiza al registro en el buffer y se convierte en el registro actual.

- *Delete*: Borra al registro actual y (eventualmente) actualiza al archivo en disco para reflejar la modificación.

- *Modify*: Modifica algunos valores de campos del registro actual y (eventualmente) actualiza el archivo en disco para reflejar la modificación.

- *Insert*: Inserta un nuevo registro en el archivo localizando el bloque donde el registro va a ser insertado, transfiriendo ese bloque a un buffer de memoria principal (si no está ya ahí), escribe el registro al buffer, y (eventualmente) escribe el buffer a disco para reflejar la inserción.

A las operaciones anteriores se les llaman operaciones de **un registro a la vez,** porque cada operación se aplica a un sólo registro. En algunos sistemas, pueden aplicarse operaciones de más alto nivel. Ejemplos de ello son:

- *FindAll*: Localiza a todos los registros en el archivo que satisfacen la condición de búsqueda.
- *FindOrdered*: Recupera todos los registros en el archivo en algún orden especificado.
- *Reorganize*: Inicia el proceso de reorganización. Como veremos, algunas organizaciones de archivos requieren de reorganización periódica. Un ejemplo es reordenar los registros sorteandolos sobre un campo especificado.

Para preparar el acceso al archivo son necesarias otras operaciones (**Open**) y para indicar que hemos finalizado de utilizar el archivo (**Close**). La operación Open típicamente recupera al encabezado del archivo y prepara los buffers de memoria para subsecuente operación con el archivo.

En este punto, vale la pena hacer notar la diferencia entre los términos *organización de archivos* y *método de acceso*. Una **organización de archivos** se refiere a la organización de los datos en un archivo en registros, bloques, y estructuras de acceso; esto incluye la forma en que los los registros y bloques se colocan y entrelazan en el medio de almacenamiento. Un **método de acceso**, por otro lado, consta de un grupo de programas que permiten aplicar operaciones -tales como las listadas anteriormente- a un archivo. En general, es posible aplicar diferentes métodos de acceso a una organización de archivos. Algunos métodos de acceso, sin embargo, pueden aplicarse sólo a archivos organizados de cierta forma. Por ejemplo, no se puede aplicar un método de acceso indexado a un archivo sin índice.

Usualmente, esperamos usar algunas condiciones de búsqueda más que a otras. Algunos archivos pueden ser **estáticos**, significando que raramente se efectúan operaciones de actualización; otros, más **volátiles** pueden cambiar frecuentemente, aplicándose así constantemente condiciones de actualización. Una organización de archivos exitosa debe realizar tan eficientemente como sea posible las operaciones que esperamos *aplicar frecuentemente* a un archivo. Por ejemplo, consideremos el archivo EMPLEADO, el cual almacena los registros de los empleados actuales en una compañía. Esperamos insertar nuevos registros (cuando se contraten empleados), borrar registros (cuando los empleados dejan la compañía), y modificar registros (cuando se cambia el salario de un empleado). Borrar o modificar un registro requiere una condición de selección para identificar un registro en particular o conjunto de registros. Recuperar uno o más registros también requiere de una condición de selección.

Si los usuarios esperan principalmente aplicar una condición de búsqueda basada en IMSS, el diseñador debe elegir una organización de archivos que facilite la localización de un registro dado un valor de IMSS. Esto puede involucrar el ordenar fisicamente los registros por valor del IMSS o definir un índice en IMSS. Supongamos que una segunda aplicación usa el archivo para generar la nómina y requiere que los cheques se agrupen por departamento. Para esta aplicación es mejor almacenar todos los registros de empleados que tienen el mismo valor del departamento contiguo, empacándolos en bloques y quizá

ordenándolos por nombre dentro de cada departamento. Sin embargo, este arreglo entra en conflicto con el ordenamiento de registros por valor de IMSS. Si es posible, el diseñador debe elegir una organización que permita que ambas operaciones se lleven a cabo de manera eficiente, pero al mismo tiempo la actualización se vuelve costosa. En tales casos debe elegirse un compromiso que tome en cuenta la mezcla esperada de recuperar y actualizar.

Para crear métodos de acceso se usan varias técnicas generales, tales como el ordenamiento, seccionado, e indexado. Además, varias técnicas generales para manejar inserciones y borrado trabajan con muchas organizaciones de archivos.

4.6 Archivos de Registros Desordenados (Archivos Apilados)

En el más simple y básico método de organización, los registros se colocan en el archivo en el orden en el cual se insertan, y los registros nuevos se insertan al final del archivo. A tal organización se le llama **cola**. Esta organización se usa con frecuencia con trayectorias de acceso adicionales, tales como los índices secundarios. También se usa para colectar y almacenar registros para uso futuro.

Insertar un nuevo registro es *muy eficiente*: el último bloque del archivo se copia al buffer; se agrega el nuevo registro; y se **reescribe** el bloque disco. La dirección del último bloque se guarda en el encabezado del archivo. Sin embargo, la localización de un registro usando cualquier condición de búsqueda involucra una **búsqueda lineal** a través del archivo por bloque -un procedimiento caro. Si sólo un registro satisface la condición de búsqueda, entonces, en promedio, un programa leerá la memoria y buscará la mitad de los bloques antes de encontrar el registro. Para un archivo de b bloques, esto requiere una búsqueda de (b/2) bloques, en promedio. Si ningún registro satisface la condición de búsqueda, el programa deberá leer y buscar todos los bloques del archivo.

Para borrar un registro, el programa debe encontrarlo, copiar el bloque al buffer, luego borrar el registro del buffer, y finalmente **reescribir el bloque** al disco. Esto deja espacio extra inutilizado en el disco. Borrar un gran número de registros de esta forma resulta en espacio desperdiciado. Otra técnica utilizada para el borrado de registros es tener un byte o bit extra, llamado **marcador de borrado**, almacenado en cada registro. Un registro se borra colocando el marcador de borrado en cierto valor. Un valor diferente del marcador indica un registro válido (no borrado). Los programas de búsqueda consideran sólo a los registros válidos en un bloque. Ambas técnicas de borrado requieren **reorganización** periódica del archivo para recuperar el espacio no usado de registros borrados. Después de tal reorganización, los bloques se llenan a su capacidad una vez más. Otra posibilidad es usar el espacio de los registros borrados cuando se insertan nuevos registros, a pesar de que esto requiere registro extra para seguir la pista de las localidades vacías.

Para un archivo desordenado podemos usar organización puenteada o no puenteada y registros de longitud fija y variable. Modificar un registro de longitud variable puede

requerir borrar el registro anterior e insertar un registro modificado, debido a que el registro modificado puede no caber en el espacio anterior en el disco.

Para leer todos los registros en orden de los valores de algun campo, creamos una copia sorteada del archivo. El sorteo es una operación costosa para un archivo grande en disco, y se usan técnicas especiales de **sorteo externo**. Un método común es una variación de técnica de sorteo y carga. Primero, se sortean los registros dentro de cada bloque. Luego se cargan los bloques para crear grupos de registros sorteados, cada uno del tamaño de dos bloques. A cada uno de los grupos de registros se le llama **corrida**. Después se cargan corridas de dos bloques para formar corridas de cuatro, y así, hasta que la corrida final es el archivo completamente sorteado.

Para un archivo de *registros de longitud fija* desordenado usando *bloques no puenteados* y *localización contigua*, es directo el acceso a cualquier registro por su **posición** dentro del archivo. Si los registros están numerados 0, 1, 2...r-1 y los registros en cada bloque están numerados 0, 1...bfr - 1, donde bfr es el factor de bloqueo, entonces el iésimo registro del archivo se localiza en el bloque (i/bfr) y es el registro (i mod bfr) en ese bloque. A tal archivo se le llama **archivo relativo** porque los registros pueden accesarse fácilmente por su posición relativa. Accesar un registro por su posición no ayuda a localizarlo basado en una condición de búsqueda; sin embargo, facilita la construcción de trayectorias de acceso al archivo.

4.7 Archivos de Registros Ordenados (Archivos Sorteados)

Los registros de un archivo en disco se pueden ordenar físicamente basado en los valores de uno de sus campos -llamado el **campo de ordenamiento**. Esto conduce a un archivo **ordenado** o **secuencial**. Si el campo de ordenamiento es también el **campo llave** del archivo -un campo que garantiza tener un valor único en cada registro- entonces al campo también se le llama **llave de ordenamiento**. La Figura 4.7 muestra un archivo ordenado con NOMBRE como el campo llave de ordenamiento (asumiendo que los empleados tienen nombres distintos).

Los registros ordenados tienen algunas ventajas sobre los archivos desordenados. Primero, la lectura ordenada se vuelve extremadamente eficiente, debido a que no se requiere de sorteo. Segundo, encontrar al siguiente registro del actual no requiere de acceso adicional de bloques, porque el siguiente registro está en el mismo bloque que el actual (a menos que el registro actual sea el último del bloque). Tercero, usando una condición de búsqueda basada en el valor de una llave de ordenamiento da como resultado un acceso más rápido cuando se usa la técnica de búsqueda binaria, la cual constituye una mejora sobre la búsqueda lineal a pesar de que no es usada con frecuencia para archivos en disco.

Puede hacerse una **búsqueda binaria** para archivos en disco en los bloques en vez de los registros. Supongamos que el archivo tiene b bloques numerados 1, 2..., b; los registros se ordenan por valor ascendente de su campo llave; y estamos buscando un registro cuyo valor de campo es k. Asumiendo que las direcciones de disco de los bloques de archivo

están disponibles en el encabezado, la búsqueda binaria puede describirse por el algoritmo 4.1 Una búsqueda binaria usualmente accesa log2(b) bloques, ya sea que se encuentre o no el registro, -una mejora sobre la búsqueda lineal, donde, en promedio, se accesan (b/2) bloques cuando se encuentra el registro y b bloques cuando no.

ALGORITMO 4.1 Búsqueda binaria sobre una llave de ordenamiento de un archivo en disco.

```
l←1; u←b; (b es el número de bloques)
while(u >= l) do
  begin i ←(l + u) div 2;
        read block i del archivo en buffer;
        if k < valor del campo de ordenamiento del primer registro en el bloque
              then u ← i -1
              else if k > valor del campo de ordenamiento del último registro en el bloque
                    then l ← i + 1
                    else if registro con valor del campo de ordenamiento = k está en el buffer
                          then goto found
                          else goto notfound
  end;
goto notfound;
```

Un criterio de búsqueda que involucra las condiciones >, <, ≥ y ≤ en el campo de ordenamiento es poco eficiente, debido a que el ordenamiento físico de los registros significa que todos los registros que satisfacen la condición están contiguos en el archivo. Por ejemplo, refiriéndonos a la Figura 4.7, si el criterio de búsqueda es (NOMBRE < 'F') - donde < significa "alfabéticamente antes" -los regitros que satisfacen el criterio de búsqueda son aquéllos del inicio del archivo hasta el primer registro que tenga un valor de NOMBRE con la letra F.

El ordenamiento no proporciona ninguna ventaja del acceso aleatorio u ordenado de los registros basados en los valores de un *campo no ordenado* del archivo. En estos casos hacemos una búsqueda lineal para acceso aleatorio. Para accesar los registros ordenados basados en un campo no ordenado, es necesario crear otra copia sorteada del archivo.

Insertar y borrar registros son operaciones costosas en un archivo ordenado porque corrigen la posición dentro del archivo, basado en el valor de su campo de ordenamiento, y luego hacen espacio para insertar el registro en esa posición. Para un archivo grande esto puede ser consumidor de tiempo porque, en promedio, debe moverse la mitad de los registros del archivo para hacer espacio al nuevo registro. Esto significa que la mitad de los bloques del archivo deben leerse y escribirse después de que los registros se han movido entre ellos. Para el borrado de registros el problema es menos severo, si usamos marcadores de borrado y se reorganiza el archivo periódicamente.

bloque 1

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Aaron, Ed					
Abbott, Diane					
Acosta, Marc					

bloque 2

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Adams, John					
Adams, Robin					
Akers, Jan					

bloque 3

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Alexander, Ed					
Alfred, Bob					
Allen, Sam					

bloque 4

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Allen, Troy					
Anders, Keith					
Anderson, Rob					

bloque 5

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Anderson, Zach					
Angeli, Joe					
Archer, Sue					

bloque 6

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Arnold, Mack					
Arnold, Steven					
Atkins, Tim					

bloque n-1

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Wong, James					
Wood, Donald					
Woods, Manny					

bloque n

NOMBRE	IMSS	FECHA_NAC	PUESTO	SALARIO	SEXO
Wright, Pam					
Wyatt, Charl					
Zimmer, Byron					

Figura 4.7 Algunos bloques de un archivo de registros **EMPLEADO** ordenado por **NOMBRE** como el campo de ordenamiento.

Una opciòn para hacer la inserción más eficiente es reservar algun espacio no usado en cada bloque para nuevos registros. Sin embargo, una vez usado este espacio, el problema original re-emerge. Otro método frecuentemente empleado es crear un *archivo temporal desordenado* llamado archivo de **sobreflujo** o **transacción**. Con esta técnica, al archivo ordenado se le llama **principal** o **maestro**. Los registros nuevos se insertan al final del archivo de sobreflujo en vez de en su posición correcta en el archivo principal. Periódicamente, se carga al archivo de sobreflujo con el maestro durante la

reorganización. La inserción se vuelve muy eficiente, pero a costa de incrementar la complejidad en el algoritmo de búsqueda. Si después de la búsqueda binaria, el registro no se encuentra en el archivo principal, debe buscarse en el archivo de sobreflujo usando una búsqueda lineal. Para aplicaciones que no requieren información muy actualizada, los registros de sobreflujo se ignoran durante la búsqueda.

Modificar el valor del campo de un registro depende de dos factores: la condición de búsqueda para localizar el registro, y el campo a modificar. Si la condición de búsqueda involucra el ordenamiento de un campo llave, podemos localizar el registro usando búsqueda binaria; de otro modo debemos hacer una búsqueda lineal. Un campo no ordenado puede modificarse cambiando al registro y reescribiéndolo en la misma localidad física del disco -suponiendo registros de longitud fija. Modificar el campo de ordenamiento significa que el registro puede cambiar su posición dentro del archivo, lo cual requiere el borrado del registro anterior seguido de la inserción del registro modificado.

La lectura de registros en el orden del campo ordenador es poco eficiente si ignoramos los registros en sobreflujo, debido a que los bloques pueden leerse consecutivamente usando doble buffereo. Para incluir los registros en sobreflujo, debemos cargarlos en sus posiciones correctas; en esta caso, podemos reorganizar primero al archivo, y luego leer sus bloques secuencialmente. Para reorganizar el archivo, debemos sortear los registros en sobreflujo, y luego cargarlos con el archivo maestro. Durante la reorganización se eliminan los registros marcados como borrados.

Ráramente se usan los archivos ordenados en aplicaciones de bases de datos a menos que se incluya con el archivo una trayectoria de acceso adicional, llamada *índice primario*. Esto mejora el tiempo de acceso aleatorio en el campo llave de ordenamiento.

4.8 Técnicas de Seccionamiento

Otro tipo de organización primaria de archivos se basa en el seccionado, el cual bajo ciertas condiciones de búsqueda proporciona acceso muy rápido a registros. A esta organización usualmente se le llama archivo **seccionado** o **directo**. La condición de búsqueda debe ser una igualdad sobre un solo campo, llamado el **campo seccionado** del archivo. A menudo, el campo seccionado es la llave del archivo, en cuyo caso se le llama **llave seccionada**. La idea detrás del seccionamiento es proporcionar una función *h*, llamada **función de seccionamiento** o **función aleatoria**, que se aplica al valor del campo seccionado de un registro y produce la *dirección* del bloque en la cual se almacena el registro. Una búsqueda de registro dentro del bloque puede llevarse a cabo en un buffer de memoria principal. Para la mayoría de los registros, necesitamos un solo bloque de acceso sencillo para ser recuperado.

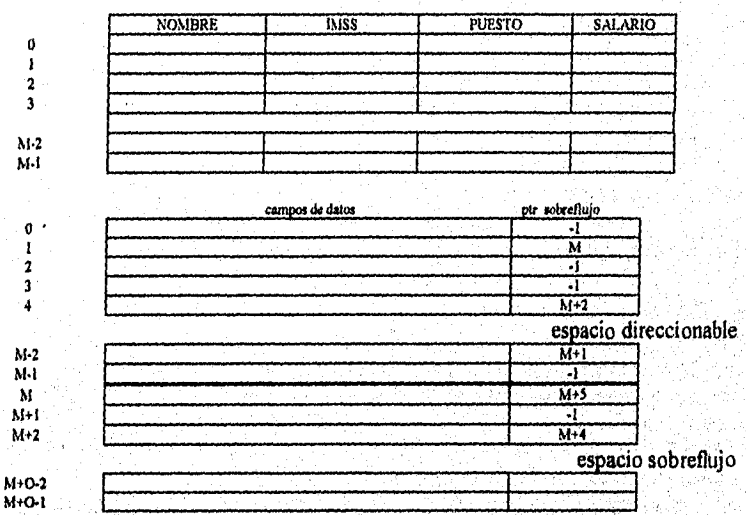
El seccionado se usa también como una estructura de datos interna dentro de un programa cada vez que un pequeño archivo temporal de registros se accesa exclusivamente usando el valor del campo.

4.8.1 Seccionamiento Interno

Para archivos internos, el seccionado se implementa típicamente a través del uso de arreglos de registros. Supongamos que el índice del arreglo va de 1 a M-1 (Figura 4.8(a)); entonces tendremos M **ranuras** cuya dirección corresponde a los índices de los arreglos. Seleccionamos una función de seccionado que transforma el valor del campo seccionado a un entero entre 0 y M - 1. Una función común de seccionado es h(K) = K **mod** M, la cual retorna el residuo de un campo valor K después de su división por M; este valor entonces se usa para la dirección del registro.

Los valores no enteros de seccionado pueden transformarse a enteros antes de aplicar la función mod. Para cadenas de caracteres, los códigos numéricos asociados con caracteres pueden usarse en la transformación -por ejemplo, multiplicando esos valores. Para un campo seccionado cuyo tipo de dato es una cadena de 20 caracteres, puede usarse el Algoritmo 4.2(a). Suponemos que la función código retorna el código numérico de un caracter y que se da un valor de campo seccionado K de tipo arreglo[1..20] de char.

(a)



- apuntador nulo = -1
- apuntador de sobreflujo se refiere a la posición del siguiente registro en la lista enlazada.

Figura 4.8 Estructuras de datos con seccionamiento interno. (a) Arreglo de M posiciones para usarse en seccionamiento interno. (b) Resolución de colisión encadenando registros.

ALGORITMO 4.2 Dos algoritmos sencillos de seccionamiento. (a) Aplicando la función de seccionamiento mod a una cadena de caracteres. (b) Resolución de colisión por direccionamiento abierto.

(a)
```
temp ← 1;
for i ← 1 to 20 do temp ← temp * code(K[i]);
        hash_address ← temp mod M;
```

(b)
```
i ← hash address;
if localidad i es ocupada
  then begin i ← (i + 1) mod M;
  while(i <> hash_address) y localidad i está ocupada
        do i ← (i + 1) mod M;
  if(i=hash_address) then todas las posiciones llenas
        else new_hash_address ← i;
  end;
```

Pueden utilizarse otras funciones de seccionamiento. Una técnica, llamada **folding**, involucra aplicar una función aritmética tal como la suma o una función lógica tal como "or exclusiva" a diferentes porciones del valor del campo seccionado para calcular la dirección. Otra técnica involucra tomar algunos dígitos del valor del campo seccionado -por ejemplo el tercero, quinto y octavo dígitos- para formar la dirección de seccionamiento. El problema con la mayoría de estas funciones es que no garantizan que distintos valores seccionarán a distintas direcciones, porque el **espacio de campo seccionado** -el número de posibles valores que un campo seccionado puede tomar- es usualmente mucho mayor que el **espacio de dirección** -el número de direcciones disponibles para registros. La función de seccionamiento mapea el espacio del campo seccionado al espacio de direcciones.

Una **colisión** ocurre cuando el valor del campo seccionado de un nuevo registro que está siendo insertado secciona a una dirección que ya contiene un registro distinto. En esta situación, debemos insertar al nuevo registro en alguna otra posición, debido a que su dirección de seccionado está ocupada. Al proceso de encontrar otra posición se le llama **resolución de colisión**. Existen numerosos métodos para la resolución de colisiones, incluyendo los siguientes:

- *Direccionamiento abierto*: Siguiendo adelante de la posición ocupada especificada por la dirección de seccionado, el programa checa las posiciones siguientes hasta encontrar una posición vacía. Para este fin puede usarse el Algoritmo 4.2(b).

- *Encadenamiento*: En este método, se guardan varias localidades de sobreflujo, usualmente extendiendo el arreglo con un número de posiciones de sobreflujo. Además, se agrega un apuntador de campo a cada localidad de registro. Una colisión se resuelve colocando al nuevo registro en una localidad de sobreflujo desocupada y colocando el apuntador de la sección ocupada en la dirección del sobreflujo.

Manteniendo una lista enlazada de registros sobreflujo para cada dirección seccionada, como se muestra en la Figura 4.8(b).

- *Seccionamiento múltiple*: Si en el primero resulta una colisión el programa aplica un segundo seccionamiento. Si resulta otra colisión, el programa utiliza direccionamiento abierto o aplica un tercer seccionamiento y de ser necesario usa el direccionamiento abierto.

Cada método de resolución de colisiones requiere su propio algoritmo de inserción, recuperación y borrado de registros. Los algoritmos de encadenamiento son los más sencillos. Los algoritmos de borrado para direccionamiento abierto son complicados.

La meta de una buena función de seccionado es distribuir los registros uniformememnte sobre el espacio direccionable para minimizar las colisiones sin dejar muchas localidades desocupadas. Los estudios de análisis y simulación han mostrado que es usualmente mejor guardar una tabla de seccionamiento entre el 70 y 90% de saturación de tal forma que el número de colisiones permanezca bajo y no desperdiciemos mucho espacio. Por lo tanto, si esperamos tener r registros para almacenarse en la tabla, debemos seleccionar M localidades para el espacio direccionable tal que (r/M) esté entre 0.7 y 0.9. También puede ser útil elegir un número primo para M, ya que se ha demostrado que esto distribuye las direcciones de seccionado mejor sobre el espacio direccionable cuando se usa la función de seccionado mod. Otras funciones de seccionado pueden requerir que M sea una potencia de 2.

4.8.2 Seccionado Externo

Al seccionado de archivos en disco se le llama **seccionado externo**. Para cumplir con las características de almacenamiento en disco, el espacio direccionable se conforma de **cubetas**, cada una de las cuales almacena varios registros. Una cubeta es un bloque de disco o un cluster de bloques contiguos. La función de seccionado mapea una llave en un número de cubeta relativo, en lugar de asignar un bloque de direcciones absoluta a la cubeta. Como se ilustra en la Figura 4.9 se guarda una tabla en el encabezado que convierte el número de cubeta a la dirección del bloque correspondiente en disco.

El problema de colisiones es menos severo en las cubetas, porque cabrán tantos registros dentro de una cubeta como puedan seccionar a la misma sin ocasionar problemas. Sin embargo, se deben tomar precauciones para los casos donde una cubeta se llena a capacidad y un nuevo registro se inserta seccionando a esa cubeta. Podemos usar una variante de encadenamiento en la cual mantenemos un apuntador en cada cubeta a una lista enlazada de registros de sobreflujo, como se muestra en la Figura 4.10. Los apuntadores en la lista enlazada deben ser **apuntadores a registro**, los cuales incluyen un bloque de direciones y una posición relativa del registro dentro del bloque.

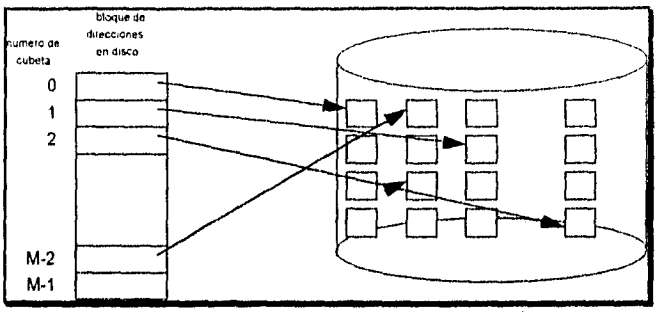


Figura 4.9 Números correspondientes de una cubeta a bloques en disco.

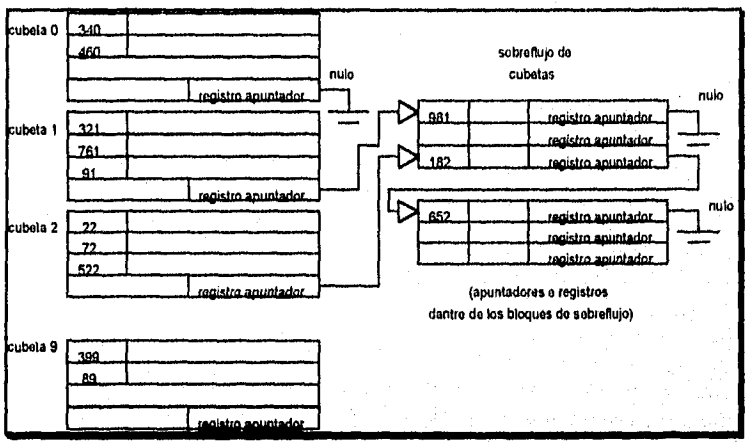


Figura 4.10 Manejo de sobreflujo en cubetas mediante encadenamiento.

A pesar de que el seccionado proporciona el acceso más rápido posible para recuperar un registro cualquiera, dado el valor de su campo seccionado, no es muy útil cuando se requieren otras aplicaciones del mismo archivo a menos que se construyan trayectorias adicionales de acceso. Por ejemplo, si queremos recuperar registros en el orden de sus campos seccionados, el seccionado no es muy aconsejable, debido a que la mayoría de las funciones de seccionado no mantienen a los registros ordenados. Algunas funciones de seccionado, llamadas **preservadoras de orden**, pueden mantener el orden de registros por el valor de los campos seccionados. Un ejemplo sencillo es tomar los tres dígitos más a la izquierda del campo número de pedido como la dirección de seccionado y guardar los registros sorteados por número de pedido dentro de cada cubeta. Otro ejemplo es usar una llave entera seccionada directamente como un índice a un archivo relativo, si los valores llave de seccionado llenan un intervalo en particular; por ejemplo, si los números de empleado de una compañía están asignados como 1, 2, 3... hasta el total de empleados, podemos usar la función de identidad de seccionado que mantiene el orden.

Desafortunadamente, esto solo trabaja si las llaves se generan ordenadamente por alguna aplicación.

Otra desventaja del seccionado es la cantidad fija de espacio localizado en el archivo. Supongamos que localizamos M cubetas para el espacio direccionable y dejamos que m sea el número máximo de registros que caben dentro de una cubeta; entonces cabrán como máximo m * M registros en el espacio localizable. Si el número de registros cambia para ser sustancialmente menor que m * M, estamos dejando muchos espacio sin uso. Por otro lado, si el número de registros incrementa para sustancialmente ser mayor que m * M, resultarán numerosas colisiones y la recuperación se hará más lenta debido a las largas listas de sobreflujo. En cualquier caso, podemos cambiar el número de bloques localizables y luego usar una función diferente de seccionado para redistribuir los registros entre las cubetas. Las organizaciones de archivos más recientes basadas en el seccionado permiten que varíe dinámicamente el número de cubetas.

En el seccionado externo regular; como en el caso de un archivo desordenado, la búsqueda de un registro dado el valor de algun campo distinto al campo seccionado es costoso. El borrado de registros puede implementarse eliminando al registro de su cubeta. Si la cubeta tiene una cadena de sobreflujo, podemos mover uno de los registros de sobreflujo a la cubeta para reemplazar el registro eliminado. Si el registro a borrar está ya en sobreflujo, simplemente lo eliminamos de la lista enlazada. Notemos que eliminar un registro del sobreflujo implica que debemos rastrear las posiciones vacías en el sobreflujo. Esto se puede lograr fácilmente manteniendo una lista enlazada de localidades de sobreflujo no usadas.

Modificar el valor de un campo depende de dos factores: la condición de búsqueda para localizar el registro, y el campo a modificar. Si la condición de búsqueda es una igualdad en el campo seccionado, podemos localizar el registro eficientemente usando la función de seccionado; de otra forma, debemos hacer una búsqueda lineal. Un campo no seccionado puede modificarse cambiando al registro y reescribiéndolo en la misma cubeta. Modificar el campo seccionado significa que el registro puede moverse a otra cubeta, lo cual requiere el borrado del registro previo seguido por la inserción del registro modificado.

4.8.3 Técnicas de Seccionado que Permiten Expansión Dinámica de Archivos

Una gran desventaja del esquema de seccionado *estático* es que la dirección de espacio seccionado es fijo. Por lo tanto, es difícil expander o comprimir al archivo dinámicamente. Los esquemas descritos en esta sección intentan remediar esta situación. Los primeros dos esquemas -seccionado dinámico y extendible- almacenan una estructura de acceso además del archivo, y por lo tanto son algo similares al indexado. La principal diferencia es que el acceso de estructura se basa en los valores que resultan después de la aplicación de la función de seccionado al campo de búsqueda. En el indexado, la estructura de acceso se basa en el valor del campo de búsqueda. La tercer técnica -seccionado lineal- no requiere del uso de ninguna estructura de acceso adicional.

Estos esquemas sacan ventaja del hecho de que el resultado de la mayoría de las funciones de seccionado es un entero no negativo y por lo tanto puede representarse como número binario. La estructura de acceso se construye sobre la **representación binaria** del resultado de la función de seccionado, lo cual es una cadena de **bits**. A esto se le llama el **valor seccionado** de un registro. Los registros se distribuyen entre las cubetas basados en los valores de los *bits más significativos* en sus valores de seccionado.

Seccionado dinámico: El número de cubetas no es fijo (como en el seccionado regular) pero crece o disminuye cuando es necesario. El archivo puede empezar con una sola cubeta; una vez que se llena, y se inserta un nuevo registro, la cubeta se **sobrellena** y se derrama hacia otra cubetas. Los registros se distribuyen entre las dos cubetas basadas en el valor del primer bit (el más significativo) de sus valores de seccionado. Los registros cuyos valores de seccionado empiezan con un 1 se almacenan en otra cubeta. En este momento, se construye una estructura de árbol binario, llamada **directorio** (o **índice**). El directorio tiene dos tipos de nodos:

- **Nodos internos** guían la búsqueda; cada uno tiene un apuntador izquierdo que corresponde a un bit 0 y un apuntador derecho que corresponde a un 1.

- **Nodos hoja** guardan un apuntador en una cubeta -una cubeta de dirección. La Figura 4.11 ilustra un directorio y las cubetas del archivo de datos.

ALGORITMO 4.3 El procedimiento de búsqueda para seccionado dinámico.

```
h ← valor seccionado del registro;
t ← nodo raíz del directorio;
i ← 1;
while t es un nodo interno del directorio do
        begin
        if el iésimo bit de h es 0
          then t ← izquierdo hijo de t;
          else t ← derecho hijo de t;
        i ← i + 1
end;
busca la cubeta cuya dirección está en el nodo t;
```

La búsqueda de un registro se lleva a cabo como se muestra en el Algoritmo 4.3. El directorio puede almacenarse en memoria principal a menos que se vuelva demasiado grande. Si el directorio no cabe en un bloque, se distribuye sobre dos o más niveles. Nótese que las entradas de directorio son algo compactas. Cada nodo interno almacena un bit etiqueta para especificar el tipo de nodo, más los apuntadores izquierdo y derecho. También puede ser necesario un apuntador padre. Cada nodo hoja almacena una cubeta dirección. Para reducir el espacio necesario por izquierdo, derecho, y apuntadores padres de nodos internos pueden usarse representaciones especiales de árboles binarios. En

general, si un directorio de x niveles se almacena en disco, necesitamos de x+1 bloques de acceso para recuperar una cubeta.

Si una cubeta se desborda, se divide en dos, y los registros se distribuyen basados en el siguiente bit significativo de cualquier valor seccionado. Por ejemplo, si se inserta un nuevo registro a la cubeta cuyo valor seccionado inicia con 10 -la cuarta cubeta en la Figura 4.11- y ocasiona sobreflujo; entonces todos los registros cuyo valor de seccionado inicie con 100 se colocan en la primer cubeta desbordada, y la segunda contiene a aquéllas cuyo valor de seccionado inicie con 101. El directorio se expande con un nuevo nodo interno para reflejar el desborde; este nodo apunta a dos nodos hoja que apuntan a las dos cubetas. Los niveles del árbol binario pueden expandirse dinámicamente. Sin embargo, el número de niveles no puede exceder el número de bits en el valor seccionado.

Si la función de seccionado distribuye a los registros uniformemente, el directorio árbol estará balanceado. Las cubetas pueden combinarse si una se vacía o si el número total de registros en dos cubetas vecinas puede caber en una sola. En este caso, el directorio pierde un nodo interno y los dos nodos hoja se combinan para formar un sólo nodo que apunta a la nueva cubeta. Así los niveles del árbol binario pueden comprimirse dinámicamente.

Seccionado extendible: Se mantiene un tipo de directorio diferente -un arreglo de 2 a la d direcciones cubeta-, donde a d se le llama la **profundidad global** del directorio. El valor entero corresponde a los primeros d bits (de mayor orden) del valor seccionado usado como índice al arreglo para determinar una entrada de directorio, la dirección en esa entrada determina la cubeta en la cual se almacenan los registros correspondientes. Sin embargo, no tiene que ser una cubeta distinta para cada una de las 2 a la d direcciones. Varias localidades con los mismos primeros d bits de sus valores de seccionado pueden contener la misma dirección si todos los registros que seccionan a estas localidades caben en una cubeta. Una **profundidad local** d' -almacenada en cada cubeta- especifica el número de bits sobre los cuales se basa el contenido de la cubeta. La Figura 4.12 muestra un directorio con profundidad global = 3.

El valor d puede crecer o disminuir de uno en uno. El duplicado es necesario si una cubeta cuya profundidad local d' es igual a la profundidad global de sobreflujo d. La mayoría de las recuperaciones de registros requieren dos bloques de acceso -uno al directorio, y el otro a la cubeta.

Para ilustrar el desbordamiento, supongamos que la inserción de un registro ocasiona un sobreflujo en la cubeta cuyo valor de seccionado inicia con 01 -la tercer cubeta en la Figura 4.12. Los registros se distribuirán entre dos cubetas: la primera contiene todos los registros cuyos valores de seccionado inician con 010, y la segunda con 011. Ahora las dos localidades de directorio 010 y 011 apuntan a las dos nuevas cubetas. Antes del desborde, apuntaban a la misma cubeta. La profundidad local d' de las dos cubetas nuevas es 3, la cual es mayor en uno que la profundidad local de la cubeta anterior.

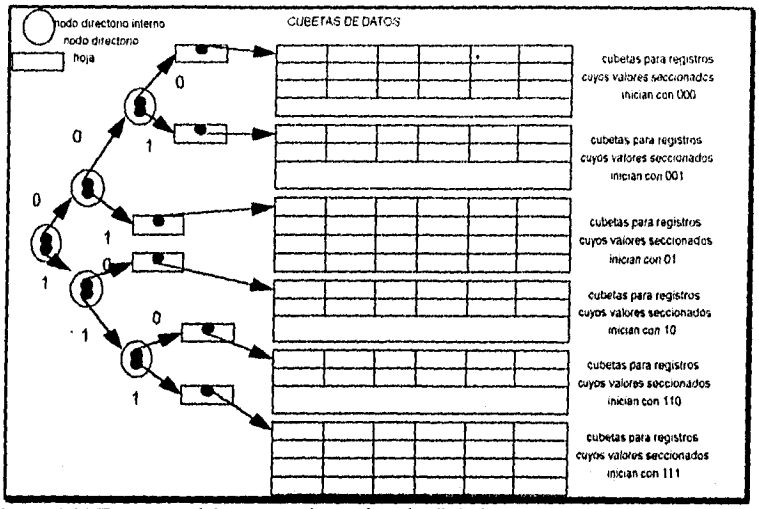


Figura 4.11 Estructura del esquema de seccionado dinámico

Si una cubeta que se desborda y se derrama solía tener una profundidad local d' igual a la profundidad global d del directorio; entonces el tamaño del directorio debe duplicarse de tal manera que podamos usar un bit extra para distinguir las dos cubetas nuevas. Por ejemplo, si la cubeta de registros cuyos valores de seccionado inician con 111 en la Figura 4.12, los dos sobreflujos etiquetados 110 y 111, y por lo tanto la profundidad global d=4, debido a que las dos cubetas están ahora etiquetadas 1110 y 1111, y por lo tanto sus profundidades locales = 4. El tamaño del directorio por lo tanto se duplica, y cada una de las localidades originales en el directorio también se derrama en dos localidades, ambas teniendo el mismo apuntador como en la localidad original. El tamaño máximo del directorio es 2 a la k, donde k es el número de bits en un valor seccionado.

Seccionado local: La idea es permitir un archivo seccionado para expandir y comprimir su número de cubetas dinámicamente *sin* la necesidad de un directorio. Supongamos que el archivo empieza con M cubetas numeradas 0, 1, ..., M-1 y se usa la función de seccionado mod h(k) = K mod M; a esta función se le llama función inicial de seccionado h_0. El sobreflujo debido a colisiones se sigue manejando manteniendo las cadenas individuales de sobreflujo de cada cubeta. Sin embargo, cuando una colisión conduce a un sobreflujo en *cualquier* cubeta, la *primer* cubeta del archivo -cubeta 0- se reparte en dos cubetas;: la original y una nueva al final del archivo M. Los registros que originalmente estaban en la cubeta 0 se distribuyen entre las dos basados en una función diferente de seccionado $h_1(k)$=K mod 2M. Una propiedad clave de las dos funciones de seccionado h_0 y h_1 es que cualquier registro que se haya seccionado a la cubeta 0 basado en h_0 seccionará a la cubeta 0 o M basado en h_1; esto es necesario para que trabaje el seccionado lineal.

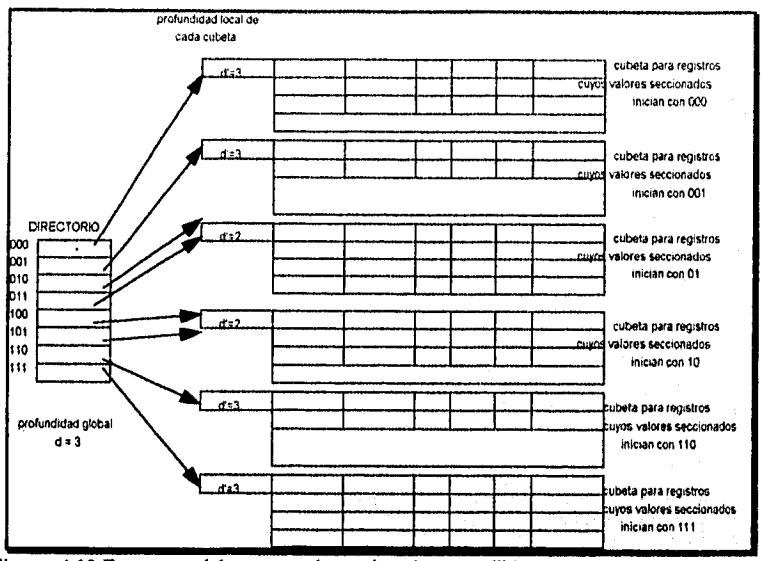


Figura 4.12 Estructura del esquema de seccionado extendible

Conforme más colisiones llevan al ocurrimiento de desbordes, las cubetas adicionales se reparten en orden *lineal* 1, 2, 3, ... Si ocurren suficientes sobreflujos, todas las cubetas se desbordan, de tal forma que los registros en sobreflujo se redistribuyen en cubetas regulares, usando la función h_1 vía un *retraso de desborde* de sus cubetas. No hay directorio; sólo es necesario un valor n para determinar qué cubetas han sido derramadas. Para recuperar un registro con valor de llave seccionada K, primero se aplica la función h_0 a K; si $h_0(k) < n$, entonces se aplica la función h_1 sobre k porque la cubeta ya está derramada. Inicialmente, n=0, indicando que la función h_0 se aplica a todas las cubetas; n crece linealmente conforme se vacían las cubetas.

Cuando n =M, todas las cubetas originales se han derramado y la función de seccionado h_1 se aplica a todos los registros del archivo. En este punto, n se resetea a 0, y cualquier nueva colisión que ocasione sobreflujo conduce al uso de una nueva función de seccionado $h_2(K)=K \bmod 4M$. En general, se usa una secuencia de funciones de seccionado $h_j(K)=K \bmod (2 \text{ a la } j\ M)$, donde j = 0, 1, 2, ...; es necesaria una función h_{j+1} para cada una de las cubetas 0, 1, ... (2 a la j M) -1 que se han derramado y n se resetea a 0. La búsqueda de un registro con valor de llave seccionada K se da en el Algoritmo 4.4.

ALGORITMO 4.4 Procedimiento de búsqueda en un seccionado lineal.

```
if n = 0
  then m ← hj(k) (*m es el valor seccionado del registro con llave seccionada K *)
  else begin
       m ← hj(k);
       if m < n then m ← hj+1(k)
       end;
busca la cubeta cuyo valor seccionado es m (y su sobreflujo; si lo hay);
```

4.9 Otras Organizaciones Primarias de Archivos

4.9.1 Archivos de Registros Mixtos

La organización de archivos que hemos estudiado hasta ahora asume que todos los registros de un archivo en particular son del mismo tipo. Los registros podrían ser de EMPLEADO, PROYECTO, ESTUDIANTE, O DEPARTAMENTOS, pero cada archivo contiene registros de un sólo tipo. En la mayoría de las aplicaciones de bases de datos, encontramos situaciones en las cuales numerosos tipos de entidades se interrelacionan de varias formas. Las relaciones entre registros de varios archivos puede representarse por **campos conectores**. Por ejemplo, un registro ESTUDIANTE puede tener un campo conector DEPTMAYOR cuyos valores dan el nombre del DEPARTAMENTO al cual pertenece su carrera. Este campo DEPTCARRERA *se refiere* a una entidad DEPARTAMENTO, lo cual debe representarse por un registro propio en el archivo DEPARTAMENTO. Si queremos recuperar valores de campo de dos registros relacionados, debemos recuperar primero uno de los registros. Luego podemos usar el valor del campo conector para recuperar el registro relacionado en el otro archivo. Por lo tanto, las relaciones se implementan por **referencias lógicas de campo** dentro de los registros en distintos archivos.

Las organizaciones de archivos en DBMSs jerárquicos y de red implementan relaciones entre registros como **relaciones físicas** realizadas por proximidad física de registros relacionados o apuntadores físicos. Estas organizaciones típicamente asignan un **área** del disco para retener registros de más de un tipo de tal forma que los registros puedan relacionarse físicamente. Si se espera utilizar una relación en particular con frecuencia, implementar la relación física puede mejorar la eficiencia del sistema al recuperar registros relacionados. Por ejemplo, si el query para recuperar un registro de DEPARTAMENTO y todos los registros de ESTUDIANTE con carrera en ese departamento es muy frecuente, sería deseable colocar cada registro DEPARTAMENTO y su cluster de ESTUDIANTE contiguamente en disco en un archivo mixto.

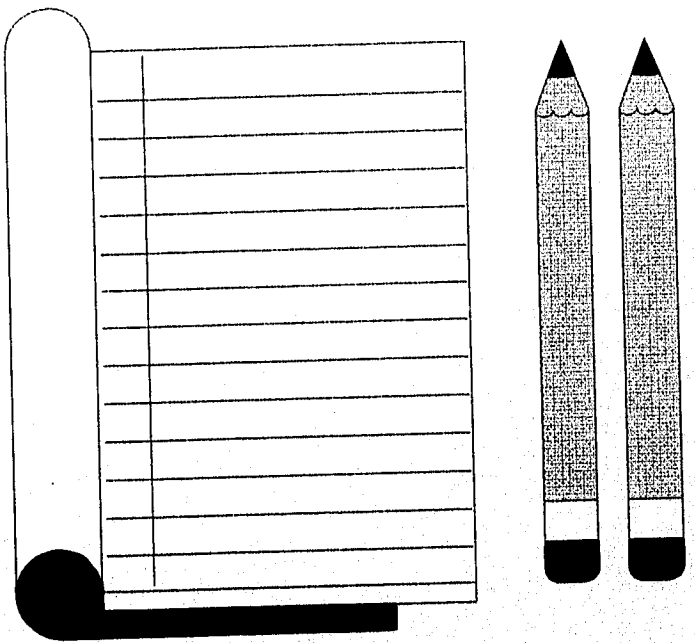
Para distinguir los registros de un archivo mixto, cada registro tiene -además de sus valores de campo- un campo **tipo de registro**. Este es típicamente el primer campo en cada registro y se usa por el software del sistema para determinar el tipo de registro que se

va a procesar. Usando la información del catálogo, el DBMS puede determinar los campos de ese tipo de registro y sus tamaños, para inetrpretar los datos.

4.9.2 B-trees y Otras Estructuras de Datos

Pueden emplearse otras estructuras para la organización primaria de archivos. Por ejemplo, si el tamaño del registro y el número de registros en un archivo es pequeño, algunos DBMSs ofrecen la opción de usar B-trees como organización primaria.

CAPITULO 5



Modelo de Datos y Algebra Relacional

5.1 Conceptos del Modelo Relacional

El modelo relacional representa a la base de datos como una colección de relaciones. Informalmente, cada relación se parece a una tabla o, a alguna extensión, un simple archivo. Como pronto veremos, existen diferencias importantes entre relaciones y archivos.

Cuando se piensa en una relación como una tabla de valores, cada renglón en la tabla representa una colección de valores de datos relacionados. Estos valores pueden interpretarse como términos describiendo una entidad o relación del mundo real. Los nombres y columnas de la tabla se usan para ayudar en la interpretación del significado de los valores de cada renglón. Por ejemplo, una tabla se puede llamar ESTUDIANTE porque cada renglón representa hechos acerca de una entidad estudiante en particular. Los nombres de las columnas -Nombre, NúmeroEstudiante, Clase- especifican cómo interpretar los valores de cada renglón, basado en el valor de cada columna. Todos los valores de una columna son del mismo tipo.

En terminología del modelo relacional, a un renglón se le llama *tupla*, al encabezado de una columna se le llama *atributo*, y a la tabla se le llama *relación*. A los tipos de datos que describen los valores que pueden aparecer en cada columna se les llama *dominio*.

5.1.1 Dominios, Tuplas, Atributos y Relaciones

Un *dominio* D es un conjunto de valores atómicos. Por atómico, entendemos que cada valor en el dominio es indivisible tanto como al modelo relacional le atañe. Es útil especificar un nombre para el dominio, que ayude en la interpretación de sus valores. Algunos ejemplos de dominios son los siguientes:

- Números_telefónicos_México: conjunto de 11 dígitos válidos en la República Mexicana.
- Números_telefónicos_locales: conjunto de 7 números telefónicos válidos dentro de un área de código en particular.
- Números de cartilla: conjunto válido de 7 dígitos con números de cartilla.
- Nombres: conjunto de nombres de personas.

Las anteriores son definiciones lógicas de dominio. Para cada dominio se especifica un **tipo de dato** o **formato**. Por ejemplo, el tipo de datos para el dominio números_telefónicos_México puede declararse como una cadena de caracteres de la forma (ddd)d-dd-dd-dd donde cada d es un dígito (decimal) y los tres primeros dígitos forman un código de área válido. El tipo de dato para edad_empleado es un entero entre 16 y 80.

Así se le asigna nombre, tipo de dato y formato a un dominio. También puede darse información adicional para la interpretación de los valores de un dominio; por ejemplo, un dominio numérico tal como peso_persona debe de tener las unidades de medición especificadas -libras o kilogramos.

Un esquema relacional R, denotado por R(A1, A2..., An), se conforma del nombre de la relación R y una lista de atributos A1, A2,...An. Cada atributo Ai es el nombre de un papel jugado por algún dominio D en el esquema relacional R. Al dominio de A1 se le llama D y se denota por dom(Ai). Para describir una relación se usa un esquema de relación; R es el nombre de la relación. El grado de una relación es el número de atributos n de su esquema de relación.

Un ejemplo de esquema de relación de grado 7, el cual describe estudiantes universitarios, es el siguiente:

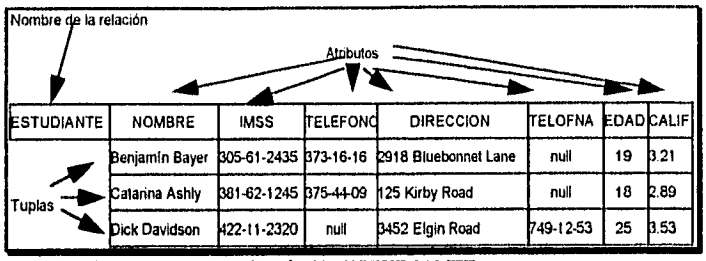
ESTUDIANTE(Nombre, NC, Teléfono, Dirección, TeléfonoOficina, Edad, Cal)

Para este esquema de relación, ESTUDIANTE es el nombre de la relación, la cual tiene siete atributos. Para algunos de los atributos de la relación ESTUDIANTE podemos especificar los siguientes dominios: dom(Nombre)=Nombres; dom(NC)=Número de cartilla; dom(Teléfono)=Número telefónico local; dom(TeléfonoOficina)= número_telefónico_local; dom(Cal) = Promedio.

Una relación r del esquema de relación R(A1, A2,...An), también denotada por r(R), es un conjunto de n-tuplas $r=\{t_1, t_2, \ldots t_m\}$. Cada n-tupla t es una lista ordenada de valores $t = <V_1, V_2, \ldots V_n>$, donde cada valor V_i, $1 \leq i \leq n$, es un elemento del dominio (Ai) o es un valor especial nulo. También son comúnmente usados los términos intensión de relación para el esquema R y extensión de relación para una instancia de la relación r(R).

La Figura 5.1 muestra un ejemplo de una relación ESTUDIANTE, la cual corresponde al esquema ESTUDIANTE especificado con anterioridad. Cada tupla en la relación representa una entidad de estudiante en particular. Desplegamos la relación como una tabla, donde cada tupla se muestra como un renglón y cada atributo corresponde a un encabezado de columna indicando un papel o interpretación de los valores en esa columna. Los valores nulos representan atributos cuyos valores son desconocidos o no existen para algunas tuplas individuales de ESTUDIANTE.

La definición anterior de una relación puede reestablecerse como sigue. Una relación r(R) es un subconjunto del producto Cartesiano de los dominios que definen a R.



ESTUDIANTE	NOMBRE	IMSS	TELEFONO	DIRECCION	TELOFNA	EDAD	CALIF
	Benjamín Bayer	305-61-2435	373-16-16	2918 Bluebonnet Lane	null	19	3.21
	Catarina Ashly	381-62-1245	375-44-09	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-12-53	25	3.53

Figura 5.1 Atributos y tuplas de la relación ESTUDIANTE

El producto Cartesiano especifica todas las posibles combinaciones de valores desde los dominios subyacentes. Por lo tanto, si definimos el número de valores o **cardinalidad** de un dominio D por |D|, y asumimos que todos los dominios son finitos, el número total de tuplas en el producto Cartesiano será

$$|dom(A_1)| * |dom(A_2)| *...*|dom(A_n)|$$

Fuera de todas estas posibles combinaciones, una instancia de relación, en un momento dado, -el **estado actual de la relación**- refleja sólo las tuplas válidas que representan a un estado en particular del mundo real. En general, como el estado del mundo real cambia, así lo hará la relación, siendo transformada en otro estado de la misma. Sin embargo, el esquema R es relativamente estático y no cambia excepto muy de vez en cuando -por ejemplo, como resultado de agregar un atributo para representar nueva información que no fue almacenada originalmente en la relación.

Es posible que varios atributos tengan el mismo dominio. Los atributos indican diferentes **papeles**, o interpretaciones, para el dominio. Por ejemplo, en la relación ESTUDIANTE, el mismo dominio números_telefónicos_locales juega el rol de telcasa, refiriéndose al "número telefónico de un estudiante", y el rol de telofna, refiriéndose al "número telefónico de la oficina del estudiante".

5.1.2 Características de las Relaciones

La definición anterior de relaciones implica ciertas características que hacen a una relación diferente de un archivo o tabla.

Ordenamiento de Tuplas en una Relación. Una relación se define como un conjunto de tuplas. Matemáticamente, los elementos de un conjunto no tienen orden entre ellos; por lo tanto, las tuplas en una relación no tienen ningún orden en particular. Sin embargo, en un archivo, los registros están físicamente almacenados en disco de tal forma que siempre existe un orden entre ellos. Este ordenamiento indica al primero, segundo, iésimo, y último registro en el archivo. De igual manera, cuando desplegamos una relación como una tabla, los renglones están desplegados bajo cierto orden.

ESTUDIANTE	Nombre	IMSS	TelCasa	Dirección	TelOfna	Edad	Prom
	Dick Davidson	422-11-2320	null	3452 Elgi	749-1253	25	3.53
	Barbara Benson	533-69-1238	839-8461	7384 Font	null	19	3.25
	Charles Cooper	489-22-1100	376-9821	265 Lark	749-6492	28	3.93
	Catarina Ashly	381-62-1245	375-4409	125 Kirby	null	18	2.89
	Benjamín Bayer	305-61-2435	373-1616	2918 Blue	null	19	3.21

Figura 5.2 La misma relación ESTUDIANTE de la Figura 5.1 con un orden de renglones diferente.

El ordenamiento de tuplas no es parte de la definición de una relación, porque una relación intenta representar hechos a un nivel lógico o abstracto. En una relación pueden especificarse muchos ordenes lógicos; por ejemplo, las tuplas en la relación ESTUDIANTE de la Figura 5.1 podrían ordenarse lógicamente por los valores de Nombre, Edad, o algún otro atributo. La definición de relación no especifica ningún orden: no hay preferencia para un ordenamiento lógico sobre otro. Por lo tanto, la relación desplegada en la Figura 5.2 se considera idéntica a la mostrada en la Figura 5.1. Cuando una relación se implementa como archivo, puede especificarse un ordenamiento físico sobre los registros del archivo.

Ordenamiento de Valores Dentro de una Tupla, y definición de una Relación Alternativa. De acuerdo a la definición de relación, una n-tupla es una lista ordenada de n valores, así es importante el ordenamiento de valores en una tupla -y por lo tanto de atributos en una definición de esquema de relación. Sin embargo, a nivel lógico, el orden de los atributos y sus valores no son realmente importantes mientras se mantenga la correspondencia entre atributos y valores.

Puede darse una **definición alterna de relación**, haciendo innecesario el ordenamiento de valores en una tupla. En esta definición, un esquema de relación $R=\{A_1, A_2, ..., A_n\}$ es un conjunto de atributos, y una relación $r(R)$ es un conjunto finito **de mapeos** $r=\{t_1, t_2, ..., t_m\}$, donde cada tupla t_i es un mapeo de R a D, y D es la unión del dominio de los atributos; esto es, $D = dom(A_1) \cup dom(A_2) \cup ... \cup dom(A_n)$. en esta definición, $t(A_i)$ debe estar dentro del $dom(A_i)$ para $1 \leq i \leq n$ para cada mapeo t en r. A cada mapeo t_i se le llama tupla.

De acuerdo a esta definición, una **tupla** puede considerarse como un **conjunto** de pares (<atributo>, <valor>), donde cada par da el valor del mapeo de un atributo A_i para un valor V_i del $dom(A_i)$. El ordenamiento de atributos no es importante, debido a que el nombre del atributo aparece con su valor. Por esta definición, las dos tuplas mostradas en la Figura 5.3 son idénticas. Esto tiene sentido a un nivel lógico o abstracto, debido a que no hay razón para preferir tener un valor de atributo antes que otro en una tupla.

t=<(Nombre, Dick Davidson),(NC,422-11-2320),(Telcasa, null),(Dirección, 3452),(Telofna, 749-1253),(Edad, 25),(Cal, 3.53)>

t=<(Dirección, 3452),(Nombre, Dick Davidson),(NC,422-11-2320),(Edad, 25),(Telofna, 749-1253),(Cal, 3.53),(Telcasa, null)>

Figura 5.3 Dos tuplas idénticas cuando el orden de atributos y valores no es parte de la definición en una relación.

Cuando se implementa una relación como archivo, los atributos pueden ordenarse físicamente como campos dentro de un registro. Usaremos la primera definición de relación, donde los atributos y los valores dentro de las tuplas están ordenados, debido a que simplifica mucho la notación. Sin embargo, la definición alterna dada aquí es más general.

Valores en las Tuplas. Cada valor en una tupla es un valor **atómico**; esto es, no es divisible en componentes dentro del marco del modelo relacional. Por lo tanto, no se permiten atributos compuestos y multivaluados. Mucha de la teoría detrás del modelo relacional fue desarrollada con esta suposición en mente, a la cual es le llama **primer forma de normalización**. Los atributos multivaluados deben representarse mediante relaciones separadas, y los atributos compuestos mediante sus componentes sencillos. Recientes investigaciones en el modelo relacional intentan eliminar estas restricciones usando el concepto de **no primer forma de normalización** o relaciones **anidadas**.

Los valores de algunos atributos dentro de una tupla en particular pueden ser desconocidos o no aplicar a esa tupla. En estos casos, se usa un valor especial llamado **nulo**. Por ejemplo, en la Figura 5.1, algunas tuplas de estudiantes tienen nulo en teléfono de oficina, presumiblemente porque no tienen teléfono en el trabajo o porque no lo conocen. En general, podemos tener varios tipos de valores nulos, tales como "valor desconocido", "el atributo no aplica a esta tupla", o "esta tupla no tiene valor para este atributo". Algunas implementaciones actualmente desarrollan diferentes códigos para diferentes tipos de valores nulos.

Interpretación de una Relación. El esquema de relación puede interpretarse como una declaración o un tipo de aserción. Por ejemplo, el esquema de la relación ESTUDIANTE declara que, en general, una entidad estudiante tiene un nombre, NC, Teléfono, Dirección, Edad y Cal. Cada tupla en la relación puede entonces interpretarse como un **término** o una instancia en particular de la aserción. Por ejemplo, la primer tupla en la Figura 5.1 aserta el factor que existe un ESTUDIANTE cuyo nombre es Benjamín Bayer, CN 305-61-2435, de edad 19, etc.

Algunas relaciones pueden representar términos acerca de entidades, considerando que otras relaciones pueden representar términos acerca de relaciones. Por ejemplo, un esquema de relación CARRERAS (NCEstudiante, CódigoDepto) aserta a los estudiantes con carreras en departamentos académicos; una tupla en esta relación relaciona a un estudiante a su departamento de carrera. Por lo tanto, el modelo relacional representa términos acerca de ambas entidades y relaciones uniformemente como relaciones.

5.1.3 Notación del Modelo Relacional

En nuestra presentación utilizaremos la siguiente notación:

- Un esquema de relación R de grado n se representa por $R(A_1, A_2, ..., A_n)$.
- Una n-tupla t en una relación r(R) se representa por $t = <V_1, V_2, ..., V_n>$, donde Vi es el valor correspondiente al atributo Ai. La siguiente notación se refiere a **valores componentes** de tuplas:

  - t[Ai] se refiere a los valores Vi en t para el atributo Ai.
  -t[A_u, A_w, ..., A_z], donde A_u, A_w, ..., A_z es una lista de atributos de R, se refiere a la subtupla de valores $<V_u, V_w, ..., V_z>$ desde t correspondiente a los atributos especificados en la lista.

- Las letras Q, R, S representan nombres de relaciones.
- Las letras q, r, s representan estados de relaciones.
- Las letras t, u, v representan tuplas.
- En general, el nombre de una relación tal como ESTUDIANTE indica el conjunto actual de tuplas en esa relación -por ejemplo, ESTUDIANTE.Nombre o ESTUDIANTE.Edad.

Considere la tupla t = <'Barbara Benson', '533-69-1238', '839-8461', '7384 Fontana Lane', nulo, 19, 3.25> de la relación ESTUDIANTE en la Figura 5.1; tenemos t[Nombre] = <'Barbara Benson'>, y t[NC, Cal, Edad] = <533-69-1238', 3.25, 19>.

5.2 Argumentos del Modelo Relacional

Existen varios tipos de argumentos que pueden especificarse en un esquema relacional. Estos incluyen argumentos de dominio, integridad de la entidad e integridad referencial. Otros tipos de argumentos, llamados dependencias de datos (los cuales incluyen dependencias funcionales y dependencias multivaluadas), se usan principalmente para el diseño de bases de datos por normalización.

5.2.1 Argumento del Dominio

Especifica que el valor de cada atributo A debe ser un valor atómico del dominio dom(A) para ese atributo. Los tipos de datos asociados con dominios típicamente incluyen tipos de datos numéricos estandar para enteros (tales como short integer, integer, long-integer) y números reales (float y float doble precisión). Caracteres, cadenas de longitud fija, y cadenas de longitud variable, como son fechas, horas, etc. Otros posibles dominios pueden

describirse por un subrango de valores de un tipo de datos o como un tipo enumerado donde todos los valores posibles están explícitamente enlistados.

5.2.2 Argumento de Llaves

Por definición, todos los elementos de un conjunto son diferentes; por lo tanto, todas las tuplas en la relación deben ser distintas. Esto significa que no existen dos tuplas que tengan la misma combinación de todos los atributos. Usualmente, hay otros **subconjuntos de atributos** de un esquema de relación **R** con la propiedad de que ningún par de tuplas en ninguna instancia de relación r de R debe tener la misma combinación de valores para estos atributos. Supóngase que se denota uno de tales subconjuntos de atributos por SK; entonces para cualquier par de tuplas distintas t1 y t2 en una instancia de relación r de R, tenemos el argumento que:

$t_1[SK] <> t_2[SK]$

A cualquier conjunto de atributos SK se le llama **superllave** del esquema de relación R. Cada relación tiene al menos una superllave -el conjunto de todos sus atributos. Una superllave puede tener atributos redundantes, sin embargo, un concepto más util es el de llave, el cual no tiene redundancia. Una llave K de un esquema de relación R es una superllave de R con la propiedad adicional de que eliminando cualquier atributo A de K deja un conjunto de atributos K' que no es una superllave de R. Por lo tanto, una llave es una superllave mínima; una superllave de la cual no podemos eliminar ningún atributo y seguir teniendo el único argumento que retener.

Por ejemplo, consideremos la relación ESTUDIANTE. El conjunto de atributos {NC}, es una llave de ESTUDIANTE porque ningún par de tuplas de estudiantes puede tener el mismo valor en NC. Cualquier conjunto de atributos que incluya NC -por ejemplo, {NC, Nombre, Edad}- es una superllave. Sin embargo, la superllave {NC, Nombre, Edad} no es una llave de ESTUDIANTE, porque eliminando Nombre o edad o ambos del conjunto siguen dejándonos con una superllave.

El valor de un atributo llave puede utilizarse para identificar de manera única a una tupla en la relación. Por ejemplo, el valor de NC 305-61-2435 identifica de manera única a la tupla correspondiente a Benjamín Bayer en la relación ESTUDIANTE. Nótese que el conjunto de atributos que constituyen una llave es una propiedad del esquema de relación; es un argumento que debe retenerse en cada instancia de relación del esquema. Una llave se determina del significado de los atributos en el esquema de relación. Por lo tanto, la propiedad es invariante en el tiempo; debe continuar reteniéndola cuando se inserten nuevas tuplas en la relación. Por ejemplo, no podemos ni debemos designar el atributo nombre de la relación ESTUDIANTE como una llave, porque no hay garantía de que nunca existirán dos estudiantes con nombres idénticos.

En general, un esquema de relación puede tener más de una llave. En este caso, a cada una de las llaves se le llama **candidato de llave**. Por ejemplo, la relación CAR en la Figura 5.4

tiene dos candidatos a llaves: NúmeroLicencia y NúmeroSerieMáquina. Es común designar a uno de los candidatos a llave como la **llave primaria** de la relación. Esta es el candidato de llave cuyos valores se usan para identificar tuplas en la relación. Usamos la convención de que los atributos que forman la llave primaria de un esquema de relación están subrayados. Nótese que, cuando un esquema de relación tiene varios candidatos a llave, la elección de una para volverse llave primaria es arbitraria; sin embargo, es usualmente mejor elegir una llave primaria con un atributo sencillo o un pequeño número de atributos.

CAR	NumLicencia	NumSerMotor	Fabrica	Modelo	Año
	Texas ABC-739	A69352	Ford	Mustang	90
	Florida TVP-347	B43696	Oldsmobile	Cutlass	93
	New York MPO-22	X83554	Oldsmobile	Delta	89
	California 432-TFY	C43742	Mercedes	190-D	87
	California RSK-629	Y82935	Toyota	Camry	92
	Texas RSK-629	U028365	Jaguar	XJS	92

Figura 5.4 La relación CAR con dos candidatos a llaves: NumLicencia y NumserMotor

5.2.3 Esquemas de Bases de Datos Relacionales y Argumentos de Integridad

Una base de datos relacional usualmente contiene muchas relaciones, con tuplas en relaciones que se relacionan de varias formas. Un **esquema de bases de datos relacionales** es un conjunto de esquemas de relación $S = \{R_1, R_2, \ldots, R_n\}$ y un **conjunto argumentos de integridad** IC. Una instancia de base de datos relacional DB de S es un conjunto de instancias de relación $DB = \{r_1, r_2, \ldots, r_n\}$ de tal forma que cada r_i es una instancia de R_i y que las r_i relaciones satisfacen los argumentos de integridad especificados en IC. La Figura 5.5 muestra un esquema de bases de datos relacional al que llamremos COMPAÑIA, y la Figura 5.6 muestra una instancia de base de datos relacional correspondiente al esquema COMPAÑIA. Cuando nos referimos a una base de datos relacional, implícitamente incluimos tanto su esquema como su instancia actual.

En la Figura 5.5, el atributo DNUMERO tanto en DEPARTAMENTO y DEPT_LOC establecen el mismo concepto del mundo real -el número asignado a un departamento. Ese mismo concepto se llama DN en EMPLEADO y DNUM en proyecto. Es permisible que en diferentes relaciones un atributo que representa al mismo concepto del mundo real tenga nombres que puedan o no ser idénticos. Similarmente, es permisible que los atributos que representan diferentes conceptos tengan el mismo nombre en diferentes relaciones. Por ejemplo, podríamos haber usado el atributo NOMBRE para PNOMBRE de PROYECTO y DNOMBRE de DEPARTAMENTO; en este caso, tendríamos dos atributos que comparten el mismo nombre pero representan diferentes conceptos reales - nombres de proyectos y de departamentos.

EMPLEADO

NOMBRE	MINIT	APELLID	IMSS	FECNA	DIREC	SEXO	SALARIO	SUPER	DN

DEPARTAMENTO

DNOMBRE	DNUMERO	MGRSSN	MGRFECINI

LOCALIDAD_DEPTO

DNUMERO	DLOC

PROYECTO

PNOMBRE	PNUMERO	PLOC	DNUM

TRABAJO

ESSN	PNO	HORAS

DEPENDENCIA

ESSN	NOM_DEP	SEXO	FECHANAC	RELACION

Figura 5.5 Esquema de la base de datos relacional COMPAÑIA; las llaves primarias están subrayadas

En algunas versiones anteriores del modelo relacional, se hizo la suposición de que el mismo concepto del mundo real, cuando es representado por un atributo, tendría nombres de atributo idénticos en todas las relaciones. Esto crea problemas cuando el mismo concepto del mundo real se usa en diferentes papeles (significados) en la misma relación. Por ejemplo, el concepto de número de seguro social aparece dos veces en la relación EMPLEADO: una vez en el papel de número de seguro social del empleado, y otra en el del supervisor. Para evitar problemas, asignamos nombres diferentes a los atributos -IMSS y SUPER, respectivamente.

Los argumentos de integridad se especifican en un esquema de bases de datos y se espera retengan a cada instancia de ese esquema. Además del dominio y argumentos de llave, se consideran parte del modelo relacional otros dos tipos de argumentos: integridad de entidad e integridad referencial.

5.2.4 Integridad de Entidad, Integridad Referencial, y Llaves Externas

El **argumento de integridad de entidad** establece que ningún valor de llave primaria puede ser nulo. Esto se debe a que el valor de llave primaria se usa para identificar algunas tuplas. Por ejemplo, si dos o más tuplas tienen nulas sus llaves primarias, podríamos no ser capaces de distinguirlas.

Los argumentos de llave y de integridad de entidad se especifican en relaciones individuales. El argumento de integridad referencial se especifica entre dos relaciones y se usa para mantener la consistencia entre tuplas de ambas relaciones. Informalmente, la consistencia de integridad referencial establece que una tupla en una relación que refiere a

otra, debe hacerlo a una tupla existente en esa relación. Por ejemplo, en la Figura 5.6, el atributo DN de EMPLEADO da el número de departamento para el cual cada empleado trabaja; por lo tanto, su valor en cada tupla de EMPLEADO debe corresponder al valor DNUMERO de alguna tupla en la relación DEPARTAMENTO.

Para definir la integridad referencial más formalmente, debemos definir primero el concepto de llave externa. Las condiciones de una llave externa, especifican un argumento de integridad referencial entre los dos esquemas de relación R1 y R2. Un conjunto de atributos FK en relación al esquema R1 es una **llave externa** de R1 si satisface las siguientes reglas:

1. Los atributos en FK tienen el mismo dominio que los atributos de la llave primaria PK de otro esquema de relación R_2; los atributos de FK se dicen **referenciar** o **referir a** la relación R_2.

2. Un valor de FK en una tupla t_1 de R_1 ocurre como un valor de PK para alguna tupla t_2 en R_2 o es nula. En el caso anterior, tenemos $t_1[FK] = t_2[PK]$, y decimos que la tupla t_1 **referencia** o **refiere a** la tupla t_2.

En una base de datos de muchas relaciones, hay usualmente muchos argumentos de integridad referencial. Para especificar estos argumentos, debemos entender primero claramente el significado del papel que cada conjunto de atributos juega en los distintos esquemas de relación de la base de datos. Los argumentos de integridad referencial típicamente surgen de las relaciones entre las entidades representadas por los esquemas de relación. Por ejemplo, consideremos la base de datos mostrada en la Figura 5.6. En la relación EMPLEADO, el atributo DN se refiere al departamento en el cual un empleado trabaja; por lo tanto, designamos a DN como una llave externa de EMPLEADO, refiriendo a la relación DEPARTAMENTO. Esto significa que un valor de DNO en cualquier tupla t_1 de la relación EMPLEADO debe corresponder con un valor de la llave primaria de DEPARTAMENTO -el atributoDNUMERO- en alguna tupla t_2 de la relación DEPARTAMENTO, o el valor de DN puede ser nulo si el empleado no pertenece a algún departamento. En la Figura 5.6, la tupla del empleado "John Smith" referencia a la tupla del departamento "Investigación", indicando que "John Smith" trabaja en este departamento.

Notemos que una llave externa puede referir a su propia relación. Por ejemplo, el atributo SUPERIMSS se refiere al supervisor de un empleado; este es otro empleado representado por una tupla en la relación EMPLEADO. Por lo tanto, SUPERIMSS es una llave externa que refernecia a la relación EMPLEADO. En la Figura 5.6 la tupla del empleado "John Smith" referencia a la tupla del empleado "Franklin Wong", indicando que el último es supervisor del primero.

Podemos desplegar diamagramalmente el argumento de integridad referencial dibujando un arco directo de cada llave externa a la relación que referencia. Para mayor claridad, la flecha puede apuntar a la llave primaria de la relación referenciada. La Figura 5.7 muestra

el esquema en la Figura 5.5 con argumentos de integridad referencial desplegados de esta manera.

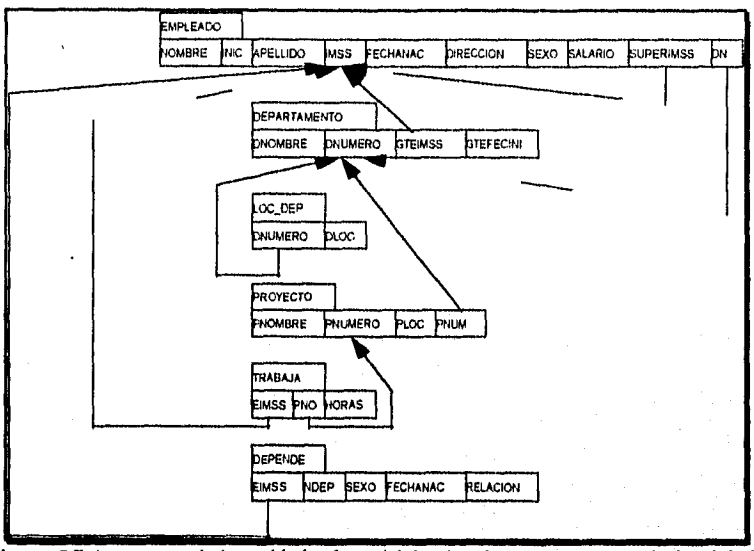


Figura 5.7 Argumento de integridad referencial desplegada en el esquema relacional de la base de datos COMPAÑIA

Todos los argumentos de integridad deben especificarse en el esquema de la base de datos si es que nos interesamos en mantener estos argumentos en todas las instancias de la base de datos. Por lo tanto, en un sistema relacional, el lenguaje de definición de datos (DDL) debe incluir provisiones para especificar los diferentes tipos de argumentos de tal forma que el DBMS pueda reforzarlos automáticamente. La mayoría de los sistemas manejadores de bases de datos relacionales soportan argumentos de integridad de llave e identidad, y muchos sistemas están haciendo provisiones para soportar la integridad referencial.

Los tipos precedentes de argumentos no incluyen a una gran clase de argumentos generales, algunas veces llamados de integridad semántica, que pueden tener que especificarse y reforzarse en una base de datos relacional. Ejemplos de tales argumentos son "el salario de un empleado no debe exceder al salario del supervisor" y "el 'número máximo de horas que un empleado puede trabajar en todos los proyectos por semana es de 56". Tales argumentos no se forzan en la mayoría de los DBMSs comerciales, pero se están desarrollando los mecanismos para especificar y forzar estos argumentos.

5.3 Operaciones de Actualización en las Relaciones

Existen tres operaciones básicas de actualización en las relaciones: insertar, borrar y modificar: La **inserción** se usa para insertar una nueva tupla o tuplas en una relación; **borrar** se usa para borrar tuplas; y **modificar** se usa para cambiar los valores de algunos atributos. Cada vez que se aplican operaciones de actualización, no deben violarse los argumentos especificados en el esquema de la base de datos relacional. En esta sección discutiremos los tipos de argumentos que pueden violarse en cada operación de actualización y las acciones que pueden tomarse si una actualización causa una violación.

5.3.1 La Operación Insert

Proporciona una lista de los valores de los atributos para una nueva tupla t que va a ser insertada a una relación R. La inserción puede violar cualquiera de los cuatro tipos de argumentos discutidos en la sección anterior. Los argumentos de dominio pueden violarse si se da un valor de atributo que no aparece en el dominio correspondiente. Los argumentos de llave pueden violarse si un valor llave en la nueva tupla t ya existe en otra tupla en la relación r(R). La entidad de integridad puede violarse si la llave primaria de la nueva tupla t es nula. La integridad referencial puede violarse si el valor de cualquier llave externa en t refiere a una tupla que no existe en la relación referenciada. Aqul están algunos ejemplos para ilustrar esta discusión:

1. Insert <'Cecilia', 'F', 'Kolonsky', '677678989', '05-ABR-50', '6537 Windy Lane, Katy, TX', F, 28000, nulo, 4> into EMPLEADO

   -Esta inserción satisface todos los argumentos, por lo tanto es aceptable.

2. Insert <'Alicia', 'J', 'Zelaya', '999887777', '05-ABR-50', '6537 Windy Lane, Katy, TX', F, 28000, '987654321', 4> into EMPLEADO

   -Esta inserción viola el argumento de llave porque otra tupla con el mismo valor de IMSS existe en la relación EMPLEADO.

3. Insert <'Cecilia', 'F', 'Kolonsky', nulo, '05-ABR-50', '6337 Windy Lane, Katy, TX', F, 28000, nulo, 4> into EMPLEADO.

   -Esta inserción viola el argumento de integridad de entidad (null para la llave primaria IMSS), de tal forma que es inaceptable.

4. Insert Insert <'Cecilia', 'F', 'Kolonsky', nulo, '05-ABR-50', '6357 Windy Lane, Katy, TX', F, 28000, '987654321', 7> into EMPLEADO.

   -Esta inserción viola el argumento de integridad referencial especificada en DN porque no existe tupla en DEPARTAMENTO con DNUMERO = 7.

Si una inserción viola uno o más argumentos, hay dos opciones disponibles. La primera es rechazar la inserción. En este caso, sería útil si el DBMS pudiera explicar al usuario por qué se rechaza la inserción. La segunda opción es intentar corregir la razón por la que se rechaza la inserción. Por ejemplo, en la operación número 3, el DBMS podría pedir al usuario proporcionar un IMSS y aceptar la inserción si fuera proporcionado un valor válido de IMSS. En la operación 4, el DBMS podría ya sea pedir al usuario cambiar el valor de DN a algún valor válido (o ponerlo nulo), o podría pedir al usuario insertar una tupla DEPARTAMENTO con DNUMERO = 7 y aceptar la inserción sólo después de que tal operación fuera aceptada.

5.3.2 La Operación Delete

Puede violar sólo a la integridad referencial, si la tupla que está siendo borrada es referenciada por llaves externas de otras tuplas en la base de datos. Para especificar borrado, una condición en los atributos de la relación selecciona la tupla (o tuplas) a ser borrada(s). Aquí están algunos ejemplos:

1. Borrar la tupla de TRABAJO con IMSS = '999887777' y PNO = 10

   - Este borrado es aceptable.

2. Borrar la tupla de EMPLEADO con IMSS = '999887777'

   - Este borrado no es aceptable, porque dos tuplas en TRABAJO se refieren a esta. Por lo tanto, si la tupla es borrada, resultarán violaciones de integridad referencial.

3. Borrar la tupla de EMPLEADO con IMSS = '333445555'

   - Este borrado resultará en violaciones peores de integridad referencial, porque la tupla involucrada es referenciada por tuplas de las relaciones EMPLEADO, DEPARTAMENTO, TRABAJO y DEPENDENCIA.

Se dispone de tres opciones en el caso de que un borrado ocasione una violación. La primera es rechazar el borrado. La segunda es intentar propagar el borrado borrando tuplas que referencien a la tupla que está siendo borrada. Por ejemplo, en la operación 2, el DBMS podría borrar automáticamente las dos tuplas delinquidas de TRABAJO con EIMSS = '999887777'. Una tercera opción es modificar los valores de los atributos referenciados que ocasionan la violación; cada valor es colocado en nulo o cambiado para referenciar a otra tupla válida. Nótese que, si un atributo de referencia que ocasiona una violación es parte de la llave primaria, no puede ser puesto nulo; porque, violaría la integridad de entidad.

También es posible la combinación de estas tres opciones. Por ejemplo, evitar tener la operación 3 ocasiona una violación, el DBMS puede borrar automáticamente todas las tuplas de TRABAJO y DEPENDENCIA con EIMSS = '333445555' y la tupla en

DEPARTAMENTO con GTEIMSS = '333445555'. Las tuplas de EMPLEADO con SUPERIMSS = '333445555' y la tupla DEPARTAMENTO con GTEIMSS = '333445555' puede ser borrada o tener sus valores cambiados a otros valores válidos o nulos. A pesar de que puede tener sentido borrar automáticamente las tuplas de TRABAJO y DEPENDENCIA que se refieren a una tupla EMPLEADO, puede no tener sentido borrar otras tuplas de EMPLEADO o DEPARTAMENTO. En general, cuando se especifica un argumento de integridad referencial, el DBMS debe permitir al usuario especificar cuál de las tres opciones aplica en caso de una violación del argumento.

5.3.3 La Operación Modify

Se usa para cambiar los valores de uno o más atributos en una tupla (o tuplas) de alguna relación R. Para seleccionar la tupla (o tuplas) a modificarse es necesario especificar una condición en los atributos de la relación R. Aquí están algunos ejemplos.

1. Modificar el SALARIO de la tupla EMPLEADO con IMSS = '999887777' a 28000.

   - Aceptable.

2. Modificar el DN de la tupla EMPLEADO con IMSS = '999887777' a 1.

   - Aceptable.

3. Modificar el DN de la tupla empleado con IMSS = '999887777' a 7.

   - Inaceptable, porque viola la integridad referencial.

4. Modificar el IMSS de la tupla EMPLEADO con IMSS = '999887777' a '987654321'

   - Inaceptable, porque viola los argumentos de integridad referencial y llave primaria.

La modificación de un atributo que no es ni llave primaria ni llave externa usualmente no ocasiona problemas; el DBMS necesita sólo checar para confirmar que el nuevo valor es del tipo de dato y dominio correcto. Modificar el valor de una llave primaria es similar a borrar una tupla e insertar otra en su lugar, porque usamos la llave primaria para identificar tuplas. Por lo tanto, se ponen en juego los temas discutidos anteriormente bajo Insert y Delete. Si un atributo de llave externa se modifica, el DBMS debe asegurarse de que el nuevo valor se refiera a una tupla existente en la relación referenciada.

5.4 Definición de Relaciones

Cuando va a implementarse una base de datos relacional para una aplicación compleja, los diseñadores comúnmente empiezan cuidadosamente a diseñar el esquema de la base de datos. Esto involucra decidir cuáles atributos pertenecen a cada relación, escogiendo nombres adecuados para las relaciones y sus atributos, especificando los dominios y los

tipos de datos de los distintos atributos, identificando los candidatos a llaves y eligiendo una llave primaria para cada relación, y especificando todas las llaves externas.

Cada DBMS relacional debe tener un Lenguaje de Definición de Datos (DDL) para definir los esquemas de la relación. La mayoría de los DDL se basan en el lenguaje SQL. El primer paso es asignar un nombre al esquema de la base relacional completa de tal forma que las relaciones individuales puedan asignarse a ella, usando una instrucción tal como la siguiente:

DECLARE SCHEMA COMPAÑIA;

El siguiente paso es declarar los dominios necesarios para los atributos, dando a cada dominio un nombre y tipo de dato. Declaramos los posibles dominios de los atributos de las relaciones EMPLEADO y DEPARTAMENTO de la Figura 5.7 como sigue:

DECLARE DOMAIN IMSS_PERSONA **TYPE** FIXED_CHAR(9);
DECLARE DOMAIN NOMBRE_PERSONA **TYPE** VARIABLE_CHAR(15);
DECLARE DOMAIN INICIAL_PERSONA **TYPE** ALPHABETIC_CHAR(1);
DECLARE DOMAIN FECHAS **TYPE** DATE;
DECLARE DOMAIN DIRECCIONES **TYPE** VARIABLE_CHAR(35);
DECLARE DOMAIN SEXO_PERSONA **TYPE** ENUMERATED {M, F};
DECLARE DOMAIN SALARIO_PERSONA **TYPE** MONEY;
DECLARE DOMAIN NUMEROS_DEPTO **TYPE** INTEGER_RANGE[1,10];
DECLARE DOMAIN NOMBRES_DEPTO **TYPE** VARIABLE_CHAR(20);

Ahora pueden definirse las relaciones individuales. Se necesita de construcciones para especificar el nombre de la relación, de los atributos y los dominios, llaves primarias y otras, y llaves externas. Para declarar las relaciones EMPLEADO y DEPARTAMENTO de la Figura 5.7, podemos usar los siguiente:

DECLARE RELATION EMPLEADO
FOR SCHEMA COMPAÑIA
ATTRIBUTES

	NOMBRE	**DOMAIN** NOMBRE_PERSONA;
	MINIT	**DOMAIN** INICIAL_PERSONA;
	APELLIDO	**DOMAIN** NOMBRE_PERSONA;
	IMSS	**DOMAIN** IMSS_PERSONA;
	FECHANAC	**DOMAIN** FECHAS;
	DIRECCION	**DOMAIN** DIRECCIONES;
	SEXO	**DOMAIN** SEXO_PERSONA;
	SALARIO	**DOMAIN** SALARIO_PERSONA;
	SUPERIMSS	**DOMAIN** IMSS_PERSONA;
	DN	**DOMAIN** NUMEROS_DEPTO;

```
CONSTRAINTS     PRIMARY_KEY (IMSS);
                FOREIGN_KEY (SUPERIMSS) REFERENCES EMPLOYEE;
                FOREIGN_KEY (DN) REFERENCES DEPARTMENT;

DECLARE RELATION DEPARTMENTO
FOR SCHEMA      COMPAÑIA
ATTRIBUTES      DNOMBRE   DOMAIN NOMBRES_DEPTO;
                DNUMERO   DOMAIN NUMEROS_DEPTO;
                GTEIMSS   DOMAIN IMSS_PERSONA;
                GTEFECINI DOMAIN FECHAS;

CONSTRAINTS     PRIMARY_KEY (DNUMERO),
                KEY (DNOMBRE),
                FOREIGN_KEY (GTEIMSS) REFERENCES EMPLOYEE.
```

Los ejemplos anteriores ofrecen una breve introducción a los constructores necesarios en un DDL relacional.

5.5 EL ALGEBRA RELACIONAL

Hasta ahora, hemos discutido los conceptos que el modelo relacional proporciona para definir la estructura y argumentos de una base de datos, y realizar operaciones de actualización relacional. Ahora enfocaremos nuestra atención al álgebra relacional, la cual es una colección de operaciones que se usan para manipular relaciones completas. Estas operaciones se usan, por ejemplo, para seleccionar tuplas de relaciones individuales y para combinar tuplas relacionadas de varias relaciones con el propósito de especificar un query -una requisición de recuperación- en la base de datos. El resultado de cada operación es una nueva relación, la cuál puede ser más ampliamente manipulada.

Las operaciones de álgebra relacional usualmente se dividen en dos grupos. Un grupo incluye al conjunto de operaciones de la teoría matemática de conjuntos; estas son aplicables porque cada relación se define como un conjunto de tuplas. Las operaciones de conjuntos incluyen UNION, INTERSECCION, DIFERENCIA, y PRODUCTO CARTESIANO. El otro grupo consta de operaciones desarrolladas específicamente para bases de datos relacionales; estas incluyen SELECT, PROJECT y JOIN, entre otras.

5.5.1 La Operación SELECT

Se usa para seleccionar un subconjunto de las tuplas en una relación que satisfagan una **condición de selección**. Por ejemplo, para seleccionar el subconjunto de tuplas de EMPLEADOS que trabajan en el departamento 4 con salario mayor a $30,000, podemos especificar individualmente cada una de estas dos condiciones con la operación SELECT, como sigue:

$\sigma_{DNO = 4}$(EMPLEADO)
$\sigma_{SALARIO > 30000}$(EMPLEADO)

En general, la operación SELECT se denota por

σ<condición selección>(<nombre de la relación>)
donde el símbolo σ (sigma) se usa para denotar el operador SELECT, y la condición de selección es una expresión Booleana especificada sobre los atributos de relación.

La operación resultante de la operación SELECT tiene los mismos atributos que la relación especificada en <nombre relación>. La expresión Booleana especificada en <condición selección> está hecha de un número de cláusulas de la forma:

<nombre atributo> <op comparación> <valor constante>, o
<nombre atributo> <op comparación> <nombre atributo>

donde <nombre atributo> es el nombre de un atributo de <nombre relación>, <op comparación> es normalmente uno de los operadores {=, <, <=, >, >=, <>), y <valor constante> es un valor constante del dominio de los atributos. Las cláusulas pueden conectarse arbitrariamente por los operadores Booleanos AND, OR y NOT para formar una condición de selección general. Por ejemplo, para seleccionar las tuplas de empleados quienes ya sea trabajan en el departamento 4 y ganan más de $25000 al año, o trabajan en el departamento 5 y ganan más de $30000, podemos especificar la siguiente operación SELECT:

σ(DN = 4 AND SALARIO > 25000) OR (DN = 5 AND SALARIO > 30000) (EMPLEADO)

El resultado se muestra en la Figura 5.8(a).

(a)

NOMBRE	MINIT	APELLIDO	IMSS	FECHAINI	DIRECCION	SEXO	SALARIO	SUPERIMSS	DN
Franklin	T	Wong	3334455	08-DIC-45	638, Voss	M	40000	8886655	5
Jennifer	S	Wallace	9876543	20-JUN-31	291 Berry	F	43000	8886655	4
Ramesh	K	Narayan	6668844	15-SEP-52	975 Fire	M	38000	3334455	5

APELLIDO	NOMBRE	SALARIO
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

(c)

SEXO	SALARIO
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Figura 5.8 Resultados de las operaciones SELECT y PROJECT.
(a) σ (DN = 4 AND SALARIO > 25000) OR (DN = 5 AND SALARIO > 30000) (EMPLEADO)
(b) π APELLIDO, NOMBRE, SALRIO(EMPLEADO)
(c) π SEXO, SALARIO(EMPLEADO)

Notemos que los operadores de comparación en el conjunto {=, <, <=, >, >=, <>) se aplican a atributos cuyo dominio son valores ordenados, tales como dominios numéricos o de fecha. Los dominios de cadenas de caracteres se consideran ordenados basados en el intercalamiento de la secuencia de los caracteres. Si el dominio de un atributo es un conjunto de valores desordenados, entonces sólo pueden aplicarse a ese atributo los operadores de comparasión en el conjunto {=, <>}. Un ejemplo de dominio desordenado es el dominio Color = {rojo, azul, verde, blanco, amarillo, ...} donde no se especifica ningún orden entre los diferentes colores. Algunos dominios permiten tipos adicionales de operadores de comparación; por ejemplo, un dominio de cadenas de caracteres puede permitir el operador de comparación SUBSTRING_OF.

En general, el resultado de una operación SELECT puede determinarse como sigue. La <condición de selección> se aplica independientemente a cada tupla t en la relación R especificada por <nombre de relación>. Esto se hace substituyendo cada ocurrencia de un atributo Ai en la condición de selección con su valor en la tupla t[Ai]. Si la condición evaluada es verdadera, entonces la tupla t es **seleccionada**. Todas las tuplas seleccionadas aparecen en el resultado de la operación SELECT. Las condiciones Booleanas AND, OR y NOT tienen su interpretación normal como sigue:

- (cond1 AND cond2) es verdadera si ambas (cond1) y (cond2) son verdaderas; de otra forma, es falsa.
- (cond1 OR cond2) es verdadera si (cond1) o (cond2) o ambas son verdaderas; de otra forma, es falsa.
- (NOT cond) es verdadera si cond es falsa; de otra forma, es falsa.

El operador SELECT es unario; esto es, se aplica a una sola relación. Por lo tanto, SELECT no puede utilizarse para seleccionar tuplas de más de una relación. El grado de la relación resultante de una operación SELECT es el mismo que el de la relación original R en la cual se aplicó la operación, porque tiene los mismos atributos que R. El número de tuplas en la relación resultante es siempre menor o igual al número de tuplas en la relación original R. La fracción de tuplas seleccionadas por una condición de selección es referida como la **selectividad** de la condición.

Notemos que la operación select es **conmutativa**; esto es,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Por lo tanto, puede aplicarse una secuencia de SELECT en cualquier orden. Además, podemos combinar una casacada de operaciones SELECT en una sola operación SELECT con una condición conjuntiva (AND); esto es,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(...(\sigma_{<condn>}(R))..)) = \sigma_{<cond1>\ AND\ (<cond2>AND...AND\ <condn>}(R)$$

5.5.2 La operación PROJECT

Si pensamos en una relación como una tabla, la operación SELECT selecciona algunos renglones de la tabla mientras que descarta a otros. La operación PROJECT, por otro lado, selecciona ciertas columnas de la tabla y descarta a las demás. Si nos interesamos en sólo ciertos atributos de la relación, utilizamos la operación PROJECT para "proyectar" la relación sobre estos atributos. Por ejemplo, para listar los nombres, apellidos y salarios de cada empleado, podemos utilizar la operación PROJECT como sigue:

$\pi_{\text{NOMBRE, APELLIDO, SALARIO}}(\text{EMPLEADO})$

La relación resultante se muestra en la Figura 5.8(b). La forma general de la operación PROJECT es

$\pi_{\text{<lista de atributos>}}(\text{<nombre de la relación>})$

donde π (pi) es el símbolo utilizado para representar la operación PROJECT y <lista de atributos> es una lista de atributos de la relación especificada por <nombre de la relación>. La relación resultante tiene sólo los atributos especificados en <lista de atributos> y en el mismo orden en que aparecen en la lista. Por lo tanto, su grado es igual al número de atributos en <lista de atributos>.

Si la lista de atributos incluye sólo atributos no llave de la relación, es probable que aparezcan tuplas duplicadas en el resultado. La operación PROJECT implícitamente elimina cualquier tupla duplicada, asi el resultado de la operación PROJECT es un conjunto de tuplas y por lo tanto una relación válida. Por ejemplo, consideremos la siguiente operación PROJECT:

$\pi_{\text{ SEXO,SALARIO}}(\text{EMPLEADO})$

El resultado se muestra en la Figura 5.8(c). La tupla <F, 25000> aparece una sola vez, a pesar de que la combinación de valores aparece dos veces en la relación EMPLEADO. Cada vez que dos o más tuplas idénticas aparecen cuando se aplica una operación PROJECT, sólo una se guarda en el resultado; esto se conoce como **eliminación del duplicado** y es necesario para asegurar que el resultado de la operación PROJECT sea también una relación -un conjunto de tuplas.

El número de tuplas en una relación resultante de una operación PROJECT es siempre menor o igual al número de tuplas en la relación original. Si la lista de proyección incluye una llave de la relación, la relación resultante tiene el mismo número de tuplas que la original. Por lo tanto,

$\pi_{\text{<lista1>}}(\pi_{\text{<lista2>}}(R)) = \pi_{\text{<lista1>}}(R)$

mientras que <lista2> contenga los atributos en <lista1>, de otra forma, el lado izquierdo es incorrecto. También es notorio que la conmutatividad no retiene a PROJECT.

5.5.3 Secuencias de Operaciones y Renombramiento de Atributos

Las relaciones mostradas en la Figura 5.8 no tienen nombres. En general, podemos aplicar varias operaciones de algebra relacional una después de la otra. Ya sea escribiendo las operaciones como una sola **expresión de álgebra relacional** anidando las operaciones, o aplicar una operación a la vez y crear relaciones con resultados intermedios. Por ejemplo, para recuperar el nombre, apellido y salario de todos los empleados que trabajan en el departamento 5, debemos aplicar SELECT y PROJECT. Podemos escribir una sola expresión de álgebra relacional como sigue:

$\pi_{\text{NOMBRE, APELLIDO, SALARIO}}(\sigma_{\text{DN}=5}(\text{EMPLEADO}))$

La Figura 5.9(a) muestra el resultado de esta expresión. Alternativamente, podemos mostrar explícitamente la secuencia de operaciones, dando un nombre a cada relación intermedia, como sigue:

(a)

NOMBRE	APELLIDO	SALARIO
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

(b) TEMP

NOMBRE	MINIT	APELLIDO	IMSS	FECHAINI	DIRECCION	SEXO	SALARIO	SUPERIMSS	DN
John	B	Smith	1234567	09-ENE-55	731 Fondr	M	30000	3334455	5
Franklin	T	Wong	3334455	08-DIC-45	638, Voss	M	40000	8886655	5
Ramesh	K	Narayan	6668844	15-SEP-52	975 Fire	M	38000	3334455	5
Joyce	A	English	4534534	31-JUL-62	5631 Rice	F	25000	3334445	5

R

NOMBRE	APELLIDO	SALARIO
John	Smith	30000
Franklin	Wong	40000
Ramesh	Narayan	38000
Joyce	English	25000

Figura 5.9 Resultados de expresiones de álgebra relacional. (a) $\pi_{\text{NOMBRE, APELLIDO, SALARIO}}(\sigma_{\text{DN}=5}(\text{EMPLEADO}))$. (b) La misma expresión usando relaciones intermedias y renombrando atributos.

DEP5_EMPS - $\sigma_{\text{DN}=5}(\text{EMPLEADO})$

RESULT - $\pi_{\text{NOMBRE, APELLIDO, SALARIO}}(\text{DEP5\_EMPS})$

A menudo es más fácil romper una secuencia compleja de operaciones especificando relaciones con resultado intermedio que escribir una sola expresión de álgebra relacional. También podemos usar esta técnica para **renombrar los atributos** en las relaciones intermedias resultantes. Esto puede ser útil en conexión con operaciones más complejas tales como UNION y JOIN. Para renombrar los atributos en una relación que resulta de

aplicar una operación de álgebra relacional, simplemente listamos los nuevos nombres de los atributos entre paréntesis, como en el siguiente ejemplo:

TEMP - $\sigma_{DN=5}$(EMPLEADO)
R(NOMBRE, APELLIDO, SALARIO) - $\pi_{NOMBRE, APELLIDO, SALARIO}$(TEMP)

Las dos operaciones anteriores se ilustran en la Figura 5.9(b). Si no se aplica el renombramiento, los nombres de los atributos en la relación resultante de una operación SELECT son los mismos y en el mismo orden que aquéllos en la relación original. Para una operación PROJECT sin renombramiento, la relación resultante tiene los mismos nombres de atributos que aquéllos en la lista proyección en el mismo orden.

5.5.4 Operaciones Teóricas de Conjuntos

El siguiente grupo de operaciones de álgebra relacional son las operaciones matemáticas estandar sobre conjuntos. Se aplican al modelo relacional porque una relación se define como un conjunto de tuplas que pueden usarse para procesar las tuplas en dos relaciones como conjuntos. Por ejemplo, para recuperar los números del IMSS de todos los empleados que trabajan en el departamento 5 o supervisan directamente a un empleado que trabaje en el departamento 5, podemos usar la operación UNION como sigue:

DEP5_EMPS - $\sigma_{DN=5}$(EMPLEADO)
RESULT1 - π_{IMSS}(DEP5_EMPS)
RESULT2(IMSS) - $\pi_{SUPERIMSS}$(DEP5_EMPS)
RESULT - RESULT1 $\cup$ RESULT2

La relación RESULT1 tiene los números del IMSS de todos los empleados que trabajan en el departamento 5, considerando que RESULT2 tiene los números del IMSS de todos los empleados que supervisan directamente a un empleado que trabaja en el departamento 5. La operación UNION produce las tuplas que están ya sea en RESULT1 o RESULT2 o ambas (véase Figura 5.10).

RES1	IMSS
	123456789
	333445555
	666885555
	453453453

RES2	IMSS
	333445555
	888665555

RES	IMSS
	123456789
	333445555
	666885555
	453453453
	888665555

Figura 5.10 RES ← RES1 $\cup$ RES2

Varias operaciones teóricas de conjuntos se usan para cargar los elementos de dos conjuntos de varias formas, incluyendo UNION, INTERSECCION, y DIFERENCIA. Estas operaciones son binarias; esto es, se aplican a dos conjuntos. Cuando estas

operaciones se adaptan a bases de datos relacionales, debemos asegurarnos que las operaciones pueden aplicarse a dos relaciones de tal forma que el resultado sea también una relación válida. Para lograr esto, las dos relaciones en las cuales se aplican cualquiera de las tres operaciones anteriores deben tener el mismo **tipo de tuplas;** a esta condición se le llama *compatibilidad de unión.* Dos relaciones $R(A_1, A_2, ..., A_n)$ y $S(A_1, A_2, ..., A_n)$ se dicen ser **compatibles de unión** si tienen el mismo grado n, y si el dom(Ai) = dom(Bi) para $1 \leq i \leq n$. Esto significa que las dos relaciones tengan el mismo número de atributos y que cada par de atributos correspondientes tengan el mismo dominio.

Podemos definir las tres operaciones UNION, INTERSECCION y DIFERENCIA sobre dos relaciones compatibles a unión R y S como sigue:

- UNION El resultado de esta operación se denota por $R \cup S$, es una relación que incluye todas las tuplas que están en R o S o en ambas R y S. Las tuplas duplicadas se eliminan.

- INTERSECCION El resultado de esta operación se denota por $R \cap S$, es una relación que incluye a todas las tuplas que están en R y en S.

- DIFERENCIA El resultado de esta operación, denotada por R - S, es una relación que incluye todas las tuplas que están en R pero no en S.

Adoptaremos la convensión de que la relación resultante tiene los mismos nombres de atributos que la *primera* relación R. La Figura 5.11 ilustra las tres operaciones. Las relaciones ESTUDIANTE e INSTRUCTOR en la Figura 5.11(a) son compatibles a unión, y sus tuplas representan los nombres de estudiantes e instructores, respectivamente. El resultado de la operación UNION (Figura 5.11(b)) muestra los nombres de todos los estudiantes e instructores. Nótese que las tuplas duplicadas aparecen sólo una vez en el resultado. El resultado de la operación INTERSECCION (Figura 5.11(c)) incluye sólo a aquéllos que están en estudiantes e instructores. Nótese que UNION e INTERSECCION son operaciones *conmutativas;* esto es

$$R \cup S = S \cup R \text{ y } R \cap S = S \cap R$$

Cualquier operación puede aplicarse a cualquier miembro de la relación, y ambas son operaciones *asociativas;* esto es

$$R \cup (S \cup T) = (R \cup S) \cup T, \text{ y}$$
$$(R \cap S) \cap T = R \cap (S \cap T)$$

La operación diferencia no es *conmutativa;* esto es, en general,

$$R - S \neq S - R$$

(a)

ESTUDIANTE	NOM	APE
	Susan	Yao
	Ramesh	Shah
	Johnny	Kohler
	Barbara	Jones
	Amy	Ford
	Jimmy	Wang
	Ernest	Gilbert

INSTRUCTOR	NOMBRE	APELLIDO
	John	Smith
	Ricardo	Browne
	Susan	Yao
	Francis	Johnson
	Ramesh	Shah

(b)

NOM	APE
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

NOM	APE
Susan	Yao
Ramesh	Shah

(d)

NOM	APE
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

NOM	APE
John	Smith
Ricardo	Browne
Francis	Johnson

Figura 5.11 El conjunto de operaciones UNION, INTERSECCION y DIFERENCIA. (a) Dos relaciones compatibles de unión. (b) ESTUDIANTE $\cup$ INSTRUCTOR. (c) ESTUDIANTE $\cap$ INSTRUCTOR. (d) ESTUDIANTE - INSTRUCTOR. (e) INSTRUCTOR - ESTUDIANTE.

La Figura 5.11(d) muestra los nombres de los estudiantes que no son instructores, y la Figura 5.11(e) muestra los nombres de los instructores que no son estudiantes.

Después discutiremos el PRODUCTO CARTESIANO, denotado por **X**. Esta es también una operación de conjunto binaria, pero la relación sobre la cual se aplica no tiene que ser compatible con la unión. Esta operación, también conocida como PRODUCTO CRUZ,o UNION CRUZ, se usa para combinar tuplas de dos relaciones de tal forma que las tuplas relacionadas puedan identificarse. En general, el resultado de $R(A_1, A_2, ..., A_n)$ **X** $S(B_1, B_2, ..., B_m)$ es una relación Q con n+m atributos $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, en ese orden. La relación resultante Q tiene una tupla para cada combinación de tuplas -una de R y una de S. Por lo tanto, si R tiene nR tuplas y S tiene nS tuplas, entonces R **X** S tendrá nR * nS tuplas. Para ilustrar el uso del PRODUCTO CARTESIANO, supóngase que deseamos recuperar para cada empleado femenino una lista de los nombres de sus dependientes; podemos lograr esto como sigue

EMPS_FEMENINO - $\sigma_{SEXO = 'F'}$(EMPLEADO)
EMPNOMBRES - $\pi_{NOMBRE, APELLIDO, IMSS}$(EMPS_FEMENINO)
EMP_DEPENDIENTES - EMPNOMBRES X DEPENDIENTE
DEPENDIENTES_ACTUAL - $\sigma_{IMSS = EIMSS}$(EMP_DEPENDIENTES)
RESULT - $\pi_{NOMBRE, APELLIDO, NOMBRE\_DEPENDIENTE}$(DEPENDIENTES_ACTUAL)

Las relaciones resultantes de la secuencia de operaciones anteriores se muestran en la Figura 5.12. La relación EMP_DEPENDIENTES, cada tupla de EMPNOMBRES se combina con cada tupla de DEPENDIENTE, dando un resultado que no es muy significativo.

El PRODUCTO CARTESIANO crea tuplas con los atributos combinados de dos relaciones. Podemos entonces SELECCIONAR sólo las tuplas relacionadas de las dos relaciones especificando una condición de selección apropiada. Debido a que esta secuencia de PRODUCTO CARTESIANO seguida por SELECT se usa muy comúnmente para identificar y seleccionar tuplas relacionadas de dos relaciones, fue creada una operación especial llamada JOIN, para especificar esta secuencia como una sola operación. El PRODUCTO CARTESIANO es raramente utilizado como una operación significativa por sí misma.

5.5.5 La Operación Join

Denotada por ◊, se usa para combinar tuplas relacionadas de dos relaciones en tuplas sencillas. Esta operación es muy importante para cualquier base de datos relacional con más de una sola relación, debido a que nos permite procesar relaciones entre relaciones. Para ilustrar JOIN, supóngase que deseamos recuperar el *nombre del gerente de cada departamento*. Para obtener el nombre del gerente, necesitamos combinar cada tupla de departamento con la de empleado cuyo IMSS coincida con el valor de GTEIMSS en la tupla departamento.

El resultado de JOIN es una relación Q con n+m atributos $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$ en ese orden; Q tiene una tupla para cada combinación de tuplas -una de R y una de S- *cada que la combinación satisface la condición join*. Esta es la principal diferencia entre el PRODUCTO CARTESIANO y JOIN: en JOIN, sólo aparecen en el resultado las condiciones de tuplas que satisfacen la condición join; mientras que en el PRODUCTO CARTESIANO se incluyen todas las combinaciones de tuplas. La condición join se especifica sobre los *atributos* de las dos relaciones R y S y se evalúa para cada combinación de tuplas. Cada combinación de tuplas para la cual la condición join evalúa verdadera a sus valores de atributo se incluye en la relación resultante Q como una sóla tupla.

Una condición join es de la forma:

<condición> AND <condición> AND ... AND <condición>

donde cada condición es de la forma Ai θ Bj, Ai es un atributo de R, Bj es un atributo de S, Ai y Bj tienen el mismo dominio, y θ es uno de los operadores comparadores {=, <, <=, >, >=, ≠}. Una operación JOIN con condición general se le llama **JOIN TETA**. Las tuplas cuyos atributos join son nulos *no aparecen* en el resultado.

El JOIN más común involucra condiciones con comparasiones de igualdad únicamente. Tal JOIN, donde el único operador de comparasión es el =, se le llama **EQUIJOIN**. Notemos que en el resultado de un EQUIJOIN siempre tenemos uno o más pares de atributos que tienen *valores idénticos* en cada tupla. Debido a que uno de cada par de atributos con valores idénticos es superfluo, fue creada la operación **JOIN NATURAL**, para deshacerse del segundo atributo en una condición equijoin. Denotamos al join natural por *. Es básicamente un equijoin seguido por la eliminación de atributos superfluos. La definición estandar de JOIN NATURAL requiere que los dos atributos join (o cada par de atributos join) tienen el mismo nombre. Si no es el caso, se aplica primero una operación de renombrado.

Si los atributos sobre los cuales se especifica el join natural *tienen los mismos nombres en ambas relaciones*, es innecesario el renombrado.

En general, el JOIN NATURAL se realiza igualando *todos* los pares de atributos que tienen el mismo nombre en las dos relaciones. Puede haber una lista de atributos join de cada relación, y cada par correspondiente debe tener el mismo nombre.

Una definición más general para el JOIN NATURAL es

Q ← R (<lista1>,<lista2>) S

En este caso, <lista1> especifica una lista de i atributos de R, y <lista2> especifica una lista de atributos de S. Las listas se usan para formar condiciones de igualdad de comparasión entre pares correspondientes de atributos; las condiciones son entonces ANDeadas juntas. Solo la lista correspondiente a atributos de la primera relación R -<lista1>- se guardan en el resultado Q.

Notemos que, si ninguna combinación de tuplas satisface la condición join, el resultado es una relación vacía con cero tuplas. En general, si R tiene nR tuplas y S tiene nS tuplas, el resultado de una operación JOIN R ◊ <lista de condiciones> S tendrá entre 0 y nR * nS tuplas. El tamaño esperado del resultado de join dividido por el máximo tamaño nR*nS lleva a un radio llamado **selectividad join**, el cual es una propiedad de cada condición join. Si no hay <condición join> que satisfacer, todas las combinaciones de tuplas califican y se vuelve un PRODUCTO CARTESIANO, también llamado JOIN CRUZ.

5.5.6 Juego Completo de Operaciones de Algebra Relacional

Se ha demostrado que el conjunto de operaciones de álgebra relacional (σ π ∪ ∩ - X) es un juego **completo**; esto es, cualquier otra operación se puede expresar como una

secuencia de operaciones de este conjunto. Por ejemplo, la INTERSECCION puede expresarse usando UNION y DIFERENCIA:

$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$

A pesar de que, estrictamente hablando no se requiere la intersección, es conveniente expresar esta expresión compleja cada que se desee especificarla. Como otro ejemplo, una operación JOIN puede especificrase como un PRODUCTO CARTESIANO seguido por una operación SELECT.

$R \Diamond_{\text{<condición>}} S \equiv \sigma_{\text{<condición>}} (RXS)$

5.5.7 La Operación DIVISION

En general, la operación DIVISION se aplica a dos relaciones $R(Z) \div S(X)$, donde $X \subseteq Z$. Supongamos $Y = Z - X$; esto es, supongamos que Y es el conjunto de atributos de R que no son atributos de S. El resultado de DIVISION es una relación T(Y) que incluye una tupla t si una tupla tR cuyo tR[Y] = t aparece en R, con tR[X] = ts *para cada tupla* ts en S. Esto significa que, para que una tupla t aparezca en el resultado T de la DIVISION, los valores en t deben aparecer en R en combinación con *cada* tupla en S.

El operador DIVISION puede expresarse como una secuencia de operaciones π, X y - como sigue:

$T_1 \leftarrow \pi_Y(R)$
$T_2 \leftarrow \pi_Y((S \times T_1) - R)$
$T \leftarrow T_1 - T_2$

5.6 Operaciones Relacionales Adicionales

Algunas requisiciones comunes no pueden llevarse a cabo con las operaciones estandar descritas anteriormente. La mayoria de los lenguajes query para DBMS relacionales incluyen la capacidad de efectuar estas requisiciones. Estas operaciones fortalecen el poder expresivo del álgebra relacional.

5.6.1 Funciones Agregadas

El primer tipo de requisición que no puede expresarse en álgebra relacional es especificar **funciones matemáticas agregadas** sobre colecciones de valores de la base de datos. Las funciones comunes aplicadas a colecciones de valores numéricos incluyen SUM, AVERAGE, MAX y MIN. La función COUNT se usa para contar tuplas. Cada una de estas funciones puede aplicarse a una colección de tuplas.

Otro tipo común de requisición involucra agrupar las tuplas en una relación por el valor de alguno de sus atributos y luego aplicar una función agregada independientemente a cada grupo.

Se puede definir una operación FUNCION, usando el símbolo $\mathfrak{I}$ (f manuscrita), para especificar este tipo de requisiciones como sigue:

<atributos de agrupamiento> $\mathfrak{I}$ <lista de funciones> (<nombre de la relación>)

donde <atributos de agrupamiento> es una lista de los atributos de la relación especificada en <nombre de la relación>, y <lista de funciones> es una lista de parejas (<función> <atributo>). En cada par, <función> es una de las funciones permitidas y <atributo> es un atributo de la relación especificada por <nombre de la relación>. La relación resultante tiene los atributos de agrupamiento más un atributo por cada elemento en la lista de funciones.

(a)

R	DNO	NO_DE_EMPLEADOS	PROM_SAL
	5	4	33250
	4	3	31000
	1	1	55000

(b)

DNO	CTA_IMSS	PROM_SAL
5	4	33250
4	3	31000
1	1	55000

(c)

CTA_IMSS	PROM_SAL
8	35125

Figura 5.16 La operación FUNCION.
(a) R(DNO, NO_DE_EMPLEADOS, PROM_SAL)←DNO $\mathfrak{I}$ COUNT IMSS, AVERAGE SALARIO (EMPLEADO)
(b) DNO $\mathfrak{I}$ COUNT IMSS, AVERAGE SALARIO (EMPLEADO)
(c) $\mathfrak{I}$ COUNT IMSS, AVERAOE SALARIO (EMPLEADO)

Si no se especifica la lista de atributos, entonces los atributos de la relación resultante que corresponden a la lista de funciones será la concatenación del nombre de la función con el nombre del atributo sobre el cual se aplica la función en la forma <función>_<atributo>.

Si no se especifican atributos de agrupamiento, la función se aplica a los valores de los atributos de *todas las tuplas* en la relación, así la relación resultante tiene *una sola tupla única.*

5.6.2 Operaciones Recursivas de Conclusión

Otro tipo de operación que, en general, no puede especificarse en el álgebra relacional es una **conclusión recursiva**. Esta operación se aplica a una **relación recursiva** entre tuplas del mismo tipo.

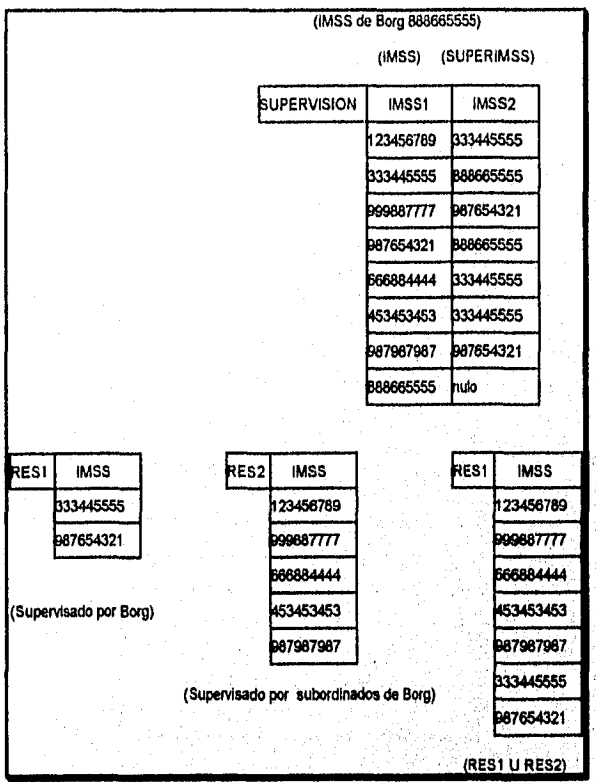


Figura 5.17 Recursión a dos niveles.

5.7 Ejemplos de Queries en el Algebra Relacional

Ahora pondremos ejemplos adicionales para ilustrar el uso de operaciones del álgebra relacional.

QUERY 1
Recuperar el nombre y dirección de todos los empleados que trabajan para "Investigación".

DEPTO_INVESTIGACION←σDNOMBRE = 'INVESTIGACION'(DEPTO)
EMPS_DEPTO_INVESTIGACION←(INVESTIGACION_DEPTO ◊ DNUMERO = DNO EMPLEADO)
RES ← π NOMBRE, APELLIDO, DIRECCION(EMPS_DEPTO_INVESTIGACION)

QUERY 2

Encontrar los nombres de todos los empleados que trabajan en *todos* los proyectos controlados por el departamento número 5.

PROYS_DEP5(PNO) ← πNUMPRO(σDNUM=5(PROYECTO))
EMP_PROY(IMSS, PNO) ← π EIMSS, PNO(TRABAJA_EN)
RES_EMP_IMSS ← EMP_PROY ÷ PROYS_DEP5

RES ← π APELLIDO, NOMBRE(RES_EMP_IMSS * EMPLEADO)

5.8 Diseño de Bases de Datos Relacionales Usando Mapeo ER a Relacional

En esta sección, mostramos cómo un esquema relacional puede derivarse de un esquema conceptual desarrollado usando el modelo Entidad-Relación (ER). Muchas herramientas CASE (Computer Aided Software Engineering) están basadas en el modelo ER y sus variantes. Estas herramientas computarizadas se usan interactivamente por diseñadores de bases de datos para desarrollar un esquema ER para su aplicación. Muchas herramientas usan diagramas ER o variantes para desarrollar el esquema gráficamente, y luego automáticemente convertirlo a un esquema relacional en el DDL de un DBMS en específico.

5.8.1 Algoritmo de Mapeo ER a Relacional

Ahora describiremos informalmente los pasos del algoritmo de mapeo ER a relacional.

PASO 1: Para cada entidad regular tipo E en el esquema ER, crear una relación R que incluya todos los atributos de E. Incluir sólo los atributos de componente sencillo de un atributo compuesto. Elegir uno de los atributos llave de E como la llave primaria de R. Si la llave elegida de E es una llave compuesta, el conjunto de atributos sencillos que la integran formarán la llave primaria de R.

PASO 2: Para cada tipo de entidad débil W en el esquema ER con entidad propia tipo E, crear una relación R, e incluir todos los atributos sencillos (o componentes sencillos de atributos compuestos) de W como atributos de R. Además, incluir como llave externa a los atributos de la llave primaria de la relación que correspondan al tipo de entidad propia de W. La llave primaria de R es la combinación de llaves primarias de la propiedad y llave parcial de la entidad débil W, si existen.

PASO 3: Para cada relación binaria 1:1 tipo R en el esquema ER, identificar las relaciones S y T que correspondan a tipos de entidad participantes en R. Elegir una de las relaciones -digamos S- e incluir como llave externa de S a la llave primaria de T. Es mejor elegir un tipo de entidad con participación total en R en el rol de S. Incluir todos los atributos sencillos (o componentes sencillos de atributos compuestos) del tipo de relación 1:1 R como atributos de S.

PASO 4: Para cada relación regular (no débil) binaria 1:N tipo R, identificar la relación S que representa el tipo de entidad participante del tipo de relación en el *lado-N*. Incluir como llave externa de S a la llave primaria de T que representa al otro tipo de entidad participante en R; esto es porque cada instancia de entidad en el lado-N se relaciona con la mayoría de las instancias de entidad del tipo de relación del lado-1. Incluir todos los atributos sencillos (o componentes sencillos de atributos compuestos) del tipo de relación 1:N como atributos de S.

PASO 5: Para cada relación binaria M:N tipo R, crear una nueva relación S para representar a R. Incluir como atributos de llave externa en S las llaves primarias de las relaciones que representan los tipos de entidad participante; su combinación formará la llave primaria de S. Notemos que no se puede representar una relación tipo M:N por un solo atributo de llave externa en una de las relaciones participantes -debido al radio de cardinalidad M:N.

Se pueden mapear relaciones 1:1 o 1:N de manera similar a relaciones M:N. Esta alternativa es particularmente útil cuando existen pocas instancias de relación, y así evitar valores nulos en llaves externas. En este caso, la llave primaria de la relación "relación" será la llave externa de *sólo una* de las relaciones "entidad" participantes. Para una relación 1:N, esta será la entidad relación del lado-N. Para una relación 1:1, se elige la entidad relación con participación total (si existe).

PASO 6: Para cada aributo multivaluado A, crear una nueva relación R que incluya un atributo correspondiente a A más el atributo llave primaria K (como llave externa en R) de la relación que representa el tipo de entidad o relación que tiene A como atributo. La llave primaria de R es la combinación de A y K. Si el atributo multivaluado es compuesto, incluiremos sus componentes sencillos.

PASO 7: Para cada relación n-aria tipo R, n > 2, crear una nueva relación S para representar a R. Incluir como atributos de llave externa en S las llaves primarias de las relaciones que representan los tipos de entidad participante. También incluir cualquier atributo de la relación tipo n-aria (o compnentes sencillos de atributos compuestos) como atributos de S. La llave primaria de S es usualmente una combinación de todas las llaves externas que referencian las relaciones representando los tipos de entidades participantes. Sin embargo, si el argumento de participación (min, max) de uno de los tipos de entidad E participantes en R tiene max = 1, entonces la llave primaria de S puede ser el atributo sencillo llave externa que referencia la relación E' correspondiente a E; esto es porque, en este caso, cada entidad e en E participará en al menos una instancia de relación de R y por

tanto únicamente puede identificar la instancia de relación. Esto concluye el procedimiento de mapeo.

El principal punto en un esquema de relación, cuando se compara a un esquema ER, es que los tipos de relación no se representan explícitamente; en su lugar, se representan teniendo dos atributos A y B, uno una llave primaria y el otro una llave externa -sobre el mismo dominio- incluidos en dos relaciones S y T. Dos tuplas en S y T se relacionan cuando tienen el mismo valor para A y B. Usando el EQUIJOIN sobre S.A y T.B podemos combinar todos los pares de tuplas relacionadas de S y T y materializar la relación. Cuando se involucra una relación binaria tipo 1:1 o 1:N, usualmente es necesaria una operación join sencilla. Para una relación binaria tipo M:N, son necesarias dos operaciones join, mientras que para los tipos de relaciones n-arias, son necesarios n joins.

La Tabla 5.1 muestra los pares de atributos que se usan en operaciones EQUIJOIN para materializar cada tipo de relación en el esquema COMPAÑIA de la Figura 3.2. Para materializar la relación 1:N tipo TRABAJA_EN, aplicamos EQUIJOIN a las relaciones EMPLEADO y DEPARTAMENTO sobre los atributos DNO de EMPLEADO y DNUM de DEPARTAMENTO. De acuerdo a la base de datos de la Figura 5.6, los empleados Smith, Wong, Narayan y English trabajan para el departamento 5 (Investigación); Zelaya, Wallace y Jablar trabajan para el departamento 4 (Administración); y Borg trabaja para el departamento 1 (Oficinas Centrales).

Otro punto sobresaliente en el esquema relacional es que creamos una relación separada para *cada* atributo multivaluado. Para una entidad en particular con un conjunto de valores para el atributo multivaluado, el valor del atributo llave de la entidad se repite una vez para cada valor del atributo multivaluado en una tupla separada. Esto se debe a que el modelo básico relacional *no* permite valores múltiples (o conjuntos de valores) para un atributo en una sola tupla. Por ejemplo, debido a que el departamento 5 tiene tres localidades, existen tres tuplas en la relación DEP_LOC de la Figura 5.6; cada tupla especifica una de las localidades. Para relacionar los valores de los atributos multivaluados a los valores de otros atributos de una entidad o instancia de relación es necesario un equijoin, pero seguirá consiguiendo múltiples tuplas. El álgebra relacional no tiene operación NEST o COMPRESS que produciría de la relación DEP_LOC de la Figura 5.6 un conjunto de tuplas de la forma {<1, Houston>}, {<4, Stafford>}, {<5, Bellaire, Sugarland, Houston>}. Esta es una seria limitante de la versión actual normalizada o "plana" del modelo relacional. Los lenguajes relacionales SQL, QUEL, y QBE no tienen opciones para el manejo de tales conjuntos de valores dentro de tuplas. En este caso, los modelos orientados a objetos, de red y jerárquico tienen mayores facilidades que el modelo relacional. El modelo relacional anidado intenta remediar esto.

5.8.2 Resúmen del Mapeo para Construcción del Modelo y Argumentos

Ahora globalizaremos las correspondencias entre los modelos ER y constructores y argumentos del modelo relacional.

Tabla 5.1 Condiciones Join para Materializar los tipos de Relación del Esquema ER COMPAÑIA

Relación ER	Relaciones Participantes	Condición Join
TRABAJA_PARA	EMPLEADO.	EMPLEADO.DNO =
	DEPARTAMENTO	DEPTO.DNUMERO
MANEJA	EMPLEADO.	EMPLEADO.IMSS=
	DEPARTAMENTO	DEPTO.GTEIMSS
SUPERVISION	EMPLEADO(E)	EMPLEADO(E).SUPERIMSS=
	EMLPLEADO(S)	EMPLEADO(E).IMSS
TRABAJA_EN	EMPLEADO.	
	TRABAJA_EN.	
	PROYECTO	
CONTROLA	DEPARTAMENTO.	DEPTO.DNUMERO =
	PROYECTO	PROY.DNUM
DEPENDIENTE_DE	EMPLEADO.	EMPLEADO.IMSS = DEPENDIENTE.EIMSS
	DEPENDIENTE	

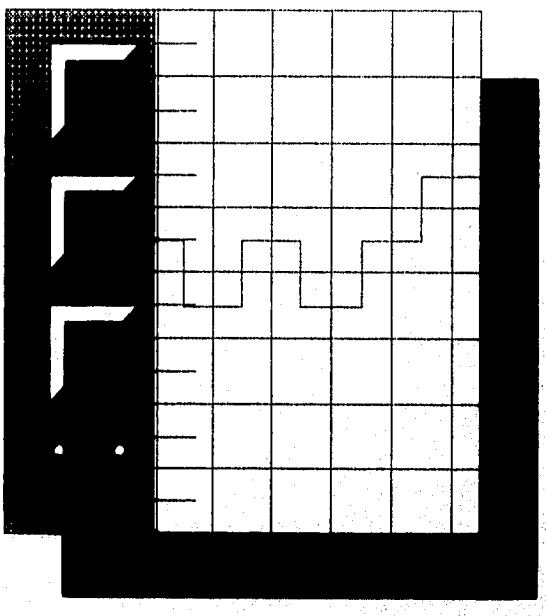
Tabla 5.2 Correspondencia entre Modelos ER y Relacional

ModeloE.R	Modelo Relacional
tipo de entidad	"entidad" relación
tipo de relación 1:1 o 1:N	llave externa (o relación "relación")
tipo de relación M:N	relación "relación" y dos llaves externas
tipo de relación n-aria	relación "relación" y n llaves externas
atributo simple	atributo
atributo compuesto	conjunto de componentes sencillos de atributos
atributo multivaluado	relación y llave externa
conjunto de valnres	dominio
atributo llave	llave primaria (o secundaria)

Tabla 5.3 Operaciones del Algebra Relacional

Operación	Propósito	Notación
SELECT	Selecciona todas las tuplas que satisfacen la condición de selección de una relación R.	σ<condición de selección> (R)
PROJECT	Produce una nueva relación con sola algunos de los atributos de R, y elimina tuplas duplicadas.	π<lista de atributos> (R)
JOIN TETA	Produce todas las combinaciones de tuplas de R1 y R2 que satisfacen la condición Join.	R1◊ <condición Join>R2
EQUIJOIN	Produce todas las combinaciones de tuplas de R1 y R2 que satisfacen una condición Join con sólo comparasiones de igualdad.	R1◊ <condición Join>R2, o R1◊(<atributos Join 1>), (<atributos join 2>) R2
JOIN NATURAL	Igual que el EQUIJOIN, excepto que los atributos Join de R2 no se incluyen en la relación resultante; si los atributos Join tienen los mismos nombres, no deben especificarse del todo.	R1*<condición Join> R2, o R1*(<atributos join 1>), (<atributos join 2>) R2 R1 * R2
UNION	Produce una relación que incluye todas las tuplas en R1 o R2 o en ambas. R1 y R2 deben ser compatibles de unión.	R1 ∪ R2
INTERSEC-CION	Produce una relación que incluye todas las tuplas en ambas R1 y R2. R1 y R2 deben ser compatibles de unión.	R1 ∩ R2
DIFERENC	Produce una relación que incluye todas las tuplas en R1 y no están en R2	R1 - R2
PRODUCTO CARTESIAN	Produce una relación que tiene los atributos de R1 y R2 e incluye como tuplas todas las posibles combinaciones de tuplas de R1 y R2.	R1 X R2
DIVISION	Produce una relación R(X) que incluye todas las tuplas t[X] en R1[Z] que aparecen en R1 en combinación con cada tupla de R2(Y), donde Z=X ∪ Y.	

CAPITULO 6



Sistema Administrador de Bases de Datos Relacional - DB2

6.1 Introducción al Sistema de Bases de Datos Relacional

Después que Codd introdujo el modelo relacional en 1970, hubo una conmoción de experimentación con ideas relacionales. Dio inicio una mayor investigación y esfuerzo de desarrollo en el Centro de Investigaciones de IBM en San José. Lo cual llevó al anuncio de dos productos DBMS relacionales comerciales por IBM en los 80s: SQL/DS para medios ambientes DOS/VSE (sistema operativo en disco/almacenamiento virtual extendido) y para VM/CMS (máquina virtual/sistema de monitoreo conversacional) introducidos en 1981; y DB2 para el sistema operativo MVS, introdicido en 1983. Otro gran DBMS relacional es INGRES, desarrollado en la Universidad de California, Berkeley, a principios de los 70's y comercializado por Relational Technology, Inc, a finales de los 70's. Ahora INGRES es un RDBMS comercial comercializado por INGRES, Inc.; Sybase de Sybase, Inc.; RDB de Digital Equipment Corp.; INFORMIX de Informix, Inc.; y UNIFY de Unify, Inc. En este capítulo discutiremos las características de DB2 de IBM para conocer lo que se ofrece típicamente en un producto RDBMS comercial. Los demás sistemas tienen arquietcturas similares, soporte de lenguajes, y herramientas para desarrollo de aplicaciones.

Además de los RDBMS mencionados anteriormente, en los 80's aparecieron muchas implementaciones del modelo relacional en las computadoras personales (PC). Se incluyen a RIM, RBASE 5000, PARADOX, OS/2 Database Manager, dBASE IV, XDB, WATCOM SQL, SQL Server, y más recientemente ACCESS. Inicialmente fueron sistemas monousuario, pero más recientemente, han empezado a ofrecer la arquitectura cliente/servidor y se están volviendo compatibles con el estandar de *Conectividad de Bases de Datos Abiertas* de Microsoft (ODBC). Este estandar permite el uso de muchas herramientas front-end con estos sistemas.

La palabra *relacional* también se usa un poco inadecuadamente por varios vendedores para referirse a sus productos como un gancho mercadológico. Para calificar como genuino a un DBMS relacional, un sistema debe tener al menos las siguientes propiedades:

1. Debe almacenar los datos como relaciones de tal manera que cada columna sea identificada independientemente por el nombre de su columna y el ordenamiento de renglones como inmaterial.

2. Las operaciones disponibles al usuario, así como las utilizadas internamente por el sistema, deben ser verdaderas operaciones relacionales; esto es, deben ser capaces de generar nuevas relaciones a partir de las anteriores.

3. El sistema debe soportar al menos una variante de la operación JOIN.

A pesar de que esta lista podría ser más grande, proponemos este criterio como un conjunto muy mínimo para probar si un sistema es relacional. Es fácil ver que muchos, así llamados, DBMS relacionales no satisfacen estos criterios.

Ahora describiremos DB2, el cual es en la actualidad uno de los sistemas relacionales más ampliamente usados en mainframes.

6.2 Arquitectura Básica de DB2

El nombre DB2 es una abreviatura de Database 2. Es un DBMS relacional de IBM para el sistema operativo MVS. Se pondrá énfasis en transmitir la complejidad de un RDBMS como DB2 y su variedad de características, en lugar de enumerar exactamente las características y facilidades de alguna versión específica de DB2.

DB2 coopera con ("se complementa con" en terminología del producto) cualquiera de los tres sistemas operativos MVS: CICS, TSO e IMS. Estos sistemas cooperan con los recursos de DB2 para proporcionar comunicaciones y manejo de transacciones. La Figura 6.1 muestra las relaciones entre varios componentes de DB2. DB2 permite acceso concurrente a bases de datos por IMS/VS-DC (Sistema de Administración de Información /Almacenamiento Virtual-Comunicaciones de Datos), CICS (Sistema de control de Información al Cliente), y TSO (Opción de Tiempo Compartido), interactivos y batch. CICS es un sistema de monitoreo de teleproceso -un producto popular de IBM usado por muchas industrias para el procesamiento de transacciones de negocios. IMS/DC es un medio ambiente de comunicaciones de datos que soporta bases jerárquicas IMS. TSO es un producto para el medio ambiente de tiempo compartido de IBM. El recurso "call attach" CAF permite que una aplicación interactúe con una base de datos DB2 sin la ayuda de estos monitores. Las bases de DB2 pueden accesarse por programas de aplicación escritos en COBOL, PL/1, FORTRAN, C, PROLOG, o lenguaje ensamblador de IBM.

Los siguientes puntos describen el uso de los diferentes subsistemas mostrados en la Figura 6.1:

1. Una aplicación DB2 consta de programas escritos en los lenguajes antes mencionados que corren bajo el control de *uno sólo* de los tres subsistemas -IMS, CICS, o TSO.

2. Las mismas bases de datos DB2 pueden compartirse por aplicaciones IMS, CICS, y TSO. CSP (Cross System Product) permite que una aplicación desarrollada bajo TSO corra bajo CICS y viceversa.

3. Existen dos facilidades principales en línea con DB2: QMF (Facilidad de Manejo de Queries) bajo CICS o CAF, y DB2 interactivo (DB2I) bajo TSO. DB2I viene con DB2; permite a usuarios profesionales ingresar a SQL interactivamente a través de una interfaz llamada SPUFI y les ayuda a preparar programas y utilerías para ejecución.

4. Además de las bases DB2, las bases IMS son también accesibles desde una aplicación DB2 bajo el medio ambientes IMS o CICS, pero no bajo TSO. La misma aplicación TSO puede ejecutarse como archivo batch o en línea, dirigiendo la I/O del programa a archivos o usando terminales de I/O.

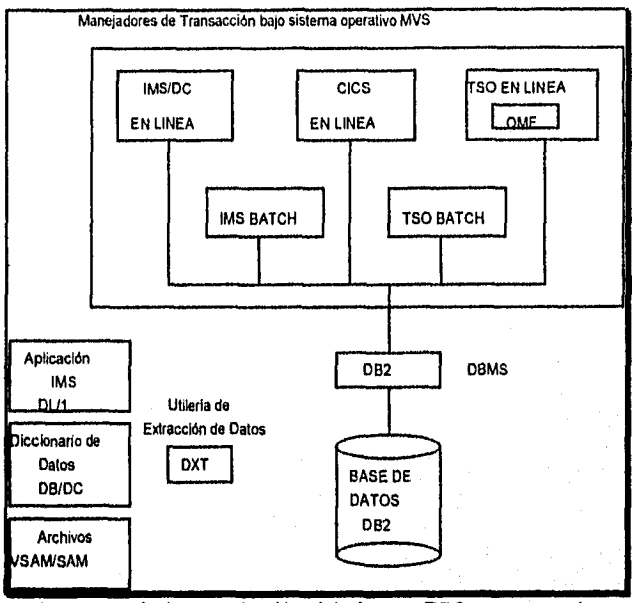


Figura 6.1 Panorama de la organización del sistema DB2, con una vista parcial de características de soporte.

Como se mostró en la Figura 6.1, otras dos facilidades -QMF y DXT- juegan papeles importantes en el uso de DB2.

QMF (Facilidad de Manejo de Queries). QMF es un producto independiente de DB2 que actúa como lenguaje query y generador de reportes. Simplemente corre como una aplicación TSO en línea. Soporta queries ad hoc de usuarios finales no técnicos en SQL o QBE y despliega los resultados como reportes formateados. Puede accesar bases de datos DB2 y SQL/DS. La salida de QMF puede dirigirse a otras utilerías para dibujar gráficas de barras (con la Utilería Interactiva de Gráficas) y otros despliegues gráficos de datos (con el Manejador de Despliegue Gráfico de Datos). Los usuarios construyen formas interactivamente para controlar el despliegue de resultados del query.

DXT (Extracción de Datos). Es una utilería que extrae datos de bases IMS o VSAM (Método de Acceso de Almacenamiento Virtual) o SAM (Método de Acceso Secuencial) y los convierte en archivos secuenciales. DXT puede especificar requisiciones de extracción interactivamente o como tareas batch. Este archivo secuencial está en un formato aconsejable para carga directa a una base de datos DB2. DXT es un producto independiente de IBM.

6.2.1 La Familia DB2 y DB2/2

SQL/DS es el primer DBMS relacional que pertenece a la "familia DB2". Las facilidades de definición y manipulación de datos de los dos sistemas son esencialmente los mismos, con algunas pequeñas diferencias sintácticas. Ambos usan a SQL como lenguaje query interactivo, así como un lenguaje de programación de bases de datos, contenido en un lenguaje huésped. El sistema DB2 inicialmente usó el código SQL/DS para las partes superiores del sistema (por ejemplo, la optimización y procesamiento de queries). Las partes "inferiores" de DB2 fueron construidas desde cero. Las facilidades QMF y DXT pueden usarse por DB2 y SQL/DS. Sin embargo, el almacenamiento físico de datos difiere en SQL/DS y DB2. Por ejemplo, conceptos tales como espacios de tabla no tienen analogía en SQL/DS. DB2 también soporta técnicas especializadas para el manejo de grandes bases de datos y pesadas cargas de trabajo que son únicas del sistema operativo MVS. Mientras que DB2 proporciona una facilidad de SQL interactivo para usuarios finales vía DB2I, SQL/DS es capáz de proporcionar algunas facilidades a los usuarios finales vía su componente ISQL (SQL Interactivo). El DB2/2 de IBM es el último de los RDBMS. Corre sobre OS/2 y fue introducido en 1988 como OS/2 Manejador de Bases de Datos Edición Extendida. Aclaremos que el DBMS de AS/400 es diferente de los RDBMS's en la familia DB2. La mayor parte de su funcionalidad reside en el hardware propio.

6.2.2 Organización de Datos y Procesos en DB2

Veamos primero como se perciben las bases de datos de DB2 por usuarios y programas de aplicación. Todo dato se visualiza como relaciones o "tablas", un término legítimo de DB2. Las tablas son de dos tipos: *tablas base*, las cuales físicamente existen como datos almacenados; y *vistas*, las cuales son tablas virtuales sin una identidad física separada en almacenamiento. Una tabla base consta de uno o más archivos VSAM.

Procesamiento de Aplicación. La Figura 6.2 muestra la preparación de una aplicación DB2 en SQL contenido, de manera simplificada. Indica la secuencia de procesos a través de los cuales debe pasar una aplicación para accesar a una base de datos DB2. Los mayores componentes de la aplicación SQL son el precompilador, el enlazador, el supervisor run-time, y el administrador de datos almacenados. Brevemente, realizan las siguientes funciones:

- **Precompilador:** La tarea del precompilador es procesar instrucciones internas SQL en un programa de lenguaje huésped. El precompilador genera dos tipos de salida: el programa fuente original, con el SQL interno reemplazado por LLAMADAS; y módulos de requisición de bases de datos (DBRMs), los cuales son colecciones de instrucciones SQL en un parseo de entrada en forma de árbol al proceso de enlace.

- **Enlace:** Este componente acomoda ambos tipos de requisiciones SQL: aquéllos que provienen de programas de aplicación para ejecutarse una y otra vez, y queries ad hoc que se ejecutan una sola vez. Para la primera categoría, sigue un análisis detallado y

optimización del query, uno o más DBRMs se compilan *una sola vez* a un plan de aplicación. El costo de este enlace se amortiza a través de ejecuciones repetidas del programa y se encuentra bien justificado. El proceso de enlace también permite que una aplicación construya y someta una instrucción (dinámica) SQL para su ejecución inmediata. El enlace parsea todas las instrucciones SQL. Mientras que las instrucciones manipuladoras de SQL están limitadas a producir código ejecutable, las instrucciones de definición y control son parseadas por enlace y dejadas en forma que se interpreten al momento de la corrida. A la salida del enlace se le llama plan o paquete.

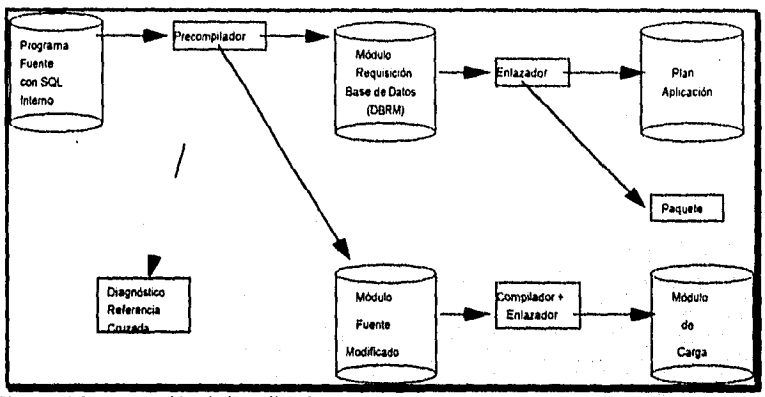


Figura 6.2 Preparación de la aplicación.

- **Supervisor Run-time:** Se refiere a los servicios de DB2 que controlan la ejecución de la aplicación al momento de la corrida. La ejecución de una llamada SQL dentro de un programa de aplicación sigue los pasos mostrados en la Figura 6.3. Se carga el programa de aplicación y hace una LLAMADA al precompilador, el control va al supervisor de run-time vía el módulo de lenguaje de interface DB2. El supervisor run-time recupera el plan de la aplicación, usa la información de control en él, y requiere al administrador de información almacenada para accesar a la base de datos.

- **Administrador de datos almacenados:** Es el componente del sistema que maneja la base de datos física. Incluye lo que DB2 llama el Administrador de Datos, así como al Administrador de Buffer, administrador de Bitácora, etc. Juntos realizan todas las funciones necesarias involucradas en trabajar con la base de datos almacenada -buscar, recuperar, actualizar- cuando se requiere en el plan de la aplicación. Este componente actualiza apropiadamente los índices. Para obtener el mejor desempeño de albercas de buffers, emplea sofisticadas técnicas de buffereo tales como el buffereo de lectura adelantada y el de vista lateral. El administrador de información almacenada es capáz de proporcionar acceso seccionado y enlazado a las tablas del sistema en el catálogo del mismo. El administrador de datos accesa datos o índices proporcionando

identificadores de pagina al administrador de buffer. El tamaño de la página es de 4096 (4 Kbytes) bytes y corresponde al tamaño de la pagina del sistema operativo.

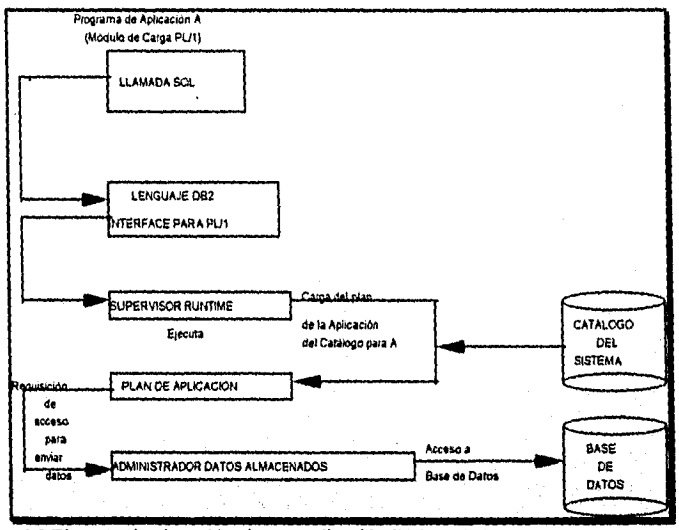


Figura 6.3 Ejemplo de ejecución de una aplicación PL/1 en DB2.

Procesamiento Interactivo. DB2 también permite que los usuarios en línea accesen a la base de datos usando la facilidad DB2I, la cual es una aplicación en línea que corre bajo DB2. Para que un usuario en línea establezca comunicación con esta aplicación, deben usarse los servicios de un administrador de comunicaciones de datos (DC). En el caso de que un usuario DB2 desee tal acceso, la función del administrador (DC) se realiza por el componente TSO de MVS, mediante la facilidad de comunicaciones de datos DC de IMS, o por el CICS. DB2I acepta instrucciones SQL desde una terminal y las pasa a DB2 para su ejecución. Aún durante ejecución interactiva, SQL se compila y se genera un plan de aplicación para él; los resultados se regresan de la ejecución a la terminal. Este plan se descarta después de la ejecución.

Utilerías. Los servicios de la base de datos incluyen un juego de utilerías. Estas se describirán posteriormente.

6.2.3 Otras Funciones Relacionadas a la Compilación y Ejecución de SQL

Durante la compilación y ejecución de queries o aplicaciones se requiere de varias funciones. Rápidamente revisaremos las más importantes:

- *Optimización*: Esta función tiene que ver con la elección de un plan de acceso óptimo para implementar una recuperación o requisición de actualización SQL.

- *Recompilación de planes de aplicación:* Cada vez que se crea o se deja un índice, el plan de la aplicación correspondiente se marca "no válido" en el catálogo por el supervisor del run-time. En una invocación subsecuente de un plan no válido, se invoca al enlazador para recompilarlo sobre las bases del estado actual de los índices. Este proceso de *enlace automático* es transparente para el usuario.

- *Autorización de chequeo*: El enlazador también checa si al usuario que lo invocó se le permite realizar operaciones involucrados con el DBRM. DB2 usa un identificador de autorización del requeridor para determinar si se permiten privilegios de acceso. IMS y CICS proporcionan identificadores de autorización y controlan su uso. Cada medio ambiente conectado indica a DB2 su tipo de conexión a través de la facilidad IDENTIFY. Para prevenir requisiciones no autorizadas de IDENTIFY, una instalación puede controlar quién se puede conectar a DB2 y qué tipo de conexión puede usar.

6.3 Definición de Datos en DB2

Las facilidades de definición permiten la creación, borrado, y alteración (cuando es apropiado) de tablas base, vistas, e índices. Estas instrucciones incluyen a las siguientes:

Para Tablas	Para Vistas	Para Indices
CREATE TABLE	CREATE VIEW	CREATE INDEX
ALTER TABLE		ALTER INDEX
DROP TABLE	DROP VIEW	DROP INDEX

No existe la instrucción ALTER VIEW. ALTER INDEX si, pero tiene que ver con los parámetros físicos de un índice. Durante la creación ningun ordenamiento de tablas base es explícitamente impuesto. El orden de las columnas es implícito en virtud del orden de los nombres de las columnas en la instrucción CREATE TABLE.

DB2 soporta el concepto de valores nulos. Cualquier columna puede contener un valor nulo a menos que la definición de esa columna en CREATE TABLE especifique explícitamente NOT NULL. Una columna en la que se permiten los valores nulos se representa físicamente en la base de datos almacenada por dos columnas: la del dato, y una oculta, un indicador de 1 byte donde un valor de X'FF' significa que el valor del dato correspondiente va a ignorarse y un valor de X'00' indica que el valor correspondiente es verdadero (no nulo).

6.3.1 El Catálogo del Sistema

En DB2 las tablas llamadas *catálogo* son accesibles vía SQL para el administrador de la base de datos u otros usuarios autorizados. El catálogo del sistema DB2 contiene una variedad de información incluyendo las definiciones de las tablas base, vistas, índices, aplicaciones, usuarios, privilegios de acceso, y planes de la aplicación. Estas descripciones son referidas por el sistema para realizar ciertas tareas; por ejemplo, durante la optimización de query el componente enlace accesa al catálogo para obtener información del índice.

DB2 toma un acercamiento uniforme para el almacenamiento de los datos y del catálogo: ambos se almacenan como tablas. En lugar de dar una descripción exhaustiva del catálogo, remarcaremos su contenido refiriendo sólo algunas tablas importantes:

1. SYSTABLES: Tiene una entrada para cada tabla base en el sistema. La información para cada tabla contiene, entre otras cosas, su nombre, el nombre del creador, y el total de columnas que contiene.

2. SYSCOLUMNS: Contiene una entrada para cada columna (atributo) definido en el sistema. Para cada nombre de columna, se almacena el nombre de la tabla al que pertenece, su tipo, y otra información. El mismo nombre de columna puede aparecer en varias tablas.

3. SYSINDEXES: Para cada índice contiene su nombre, el de la tabla indexada, el nombre del usuario quien creó el índice, etc.

Consultando al Catálogo de Información. Debido a que el catálogo se organiza en términos de tablas, puede consultarse usando SQL, tal como cualquier otra tabla. Por ejemplo, consideremos el query

```
SELECT    NAME
FROM      SYSTABLES
WHERE     COLCOUNT > 5
```

Este query SQL accesa al catálogo para listar los nombres de tablas que contienen más de cinco columnas.

El nombre del creador del catálogo de tablas es SYSIBM. Por tanto, el nombre completo de una tabla como SYSTABLES es referenciado como SYSIBM.SYSTABLES. El sistema automáticamente crea entradas catálogo para las tablas catálogo. Los usuarios autorizados tienen acceso al catálogo para consulta. Así, aquéllos que tienen el privilegio SELECT sobre catálogos del sistema, si no están familiarizados con la estructura de la base de datos, pueden hacer queries al catálogo y descubrir más acerca de él.

Por ejemplo, el query

```
SELECT    TBNAME
FROM      SYSIBM.SYSCOLUMNS
WHERE     NOMBRE='DNUMERO'
```

Lista los nombres de las tablas DEPTO y DEP_LOC que contienen la columna DNUM. La disponibilidad para el usuario de la misma interface SQL para accesar meta-data, y no sólo datos, es una facilidad importante de DB2.

Actualización de Información del Catálogo. Mientras que hacer queries al catálogo es informativo para los usuarios, actualizarlo puede ser devastador. Por ejemplo, una rutina de actualización en SQL es

```
DELETE
FROM    SYSIBM.SYSTABLES
WHERE   CREATOR=JUAN
```

elimina todas las tablas en el catálogo creadas por JUAN. Como resultado, las definiciones de estas tablas ya no existen, aunque las tablas sigan existiendo. Las tablas esencialmente se han vuelto inaccesibles! Para prevenir tales situaciones, *no se permiten* UPDATE, DELETE e INSERT contra las tablas del catálogo.

6.4 Manipulación de Datos en DB2

SQL es el lenguaje primario de manipulación de datos de DB2. Ahora ofreceremos la siguiente lista de algunos tipos de recuperaciones y actualizaciones que son soportados por DB2 SQL. Las versiones más recientes de DB2 pueden soportar algunas de las características aquí no disponibles:

1. Recuperaciones sencillas de tablas.
2. Listar la longitud completa de los renglones de la tabla que cumplen una condición pre-especificada.
3. Recuperaciones con eliminación de renglones duplicados.
4. Recuperaciones con valores calculados de columnas.
5. Ordenamiento del resultado de un query.
6. Recuperaciones con condiciones que involucran conjuntos y rangos. Estos se efectúan con varios tipos de constructores:
   a. Usando IN
   b. Usando BETWEEN: DB2 permite una construcción con BETWEEN o NOTBETWEEN; por ejemplo, se permite "WHERE SALARIO BETWEEN 50,000 AND 100,000"
7. Recuperaciones multitabla que involucran JOIN.
8. Queries anidados: El anidamiento puede lograrse pasando los resultados del query interno al externo, usando IN. En general, los queries pueden anidarse a cualquier nivel.

9. Uso de EXISTS: Cuantificación existencial conectando dos subqueries con EXISTS. Debido a que no hay soporte directo para cuantificación universal, se logra por medio de NOT EXISTS.

10. Uso de funciones internas: Las funciones en DB2 SQL incluyen COUNT, SUM, AVG, MAX y MIN. EXISTS también se considera una función interna, a pesar de que, en lugar de retornar un valor numérico o cadena, retorna un valor verdadero.

11. Agrupamiento y condiciones sobre grupos.

12. UNION: Es posible tomar la unión de resultados de subqueries. Los resultados deben ser compatibles a unión. La facilidad de incluir cadenas en la instrucción SELECT viene de la mano con UNION en DB2. Por ejemplo, para obtener una lista de las personas que trabajaron más de 40 horas y una lista de las que trabajaron en el proyecto 5, podríamos poner

```
SELECT   EIMSS, 'trabajaron más de 40 horas'
FROM     TRABAJA_EN
WHERE    HORAS > 40
UNION    SELECT   EIMSS, 'trabajó en proyecto 5'
         FROM     TRABAJA_EN
         WHERE    PNO = 5
```

El resultado podría verse así:

```
1000 trabajó más de 40 horas
1002 trabajó más de 40 horas
1003 trabajó más de 40 horas
1002 trabajo en el proyecto 5
1007 trabajo en el proyecto 5
```

Notemos que, si no hubiera habido argumentos de cadena en la cláusula SELECT, la entidad duplicada (1002) habría sido eliminada.

13. CONTAINS, INTERSECTION, y MINUS: DB2 SQL *no soporta* estos operadores. Deben manejarse usando EXISTS y NO EXISTS. Tampoco, OUTER UNION es soportada por DB2.

14. Inserción.

15. Borrado: Se logra usando DELETE. Para borrar la definición de la tabla completamente, debe usarse DROP TABLE.

16. Modificación: Se logra usando UPDATE.

Las facilidades de actualización en DB2 SQL tienen el inconveniente de que, cuando un INSERT, DELETE o UPDATE involucran a un subquery, el subquery anidado no puede referir a la tabla que es el destino de la operación.

6.4.1 Procesamiento de Vistas

Una vista puede definirse en DB2 usando CREATE VIEW AS seguido de un query SQL. Para definición de vistas en DB2 se aplican los siguientes puntos:

- Una definición de vista puede involucrar una o más tablas; puede usar joins así como funciones internas.
- Si no se especifican nombres de columnas en la definición de la vista, pueden asignarse automáticamente, excepto cuando se usan funciones internas, expresiones aritméticas, o argumentos.
- El query SQL usado en la definición de vistas puede no usar UNION u ORDER BY.
- Pueden definirse vistas sobre vistas ya existentes.
- Cuando se define una vista usando la cláusula WITH CHECK OPTION, un INSERT o UPDATE contra tal vista es sometido a un chequeo para confirmar que la definición de la vista satisfece verdaderamente al predicado.

Recuperaciones desde Vistas. Las vistas son tratadas como cualquier tabla base para especificar queries de recuperación. Pueden ocurrir problemas cuando un atributo de la vista es resultado de una función interna aplicada a una tabla base fundamental.

Actualización de Vistas. DB2 no tiene facilidad de investigar lo que un usuario desea hacer cuando especifica una actualización de la vista. Además, no hay facilidad de analizar y determinar si cierta actualización proporciona un juego único de actualizaciones sobre las relaciones de la base. Por lo tanto, DB2 toma un acercamiento restringido permitiendo actualizaciones sólo sobre vistas de una sola relación. Además, se aplican las siguientes restricciones para *vistas de una sola relación*:

- Una vista es no actualizable si (a) su definición involucra una función interna, (b) su definición involucra DISTINCT en la cláusula SELECT, (c) su definición incluye un subquery y la cláusula FROM se refiere a la tabla base sobre la cual se define la vista, o (d) hay un GROUP BY en la definición de la vista.
- Si un campo de la vista se deriva de una expresión aritmética o constante, *no se permiten* INSERT o UPDATE; sin embargo, DELETE si (ya que un renglón correspondiente puede borrarse de la tabla base).

6.4.2 Uso de SQL Contenido

Daremos algunos detalles relacionados al uso de SQL contenido en DB2, y programas en lenguaje ensamblador. Las guías del contenido son las siguientes:

- Cualquier tabla base o vista usada por el programa debe declararse por medio de una instrucción DECLARE. Esto facilita el seguimiento del programa, y ayuda al precompilador a realizar chequeos de sintaxis.

- Las instrucciones SQL contenidas deben ser precedidas por EXEC SQL e ingresarse en un lugar donde pueda ir cualquier instrucción *ejecutable* del lenguaje huésped.

- El SQL contenido puede involucrar facilidades de definición de datos, tales como CREATE TABLE y DECLARE CURSOR, que son puramente declarativas.

- Las instrucciones SQL pueden referenciar variables del lenguaje huésped precedidas por punto y coma.

- Las variables huésped que reciben valores de SQL deben tener tipos de datos compatibles con las definiciones de campo de SQL. La compatibilidad se define muy débilmente; por ejemplo, las cadenas de caracteres de longitud variable o datos numéricos de naturaleza binaria o decimal se consideran compatibles. DB2 realiza las conversiones apropiadas.

- Area de Comunicaciones SQL (SQLCA) sirve como el área común de retroalimentación entre el programa de aplicación y DB2. Un indicador de status SQLCODE contiene un valor numérico mostrando el resultado de un query.

- No se requiere cursor para queries de recuperación SQL que retornan una sola tupla o para instrucciones UPDATE, DELETE o INSERT.

- Puede usarse un programa especial de utilerías llamado DCLGEN (generador de declaraciones) para construir instrucciones DECLARE TABLE automáticamente en PL/1 a partir de las definiciones CREATE TABLE en SQL. Las estructuras PL/1 o COBOL correspondientes a la definición de tablas también se generan automáticamente.

- La instrucción WHENEVER, colocada fuera de línea, permite que sea checada una condición específica en el SQLCODE.

6.4.3 Integridad Referencial

Se basa en la noción de llaves externas en una relación que depende de las llaves primarias en algunas otras relaciones. Simplemente poner, el argumento de integridad establece que la llave externa pueda ser nula o tener un valor que refiera a un valor válido *ya presente*

6.4.2 Uso de SQL Contenido

Daremos algunos detalles relacionados al uso de SQL contenido en DB2, y programas en lenguaje ensamblador. Las guías del contenido son las siguientes:

- Cualquier tabla base o vista usada por el programa debe declararse por medio de una instrucción DECLARE. Esto facilita el seguimiento del programa, y ayuda al precompilador a realizar chequeos de sintaxis.

- Las instrucciones SQL contenidas deben ser precedidas por EXEC SQL e ingresarse en un lugar donde pueda ir cualquier instrucción *ejecutable* del lenguaje huésped.

- El SQL contenido puede involucrar facilidades de definición de datos, tales como CREATE TABLE y DECLARE CURSOR, que son puramente declarativas.

- Las instrucciones SQL pueden referenciar variables del lenguaje huésped precedidas por punto y coma.

- Las variables huésped que reciben valores de SQL deben tener tipos de datos compatibles con las definiciones de campo de SQL. La compatibilidad se define muy débilmente; por ejemplo, las cadenas de caracteres de longitud variable o datos numéricos de naturaleza binaria o decimal se consideran compatibles. DB2 realiza las conversiones apropiadas.

- Area de Comunicaciones SQL (SQLCA) sirve como el área común de retroalimentación entre el programa de aplicación y DB2. Un indicador de status SQLCODE contiene un valor numérico mostrando el resultado de un query.

- No se requiere cursor para queries de recuperación SQL que retornan una sola tupla o para instrucciones UPDATE, DELETE o INSERT.

- Puede usarse un programa especial de utilerías llamado DCLGEN (generador de declaraciones) para construir instrucciones DECLARE TABLE automáticamente en PL/1 a partir de las definiciones CREATE TABLE en SQL. Las estructuras PL/1 o COBOL correspondientes a la definición de tablas también se generan automáticamente.

- La instrucción WHENEVER, colocada fuera de línea, permite que sea checada una condición específica en el SQLCODE.

6.4.3 Integridad Referencial

Se basa en la noción de llaves externas en una relación que depende de las llaves primarias en algunas otras relaciones. Simplemente poner, el argumento de integridad establece que la llave externa pueda ser nula o tener un valor que refiera a un valor válido *ya presente*

como valor de llave primaria en alguna otra tabla. Si este argumento se viola durante una operación de actualización, el RDBMS se supone debe rechazar la actualización o tomar una *acción correctiva*.

DB2 permite la designación de llaves primarias y externas en los comandos CREATE TABLE y ALTER TABLE. También hay una opción RESTRICT en DB2. Por ejemplo, para prevenir el borrado de un departamento en el cual los empleados están actualmente asignados.

DB2 permite que las llaves primarias sean declaradas NOT NULL o NOT NULL WITH DEFAULT. El último permite un blanco o cero como valor de llave primaria. Con un índice único sobre la columna llave primaria, si no se especifica NOT NULL WITH DEFAULT, sólo se permite un valor con cero o blanco en la llave.

En DB2, se aplican ciertas reglas a la inserción, borrado, o modificación de valores de llaves externas, así como para borrar o modificar valores de llave primaria. Debemos notar que el borrado de una llave externa (colocándole en valor nulo) o inserción de un valor de llave primaria *no ocasiona* una violación de argumento de integridad. Las reglas pueden resumirse como sigue:

1. Un valor de llave primaria puede modificarse si tiene valores llave externos no correspondientes.

2. Si un usuario intenta borrar un valor de llave primaria, y existe un valor correspondiente de llave externa, entonces:

    a. El borrado es excluido si el argumento de llave externa es RESTRICT.

    b. Las tuplas correspondientes (en otras tablas) con valores llave externa correspondientes se borran si el argumento de especificación es CASCADA.

    c. Los valores llave externa correspondientes se ponen en nulo si la especificación de argumento es SET NULL.

3. La inserción o modificación de un valor llave externa se permite solo si existe un valor de llave primaria correspondiente.

La aplicación de argumentos de integridad referencial de acuerdo a las reglas precedentes implica que el sistema realice procesamiento interno involucrando las operaciones necesarias de I/O, candados de control de concurrencia, etc. En general, el sistema usa índices, si existen, para el chequeo. El sistema hace el chequeo a nivel Administrador de Datos sin pasar los datos a las "capas superiores" del sistema llamado Sistema Relacional de Datos. Así, el proceso tiende a ser más eficiente que si hubiera sido hecho por alguna aplicación. Para operaciones batch que involucran el borrado o inserción de gran número

de renglones, puede ser más eficiente para una aplicación aplicar los argumentos de integridad referencial que tomar una acción correctiva a gran escala, en vez de dejar que el sistema cheque el argumento de borrado o inserción en cada renglón.

Utilerías tales como LOAD proporcionan un medio de eliminar los argumentos de chequeo. Para eliminar el chequeo durante actualizaciones batch, puede usarse la facilidad ALTER TABLE para eliminar las especificaciones de llave externa temporalmente. Entonces, después de la actualización, puede usarse la utilería CHECK DATA para asegurarse que los datos son consistentes.

El catálogo mantiene información de integridad referencial en una variedad de lugares, incluyendo los siguientes:

- SYSFOREIGNKEYS contiene el nombre del argumento de llave externa llamado RELNAME y las columnas que contienen la llave.
- SYSCOLUMNS.KEYSEQ, FOREIGNKEY indica si la columna es parte de una llave primaria o parte de una llave externa.
- SYSTABLESPACE.STATUS indica si el espacio de tabla está en status pendiente de chequeo con respecto al argumento de integridad.

Diferentes RDBMSs proporcionan especificación de integridad referencial y aplicación a diferentes niveles. En algunos sistemas, estas funciones se dejan a que la aplicación las provea.

6.5 Almacenamiento de Datos en DB2

Una base de datos en DB2 es una colección de objetos lógicamente relacionados. Estos objetos son las distintas tablas e índices físicamente almacenados. DB2 usa una terminología especial para describir las áreas de almacenamiento particionadas. **Espacio de tabla** se refiere a la parte del almacenamiento secundario donde se almacenan las tablas, y **espacio de índice** a la parte donde se almacenan los índices. La Figura 6.4 muestra un esquema de la estructura de almacenamiento de DB2. La colección total de datos en un sistema consta de usuarios de bases de datos DBX y DBY y el catálogo del sistema de base de datos.

Una **página** es la unidad básica de transferencia de datos entre almacenamiento secundario y primario. **Espacio** es una colección de páginas dinámicamente extendible. Cada espacio pertenece a un **grupo de almacenamiento**, el cual es una colección de áreas de almacenamiento de acceso directo del mismo tipo de dispositivo. No existe correspondencia 1:1 entre una base de datos y un grupo de almacenamiento. En la Figura 6.4 el mismo grupo de almacenamiento contiene un espacio índice y espacio tabla de la base de datos DBX y un espacio de tabla de la base de datos DBY. Una **base de datos** es una unidad de **inicio/fin** que el operador de consola habilita o deshabilita a través de un

de renglones, puede ser más eficiente para una aplicación aplicar los argumentos de integridad referencial que tomar una acción correctiva a gran escala, en vez de dejar que el sistema cheque el argumento de borrado o inserción en cada renglón.

Utilerías tales como LOAD proporcionan un medio de eliminar los argumentos de chequeo. Para eliminar el chequeo durante actualizaciones batch, puede usarse la facilidad ALTER TABLE para eliminar las especificaciones de llave externa temporalmente. Entonces, después de la actualización, puede usarse la utilería CHECK DATA para asegurarse que los datos son consistentes.

El catálogo mantiene información de integridad referencial en una variedad de lugares, incluyendo los siguientes:

- SYSFOREIGNKEYS contiene el nombre del argumento de llave externa llamado RELNAME y las columnas que contienen la llave.
- SYSCOLUMNS.KEYSEQ, FOREIGNKEY indica si la columna es parte de una llave primaria o parte de una llave externa.
- SYSTABLESPACE.STATUS indica si el espacio de tabla está en status pendiente de chequeo con respecto al argumento de integridad.

Diferentes RDBMSs proporcionan especificación de integridad referencial y aplicación a diferentes niveles. En algunos sistemas, estas funciones se dejan a que la aplicación las provea.

6.5 Almacenamiento de Datos en DB2

Una base de datos en DB2 es una colección de objetos lógicamente relacionados. Estos objetos son las distintas tablas e índices físicamente almacenados. DB2 usa una terminología especial para describir las áreas de almacenamiento particionadas. **Espacio de tabla** se refiere a la parte del almacenamiento secundario donde se almacenan las tablas, y **espacio de índice** a la parte donde se almacenan los índices. La Figura 6.4 muestra un esquema de la estructura de almacenamiento de DB2. La colección total de datos en un sistema consta de usuarios de bases de datos DBX y DBY y el catálogo del sistema de base de datos.

Una **página** es la unidad básica de transferencia de datos entre almacenamiento secundario y primario. **Espacio** es una colección de páginas dinámicamente extendible. Cada espacio pertenece a un **grupo de almacenamiento**, el cual es una colección de áreas de almacenamiento de acceso directo del mismo tipo de dispositivo. No existe correspondencia 1:1 entre una base de datos y un grupo de almacenamiento. En la Figura 6.4 el mismo grupo de almacenamiento contiene un espacio índice y espacio tabla de la base de datos DBX y un espacio de tabla de la base de datos DBY. Una **base de datos** es una unidad de **inicio/fin** que el operador de consola habilita o deshabilita a través de un

comando START o STOP. Las tablas pueden moverse de una a otra base de datos sin producir ningun efecto sobre los usuarios o los programas de los usuarios.

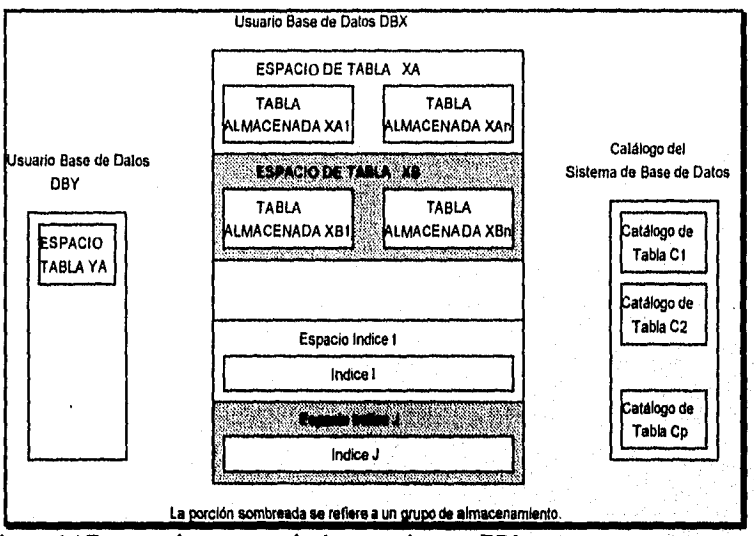


Figura 6.4 Esquema de estructura de almacenamiento en DB2.

Los grupos de almacenamiento se manejan usando archivos VSAM (juegos de datos entre-secuenciados). Dentro de una página, DB2 maneja la reorganización sin usar para nada VSAM. DB2 permite que el DBA y el administrador del sistema especifiquen los detalles de la estructura de almacenamiento, usando diferentes instrucciones. Para cada objeto descrito (tal como tabla, espacio tabla, índice, espacio índice, base de datos o grupo de almacenamiento), las tres instrucciones usadas uniformemente son **CREATE**, **ALTER** y **DROP**. Los usuarios *no requieren* saber acerca de la organización interna de almacenamiento para usar el sistema. A pesar de que se requiere de un espacio tabla para almacenar una tabla, el sistema asigna un espacio tabla *default* cuando el creador falla en la especificación de uno. Una base de datos es una entidad lógica cuyos componentes físicos pueden moverse y manipularse libremente sin afectar la integridad de la base de datos. **CREATE TABLESPACE** identifica la base de datos a la cual pertenece el espacio tabla.

6.5.1 Tablas Espacio y Tablas Almacenadas

El tamaño de la página en espacio tabla es de 4096 o 32768 bytes. Un espacio tabla puede crecer en tamaño si se agrega más almacenamiento de un grupo de almacenamiento, hasta un límite superior de 64 Gb. Un espacio tabla es la unidad de almacenamiento sujeta a reorganización o recuperación por un comando de consola. Debido a que sería muy

ineficiente manejar un gran espacio tabla de esta forma, DB2 permite particionar los espacio tabla.

A un espacio tabla no particionado se le llama **espacio tabla simple**. En la mayoría de los casos almacena una sola tabla. Las tablas múltiples pueden almacenarse en el mismo espacio tabla para mejorar el desempeño si tienen alta probabilidad de ser accesadas en conjunto. El índice, si existe, de una tabla va al espacio índice. Una tabla con índice de agrupamiento se carga inicialmente al espacio tabla en el orden de la llave, usando la utilería load. Se incluyen huecos para retener los registros subsecuentemente insertados. Sin un índice de agrupamiento, los registros pueden cargarse en cualquier orden. Las inserciones van al final del archivo.

Un **espacio tabla particionado** contiene una tabla que se particiona (agrupa) en rangos de valores de algunos campos particionados. Se *requiere* de un índice de agrupamiento sobre esos campos y no cambiarse. Así, la tabla EMPLEADO puede particionarse usando un índice de agrupamiento sobre DNO. La ventaja del espacio tabla particionado es que cada partición se trata como un objeto separado de almacenamiento para recuperación y reorganización y así asociarse con un grupo de almacenamiento diferente.

Particionar un espacio tabla proporciona varias ventajas en grandes tablas:

- *Disponibilidad de datos mejorada*: Un usuario puede efectuar mantenimiento normal sobre una partición de la tabla mientras el resto permanece disponible para las utilerías o procesamiento SQL.

- *Desempeño de utilerías mejorado*: Una utilería puede trabajar sobre todas las particiones simultáneamente, en vez de trabajar una por una. Esto puede reducir significativamente la cantidad de tiempo necesario para completar la tarea.

- *Tiempo de respuesta a query mejorado*: Cuando DB2 rastrea los datos para responder a un query, puede a veces scannear a través de particiones simultáneamente, en lugar de scannear a través del espacio completo de principio a fin. Esta mejora es más notable para queries complejos o que requieren que DB2 rastree grandes cantidades de datos.

Un **espacio tabla segmentado** está diseñado para almacenar más de una tabla. El espacio disponible se divide en grupos de páginas llamados *segmentos*, cada uno del mismo *tamaño*. Cada segmento contiene renglones de una sola tabla. Para buscar en todos los renglones de la tabla, es innecesario rastrear el espacio tabla completo, sino sólo a los segmentos que contienen esa tabla. Si una tabla es eliminada, sus segementos inmediatamente se vuelven reusables. Todos los segmentos deben residir en el mismo conjunto de datos definido por el usuario o en el mismo grupo de almacenamiento.

Cada renglón en una tabla constituye un **registro almacenado** -cadena de bytes que contiene un prefijo que contiene información de control del sistema y hasta n campos

almacenados, donde n es el número de columnas en la tabla base. Los campos nulos al final de un registro de longitud variable no se almacenan. Internamente, cada registro tiene un **id de registro (RID)** único dentro de una base de datos; que, en cambio, contiene la posición inicial del registro dentro de la página.

Cada **campo almacenado** incluye tres elementos:

- Un campo prefijo que contiene la longitud de datos, si es variable.
- Un prefijo indicador nulo que indica si el campo contiene un valor nulo.
- Un valor de datos codificado.

DB2 ha adoptado una estrategia de almacenar todo tipo de datos de tal forma que ellos son contemplados como cadenas de bytes y la instrucción "comparación lógica" siempre arroja un resultado correcto (aún para INTEGER). La interpretación de cadenas de bytes no es problema del administrador de datos almacenados. Los campos variables ocupan sólo el espacio actual requerido. La compresión o encriptado de datos se deja abierta a un procedimiento proporcionado por el usuario, el cual puede interponerse cada vez que se lee o almacena un registro. Los registros *dentro de una página* pueden reorganizarse sin cambiar sus RID's.

6.5.2 Espacio Indice e Indices

Un espacio índice corresponde al almacenamiento ocupado por un índice. A diferencia del espacio tabla, se crea automáticamente. Las páginas índice son de 4096 bytes, pero puede bloquearse un cuarto de página a la vez. Un espacio índice que contiene un índice de agrupamiento para un espacio tabla particionado se considera a sí mismo particionado.

Los índices son B+trees en el cual cada nodo es una página. Las páginas hoja se encadenan para proporcionar acceso secuencial a los renglones (en el espacio tabla). Una tabla puede tener un índice de agrupamiento y cualquier número de no agrupados. Las páginas hoja de un índice no agrupado accesan a los renglones del espacio tabla en orden diferente al del orden físico. Por tanto, para el eficiente procesamiento secuencial de una tabla, es esencial tener un índice de agrupamiento.

6.5.3 Utilerías DB2

DB2 proporciona varias utilerías que pueden usarse por desarrolladores de aplicaciones para manipular datos almacenados y evitar escribir programas individuales. La siguiente es una lista de algunas de las utilerías usadas para manejar "objetos de datos" DB2.

- **LOAD:** Carga datos de los juegos de datos del método de acceso secuencial básico (BSAM), tablas SQL/DS descargadas, y tablas DB2 descargadas a tablas DB2. La partición de un espacio tabla puede reemplazarse usando LOAD REPLACE. La

utilería cual asegura que los datos cargados en la tabla e índice sean consistentes y usables.

- **REORG:** Reorganiza espacios tabla y espacios índice. Puede reestablecer el espacio libre y la secuencia física descrita por un índice de agrupamiento. Recobra el espacio perdido por fragmentación y eliminación de tablas.

- **CHECK:** Prueba si los índices son consistentes con los datos que indexan y arroja mensajes de alarma cuando encuentra inconsistencias. La utilería CHECK DATA busca violaciones de argumento referencial, índice donde existen y puede usarse para borrarlas.

- **STOSPACE:** Reporta información acerca de del uso actual de espacio por espacio tabla y espacio índice en un grupo dado de almacenamiento DB2. Con esta información se puede actualizar al catálogo de DB2.

- **REPAIR:** Repara datos. Los datos involucrados pueden ser propiedad del usuario o datos que no se accesarían explícitamente, tales como entradas de índices.

- **Utilerías de Diagnóstico:** DB2 también proporciona una serie de utilerías para ayudar a que el equipo de desarrollo de aplicaciones diagnostique problemas. Entre otras cosas, estas utilerías pueden crear vertederos basados en eventos DB2, verificar la integridad del catálogo de espacio tabla, y desplegar los contenidos de mensajes de recuperación.

6.6 Características Internas de DB2

En esta sección resumimos las características de DB2 relacionadas con seguridad, autorización, y procesamiento de transacciones.

6.6.1 Seguridad y Autorización

Generalmente hablando, en DB2 la seguridad de los datos se direcciona en dos niveles:

1. El mecanismo vista puede usarse para ocultar datos sensibles a usuarios no autorizados.

2. El subsistema de autorización, el cual da privilegios específicos a ciertos usuarios, les permite concesionar esos privilegios a otros usuarios selectivamente y dinámicamente para revocarlo si lo desean.

El DBA o un administrador apropiado está encargado de las decisiones relacionadas a concesión y revocación de privilegios específicos a los usuarios. La decisión política puede transmitirse a DB2 en la forma de instrucciones CREATE VIEW o GRANT y REVOKE. Esta información reside en el catálogo del sistema. La responsabilidad del sistema es

utilería load asegura que los datos cargados en la tabla e índice sean consistentes y usables.

- **REORG:** Reorganiza espacios tabla y espacios índice. Puede reestablecer el espacio libre y la secuencia física descrita por un índice de agrupamiento. Recobra el espacio perdido por fragmentación y eliminación de tablas.

- **CHECK:** Prueba si los índices son consistentes con los datos que indexan y arroja mensajes de alarma cuando encuentra inconsistencia. La utilería CHECK DATA busca violaciones de argumento referencial, indica dónde existen y puede usarse para borrarlas.

- **STOSPACE:** Reporta información acerca de del uso actual de espacio por espacio tabla y espacio índice en un grupo dado de almacenamiento DB2. Con esta información se puede actualizar al catálogo de DB2.

- **REPAIR:** Repara datos. Los datos involucrados pueden ser propiedad del usuario o datos que no se accesarían explícitamente, tales como entradas de índices.

- **Utilerías de Diagnóstico:** DB2 también proporciona una serie de utilerías para ayudar a que el equipo de desarrollo de aplicaciones diagnostique problemas. Entre otras cosas, estas utilerías pueden crear vertederos basados en eventos DB2, verificar la integridad del catálogo de espacio tabla, y desplegar los contenidos de mensajes de recuperación.

6.6 Características Internas de DB2

En esta sección resumimos las características de DB2 relacionadas con seguridad, autorización, y procesamiento de transacciones.

6.6.1 Seguridad y Autorización

Generalmente hablando, en DB2 la seguridad de los datos se direcciona en dos niveles:

1. El mecanismo vista puede usarse para ocultar datos sensibles a usuarios no autorizados.

2. El subsistema de autorización, el cual da privilegios específicos a ciertos usuarios, les permite concesionar esos privilegios a otros usuarios selectivamente y dinámicamente para revocarlo si lo desean.

El DBA o un administrador apropiado está encargado de las decisiones relacionadas a concesión y revocación de privilegios específicos a los usuarios. La decisión política puede transmitirse a DB2 en la forma de instrucciones CREATE VIEW o GRANT y REVOKE. Esta información reside en el catálogo del sistema. La responsabilidad del sistema es

reforzar estas decisiones cuando se intenten operaciones de recuperación o actualización; esta función se lleva a cabo por el componente enlace (véase Figura 6.2).

Identificación del Usuario. Los usuarios legítimos son conocidos por DB2 en términos de un ID (identificador) de autorización llamado AUTHID, el cual se asigna por los administradores del sistema. Es responsabilidad del usuario usar ese ID cuando se identifica al sistema. Los usuarios de DB2 se identifican primero en CICS, IMS o TSO; ese subsistema pasa el ID a DB2. Por tanto, la responsabilidad de checar el ID del usuario cae en uno de estos subsistemas. La clave de USUARIO se refiere a una variable del sistema cuyo valor es un ID de autorización. Si cierto usuario está usando una vista (para recuperación o actualización), la variable USUARIO contiene el ID del usuario que está *usando* la vista y no el ID del que lo creó.

Vistas como Mecanismo de Seguridad. Es posible usar vistas con propósitos de seguridad bloqueando datos no deseados a usuarios no autorizados. Eligiendo las condiciones apropiadas en la cláusula WHERE, así como incluir sólo las columnas en la cláusula SELECT que se le permiten a un usuario ver, el diseñador del sistema puede guardar ciertos datos ocultos al usuario, Las vistas definidas sobre información del catálogo del sistema, permiten a un usuario ver sólo partes seleccionadas del catálogo. Aplicando funciones agregadas tales como AVG y SUM, el diseñador puede permitir que un usuario vea un sumario estadístico de la tabla base y no de valores individuales.

En DB2, cuando se Inserta o Actualiza un registro a través de una vista, el renglón nuevo o actualizado debe obedecer las condiciones de la vista o el predicado es *no* forzado. Esto puede conducir algunas veces a una situación donde el registro nuevo o actualizado desaparezca inmediatamente de la vista del usuario pero aparezca en la tabla base. Para prevenir tales inserciones o borrados, en la definición de la vista debe usarse la instrucción CHECK OPTION.

Mecanismos de Concesión y Revocación. Los privilegios concesionados a los usuarios pueden clasificarse dentro de las siguientes categorías:

- Privilegios de tablas y vistas: Se aplican a tablas base y vistas.
- Privilegios de bases de datos: Se aplican a operaciones con una base de datos (tal como crear una tabla).
- Privilegios del plan de aplicaciones: Se refieren a la ejecución de planes de aplicación.
- Privilegios de almacenamiento: Tienen que ver con el uso de ciertos objetos de almacenamiento, espacios tabla, grupos de almacenamiento, y albercas de buffers.

Existen ciertos privilegios "empaquetados", un término hecho a la medida que refiere a la variedad de privilegios:

- SYSADM (administrador del sistema) es el privilegio de más alto orden e incluye todos los privilegios posibles en el sistema.

- DBADM (administrador de la base de datos) sobre una base especifica permite al poseedor ejecutar cualquier operación sobre la base.

- DBACTRL (control de la base) es similar al DBADM excepto que sólo controla operaciones, y no se permiten operaciones de manipulación de datos.

- DBMAINT (mantenimiento de la base) sobre una base especifica permite al poseedor ejecutar operaciones de mantenimiento de sólo lectura (tales como respaldo) en la base de datos. Es un subconjunto del privilegio DBACTRL.

- SYSOPR (operador del sistema) permite al poseedor efectuar sólo funciones de operación de la consola, sin acceso a la base de datos.

Un ID de autorización tiene el privilegio SYSADM, y representa la función de administración del sistema. Otros IDs pueden poseer el privilegio SYSADM, pero ese privilegio puede revocarse. PUBLIC es un comando del sistema e incluye todos los ID's de autorización.

Algunas notas finales de la forma en que se implementan estas características en DB2:

- Resulta un mayor beneficio del hecho que pueden aplicarse muchos chequeos de autorización al momento del enlace (tal como respaldo) en lugar de ser retrasado hasta el momento de la ejecución.

- DB2 trabaja con varios sistemas acompañantes; junto con ellos, proporciona sistemas de seguridad. Los mecanismos de control individual de MVS, VSAM, IMS y CICS ofrecen protección adicional.

- El arreglo completo de mecanismos de autorización y seguridad es opcional. Así, pueden deshabilitarse, permitiendo que cualquier usuario tenga acceso completo a los privilegios.

6.6.2 Procesamiento de Transacciones

Usemos el siguiente ejemplo para demostrar cómo se escribe una transacción en DB2. Usaremos PL/1, debido a que DB2 no acepta PASCAL. La transacción consiste en dar un incremento R a un empleado con número de IMSS S:

```
TRANS1: PROC OPTIONS (MAIN);
EXEC SQL WHENEVER SQLERROR GO TO ERROR_PROC;
DCL S FIXED DECIMAL (9,0);
DCL R FIXED DECIMAL(7,2);
```

```
GET LIST (S,R);
EXEC SQL UPDATE EMPLEADO
SET SALARIO = SALARIO + R
WHERE IMSS =:S;
EXEC SQL WHENEVER NOTFOUND GO TO PRINT_MSG;
COMMIT;
GO TO EXIT;
PRINT_MSG PUT LIST('EMPLEADO', S, 'NO EN LA BASE DE DATOS'); GO TO
EXIT;
ERROR_PROC: ROLLBACK;
EXIT: RETURN;
END TRANS1;
```

En este ejemplo, la transacción puede fallar si ningún empleado tiene IMSS S (NOTFOUND) o si SQLCODE retorna un valor negativo (SQLERROR).

También podemos insertar chequeos al programa, tales como asegurarnos de que el salario no sea nulo, caso que puede hacer que la transacción falle. La operación COMMIT señala el final de una transacción exitosa: instruye al administrador de la transacción a cometer el cambio en la base de datos -esto es, lo hace permanente, señala el fin de una transacción no exitosa. Instruye al administrador de la transacción a *no* hacer ningun cambio permanente a la base de datos, pero dejarla en el estado en el que estaba antes de iniciar la transacción.

Una transacción típica puede involucrar recuperaciones y actualizaciones; sin embargo, sólo *hay una* operación COMMIT en el programa, así que o se aplican *todos los cambios* o no se aplica *ninguno*. La operación **ROLLBACK** usa las entradas de bitácora y restaura apropiadamente un elemento actualizado a su valor anterior.

Cada operación DB2 se ejecuta en el contexto de alguna transacción. Esto *incluye* aquéllas ingresadas interactivamente a través de DB2I. Una aplicación consta de una serie de transacciones. Las transacciones *no pueden* anidarse.

Ejecutar y Volver Atrás en DB2. El DBMS DB2 es *subordinado* del administrador de transacciones (IMS, CICS o TSO) bajo el cual corre. Actúa como uno de los administradores de recursos proporcionando servicio al administrador de transacciones. Por tanto, deben hacerse notar los siguientes puntos:

- COMMIT y ROLLBACK no son operaciones de bases de datos. Son instrucciones para el administrador de transacciones, el cual no es parte del DBMS.
- Si una transacción actualiza a una base de datos IMS y a una base DB2, o todas las actualizaciones se llevan a cabo o todas se regresan a su estado anterior.

- Un "punto de sincronización" (abreviado *syncpoint*) define el punto en el cual la base de datos es consistente. Al inicio de una transacción COMMIT y ROLLBACK establecen un syncpoint. Nadie más establece dicho punto.

- La operación COMMIT señala el final de una transacción exitosa, establece un syncpoint, efectúa todas las actualizaciones hechas desde el syncpoint anterior, elige todos los cursores abiertos, y libera todos los bloqueos (con algunas excepciones).

- La operación ROLLBACK señala el fin de una transacción no exitosa, establece un syncpoint, cierra todos los cursores abiertos, y libera todos los bloqueos (con algunas excepciones).

Medios de Bloqueo Explícito. DB2 soporta diferentes tipos de bloqueo. El usuario DB2 está principalmente preocupado de los bloqueos exclusivos (X) y compartidos (S). Además del mecanismo de bloqueo interno, DB2 proporciona algunos medios de bloqueo explícito. Una transacción puede contener la siguiente instrucción:

LOCK TABLE <nombre tabla> IN <tipo modo> MODE;

El tipo modo puede ser exclusivo o compartido. El nombre tabla debe ser una tabla base. Un bloqueo exclusivo permite que la transacción bloquee a la tabla completamente; el bloqueo se *libera* cuando el programa (no la transacción) finaliza. Sin embargo, un bloqueo compartido permite que otras transacciones adquieran un bloqueo concurrente sobre la misma tabla o en parte de ella. Hasta que son liberados los bloqueos compartidos, puede lograrse algún bloqueo exclusivo sobre la tabla o parte de ella.

Este medio se proporciona para mejorar la eficiencia de transacciones que necesitan procesar una tabla larga (por ejemplo, producir un listado de 10000 empleados) eliminando el tedio de liberar registros individuales.

Un valor negativo de SQLCODE retornado después de una requisición de bloqueo significa un bloqueo muerto. Para romperlo, el administrador de transacción elige una de las transacciones de bloqueo muerto como *víctima* y regresa automáticamente, o requiere que sea regresado a sí mismo. Esta transacción libera todos los bloqueos y permite que procedan algunas otras transacciones.

6.6.3 SQL Dinámico

El medio SQL dinámico está diseñado exclusivamente para soportar aplicaciones en línea. En algunas aplicaciones caracterizadas por una gran cantidad de variabilidad, en lugar de escribir un query específico para cada condición posible, el diseñador puede encontrar mucho más conveniente construir partes del query dinámicamente (al momento de la corrida) y luego enlazar y ejecutarlos dinámicamente. El proceso ocurre cuando se ingresan instrucciones SQL interactivamente a través de DB2I o QMF. Sin embargo, las

consideraremos como instrucciones SQL *internas* construidas dinámicamente. Sin entrar en detalles de sintaxis, subrayaremos este recurso:

DCL QSTRING CHAR(256) VARING INICIAL
'DELETE FROM EMPLEADO WHERE CONDITION'.

Se declara una variable SQL (SQLVAR), para retener el query al momento de la corrida:

EXEC SQL **DECLARE** SQLVAR STATEMENT;

El QSTRING se modifica adecuadamente cambiando, digamos, CONDITION en la parte WHERE del query desde la terminal. La siguiente instrucción PREPARE ocasionaría que QSTRING se precompilara, enlazara, convirtiera a código objeto, y almacenara en SQLVAR:

EXEC SQL **PREPARE** SQLVAR **FROM** :QSTRING;

Finalmente, la instrucción EXECUTE ejecuta el código compilado:

EXEC SQL **EXECUTE** SQLVAR;

Hagamos notar que PREPARE acepta las diferentes instrucciones SQL excepto EXEC SQL. Las instrucciones a preparar pueden no contener referencias a variables huésped. Sin emabrgo, deben contener parámetros que estén encerrados por signos de interrogación. Los valores de los parámetros son proporcionados al momento de la corrida a través de variables de programa. Por ejemplo, supongamos que la condición WHERE de QSTRING fuera reemplazada por "SALARIO > ? AND SALARIO < ?"; entonces

EXEC SQL **EXECUTE** SQLVAR **USING** :LIM_INF, :LIM_SUP

sustituiría los valores de *variables de programa* lim_inf y lim_sup en lugar de los dos signos de interrogación.

Esta discusión pertenece a las instrucciones de SQL que no retornan ningun valor al programa. Cuando se han recuperado valores vía SELECT generado dinámicamente, el programa típicamente no conoce tales variables. Por tanto, esa información es proporcionada dinámicamente usando otra instrucción dinámica de SQL, llamada **DESCRIBE**. La descripción de los resultados esperados retorna en un área llamada Area Descriptor SQL (SQLDA). Usando el lenguaje de programación huésped se localiza espacio para tales variables. Finalmente, el resultado se recupera un renglón a la vez, usando operaciones cursor. También se soporta la actualización de los resultados usando la opción **CURRENT**.

6.6.4 Opciones para un Mejor Desempeño

En esta sección presentaremos algunas de las mejoras en la Versión 2.3 de DB2 que ayudan a fortalecer el desempeño de aplicaciones. La siguiente es una lista no exhaustiva:

- Paquetes: Un programa fuente P1, después de precompilarse, puede convertirse al módulo de requisición de base de datos correspondiente -digamos, DBRM P1 (véase Figura 6.2). Es posible que dos planes de aplicaciones diferentes usen el mismo DBRM P1. En ese caso, es posible definir un paqueta para P1 e incluirlo en ambos planes. Si las trayectorias de acceso a usarse por DBRM P1 cambian, el paquete puede sujetarse al proceso de enlace otra vez, el cual correrá *una sola vez* sobre P1. Eventualmente, lo que se ejecuta es un plan de aplicación y no el paquete. La noción de paquetes intermedios previene el duplicado de las actividades de enlace para DBRMs que se usan en muchos planes de aplicaciones.

- La cláusula OPTIMIZE for n ROWS: Puede agregarse a cualquier instrucción SQL. Su propósito es sustituir el cálculo estimado de renglones en el resultado de un query. Colocando a n bajo, los usuarios pueden evitar el buffereo intermedio del resultado (llamado "lista de precarga").

- La opción Index Lookaside: Es útil cuando los datos afectados por un query -particularmente una actualización o join- involucran llaves localizadas juntas. En vez de ocasionar una búsqueda repetida del índice, forza a la búsqueda de las páginas hoja adyacentes o al rango de páginas debajo de un nodo índice intermedio, antes de regresar a la raíz para una búsqueda de arriba a abajo.

- Opción "SLOW CLOSE": Permite que todos los juegos de datos (incluyendo tablas o índices) permanezcan abiertos aún después de que un usuario haya usado el comando VSAM CLOSE, hasta que el conteo de juegos de datos actualmente abiertos exceda un determinado parámetro.

- Opción CICS-relacionados para cursores: Cuando tiene que ejecutarse un query SQL dentro de un programa de aplicación debe abrirse un cursor. Si se usa el mismo cursor en repetidas ocasiones, en lugar de hacer varios OPEN, podemos realizar múltiples cargas.

- Joins híbridos: Se han implementado nuevos algoritmos join que combinan las características de un ciclo join anidado y un join sort. También permiten una reducción en tablas temporales usando una lista de valores llave correspondientes, lo cual corresponde a una técnica join llamada "semijoin".

Dentro de las características introducidas en Versiones 2.3 y 3 de DB2 se incluyen las siguientes:

- Albercas de buffers: DB2 permite al usuario almacenar datos temporalmente en albercas buffer. Esto hace disponibles a los datos sin I/O. Sólo cuando cambian los datos se escriben fisicamente a disco. En estas albercas buffer, conocidas como **albercas buffer virtuales** pueden almacenarse hasta 1.6 GBytes. La versión 3 de DB2 pemite el respaldo de albercas de buffer virtuales con 8 GBytes adicionales de acceso rápido. El almacenamiento expandido viene en unidades conocidas como *hiperespacios* (de high performance space), bloques de 2 GBytes que se construyen dinámicamente cuando es necesario almacenar datos en hiperalbercas. Las *hiperalbercas* (high performance albercas) son extensiones al buffereo virtual de albercas; los datos menos accesados en una alberca virtual se mueven a su hiperalberca sin procesamiento de I/O.

- Independencia de partición: La versión 3 de DB2 permite que una aplicación trabaje en una partición de un espacio tabla o espacio índice sin bloquear las otras particiones; las particiones son así independientes la una de la otra. Es posible incrementar la disponibilidad de datos sacando ventaja de la independencia de partición de la siguiente manera:

  -Realizar mantenimientos sobre particiones en vez de espacio tabla y espacio índice completos. Por ejemplo, recuperar, reorganizar, o cargar una partición, y dejar otra disponible para utileria o procesamiento SQL.

  -Uso de los comandos START DATABASE y STOP DATABASE sobre partición en vez de tabla espacio completo. Esto libera la otra partición para utilería o procesamiento SQL.

E. F. Codd el creador del modelo relacional, publicó un artículo en dos partes en *Computer-world* que enlista 12 reglas para determinar si un DBMS es relacional y en qué medida. Codd también menciona que, de acuerdo a estas reglas, aún no se dispone de un sistema completamente relacional. En particular las reglas 6, 9, 10, 11 y 12 son difíciles de satisfacer.

Regla 1: La Regla de la Información

Toda la información en una base de datos relacional se representa explícitamente a nivel lógico de una sola forma -valores en tablas.

Regla 2: Regla de Acceso Garantizado

Todos y cada uno de los datom (dato atómico) en una base de datos relacional garantizan estar lógicamente accesibles acudiendo a una combinación de nombre tabla, valor llave primaria, y nombre de la columna.

Regla 3: Tratamiento Sistemático de Valores Nulos

Los valores nulos (distintos de las cadenas de caracteres vacías o de una cadena de caracteres blancos y distintos de cero o cualquier otro número) son soportados completamente en el DBMS relacional para representar información equivocada de manera sistemática, independientemente del tipo de dato.

Regla 4: Catálogo en Línea Basado en el Modelo Relacional

La descripción de la base de datos se representa a nivel lógico de la misma manera que los datos ordinarios, así los usuarios autorizados pueden aplicar el mismo lenguaje relacional a sus interrogantes como lo aplican a datos regulares.

Regla 5: Regla del Sublenguaje Global de Datos

Un sistema relacional puede soportar varios lenguajes y varios modos de uso terminal (por ejemplo, el modo de llenar blancos). Sin embargo, debe haber al menos un lenguaje cuyas instrucciones sean expresables, por alguna sintaxis bien definida, como cadenas de caracteres y cuya habilidad soporte a todos los elementos en su totalidad: definición de datos, definición de vistas, manipulación de datos (interactiva y por programa), argumentos de integridad, y límites de transacción (begin, commit y rollback).

Regla 6: Actualización de Vistas

Todas las vistas que son teóricamente actualizables son también actualizables por el sistema.

Regla 7: Inserción, Actualización y Borrado de Alto Nivel

Esta capacidad de manejar una relación base o derivada como un sólo operando se aplica no solo a la recuperación de datos sino también a la inserción, actualización y borrado de datos.

Regla 8: Independencia Física de Datos

Los programas de aplicación y actividades de terminal permanecen lógicamente inalterados cada que se hacen cambios en la representación de almacenamiento o en los métodos de acceso.

Regla 9: Independencia Lógica de Datos

Los programas de aplicación y actividades de terminal permanecen lógicamente inalterados cuando información que preserva los cambios de cualquier tipo teóricamente mantiene inalteradas a las tablas base.

Regla 10: Independencia de Integridad

El argumento de integridad especifica que una base de datos relacional debe ser definible en el sublenguaje relacional de datos y almacenable en el catálogo, no en los programas de aplicación.

Debe soportarse al menos los dos siguientes argumentos de integridad:

1. Integridad de entidad: No se permite que ningún componente de llave primaria tenga valor nulo.

2. Integridad referencial: Para cada valor de llave externa no nulo en una base relacional, debe existir una llave primaria correspondiente del mismo dominio.

Regla 11: Independencia de Distribución

Un DBMS relacional tiene independencia de distribución. La independencia de distribución implica que los usuarios no deben preocuparse de si la base de datos es distribuida.

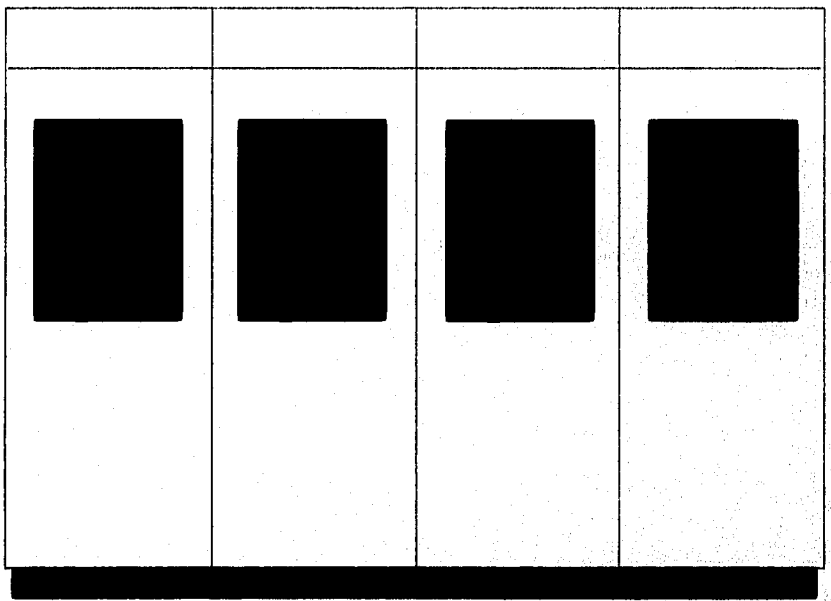
Regla 12: Nosubversión

Si un sistema relacional tiene un lenguaje de bajo nivel (un sólo registro a la vez), ese lenguaje no puede usarse para subvertir u omitir las reglas de integridad o argumento expresadas en el lenguaje relacional de alto nivel (varios registros a la vez).

Existe un extra a estas 12 reglas conocido como la **Regla Cero**: "Cualquier sistema que se diga ser relacional, debe ser capáz de manejar los datos completamente a través de capacidades relacionales".

En base a las reglas anteriores, hoy en día no existe un DBMS totalmente relacional.

CAPITULO 7



Modelo de Datos de Red y el Sistema - IDMS

7.1 Estructuras de Bases de Datos de Red

Existen dos estructuras básicas de datos en el modelo de red: registros y conjuntos.

7.1.1 Registros, Tipos de Registro, y Elementos de Datos

Los datos se almacenan en **registros**; cada registro consta de un grupo de valores de datos relacionados. Los registros se clasifican en **tipos de registros**, donde cada tipo describe la estructura de un grupo de registros que almacenan el mismo tipo de información. A cada tipo de registro se le asigna un nombre, y también un formato (tipo de dato) para cada **elemento** (o atributo). La Figura 7.1 muestra un tipo de registro ESTUDIANTE con elementos de datos NOMBRE, IMSS, DIRECCION, DEPCARRERA, y FECHANAC. En la Figura 7.1 se muestra el **formato** (o **tipo de dato**) de cada elemento.

ESTUDIANTE				
NOMBRE	IMSS	DIRECCION	DEPMAYOR	FECHANAC

nombre del elemento de dato	formato
NOMBRE	CHARACTER 30
IMSS	CHARACTER 9
DIRECCION	CHARACTER 40
DEPMAYOR	CHARACTER 10
FECHANAC	CHARACTER 9

Figura 7.1 Tipo de registro ESTUDIANTE.

El modelo de red permite la definición de elementos complejos de datos. Un **vector** es un elemento de datos que puede tener múltiples valores en un sólo registro. Un **grupo de repetición** permite la inclusión de un conjunto de valores compuestos para un dato en un solo registro. Como se muestra en la Figura 7.2 TRANSCRIPT consta de los cuatro elementos de datos AÑO, CURSO, SEMESTRE y CALIF. El grupo de repetición no es esencial para la capacidad del modelo de red, ya que podemos representar la misma situación con dos tipos de registro y un tipo de conjunto.

ESTUDIANTE					
NOMBRE		TRANSCRIPT			
		AÑO	CURSO	SEMESTRE	CALIF
Smith		1984	CC3320	Invierno	A
		1984	CC3340	Invierno	A
		1984	MAT312	Invierno	B
		1985	CC4310	Primavera	C
		1985	CC4330	Primavera	B

Figura 7.2 Grupo de repetición TRANSCRIPT.

A todos los tipos de datos anteriores se les llaman **elementos de datos actuales**, debido a que sus valores se almacenan en registros. También pueden definirse **datos virtuales** (o **derivados**). El valor de un dato virtual no se almacena en un registro; sino que, se deriva de los datos actuales usando algun procedimiento que se define especificamente para este propósito.

7.1.2 Tipos de Conjuntos y sus Propiedades Básicas

Un **tipo de conjunto** es una descripción de relación 1:N entre dos tipos de registros. La Figura 7.3 muestra cómo representar un tipo de conjunto diagramalmente como una flecha. A este tipo de representación diagramal se le llama **diagrama Bachman**. Cada definición de tipo de conjunto consta de tres elementos básicos:

- Un nombre para el tipo de conjunto.
- Un tipo de registro propietario.
- Un tipo de registro miembro.

Al tipo de conjunto de la Figura 7.3 se le llama DEP_CARRERA; DEPTO es el registro **propietario**, y ESTUDIANTE es el registro **miembro**. Representa la relación 1:N entre departamentos académicos y estudiantes con carreras en esos departamentos. En la base de datos, habrá muchas **ocurrencias de conjuntos** (o **instancias de conjuntos**) correpondientes a un tipo de conjunto. Cada instancia relaciona un registro propietario al conjunto de registros miembro relacionados a él. Por tanto, cada ocurrencia de conjunto está compuesta de:

- Un registro propietario.
- Un número de registros miembro relacionados (cero o más).

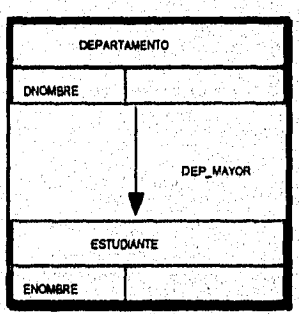


Figura 7.3 Conjunto DEP_CARRERA.

Un registro del tipo miembro *no puede existir en más de un conjunto de ocurrencia* de un tipo de conjunto en particular. En nuestro ejemplo un registro ESTUDIANTE puede relacionarse al menos a un DEPARTAMENTO y por tanto es miembro de al menos un conjunto de ocurrencia del tipo DEP_CARRERA.

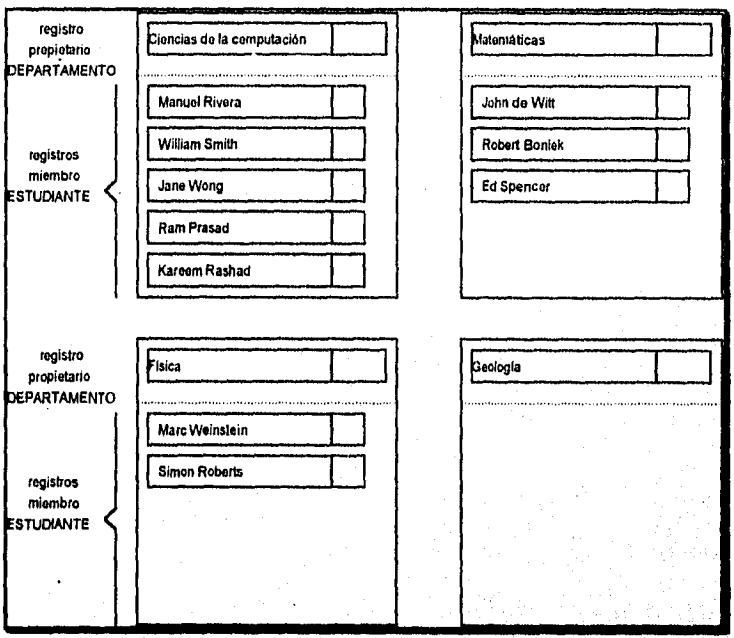


Figura 7.4 Cuatro instancias del conjunto DEP_CARRERA.

Un conjunto de ocurrencia puede identificarse ya sea por el *registro propietario* o *por cualquiera de los registros miembro*. La Figura 7.4 muestra cuatro conjuntos de ocurrencia (instancias) del tipo de conjunto DEP_CARRERA. Las cuatro instancias de conjunto de la Figura 7.4 pueden ser referidas como conjuntos de 'Ciencias de la Computación', 'Matemáticas', 'Física' y 'Geología'.

En el modelo de red, una instancia de conjunto *no es idéntica* al concepto de conjunto matemático. Existen dos diferencias principales:

- La instancia de conjunto tiene un *elemento distintivo* -el registro propietario- mientras que en un conjunto matemático no hay tal distinción entre los elementos de dicho conjunto.

- En el modelo de red, los registros miembro de un conjunto de instancias etán *ordenados*, mientras que en matemáticas el orden de elementos es inexistente. Para

distinguirlo del conjunto matemático al conjunto del modelo de red se le refiere algunas veces como un **conjunto de propiedad-acoplada** o **co-conjunto**.

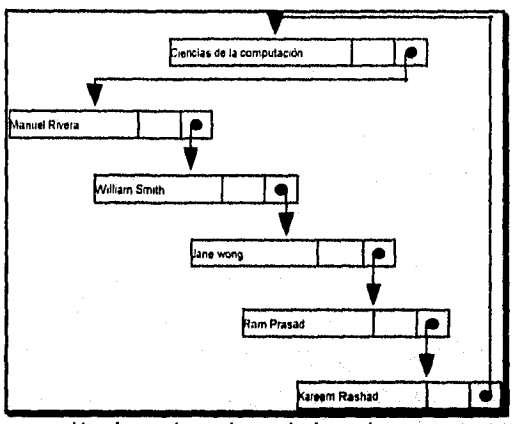


Figura 7.5 Representación alterna de una instancia de conjunto.

7.1.3 Tipos Especiales de Conjuntos

En el modelo de red CODASYL se permiten dos tipos especiales de conjuntos: conjuntos propiedad del SISTEMA y conjuntos multimiembro. En el reporte original CODASYL, no fue permitido un tercer tipo, llamado conjunto recursivo.

Conjuntos Propiedad del Sistema (Singular). Es un conjunto sin tipo de registro propietario, el propietario es el sistema. En el modelo de red los conjuntos propiedad del sistema sirven a dos propósitos principales:

- Proporcionan *puntos de entrada* a la base de datos vía los registros del tipo miembro especificado en el tipo de registro.
- Pueden usarse para *ordenar* los registros de un tipo dado usando las especificaciones de ordenamiento del conjunto. Un usuario puede accesar sus registros en diferentes órdenes especificando varios conjuntos propiedad del sistema sobre el mismo tipo de registro.

Conjuntos Multimiembro. Se usan en instancias donde el número de registros de un conjunto puede ser de *más de un tipo* de registro. No son soportados por la mayoría de los DBMSs jerárquicos. Para forzar la naturaleza de la relación 1:1, sigue siendo válido el argumento de que cada registro miembro puede aparecer en al menos un conjunto de ocurrencias.

Conjuntos Recursivos. Al tipo de conjunto en el cual el mismo tipo de registro juega el rol de propietario y miembro se le llama **recursivo**. Un ejemplo de relación recursiva 1:N es la relación SUPERVISION, la cual relaciona un supervisor de empleado a la lista de empleados directamente bajo su supervisión. En esta relación el tipo de registro EMPLEADO juega ambos roles: el de propietario del registro (empleado supervisor) y el de miembro del registro (empleados supervisados).

Con conjuntos recursivos, el mismo registro puede ser propietario de un conjunto de ocurrencias y miembro de otro, ambos conjuntos de ocurrencias son del mismo tipo de conjunto. Debido a este problema, se acostumbra representar un conjunto recursivo creando un tipo de registro adicional, el **enlace**. Para representar relaciones M:N se usa la misma técnica. La Figura 7.6(c) muestra la representación de la relación SUPERVISION, usando dos tipos de conjuntos y un tipo de registro enlace. En la Figura 7.6(c) el tipo de conjunto SUPERVISOR es realmente una relación 1:1; esto es, en cada ocurrencia de SUPERVISOR existirá al menos un registro miembro SUPERVISION. Podemos imaginar a cada registro enlace SUPERVISION como la representación de un empleado *en el rol de supervisor*. La representación directa del conjunto recursivo -usualmente prohibida en el modelo de red- se muestra en la Figura 7.6(d).

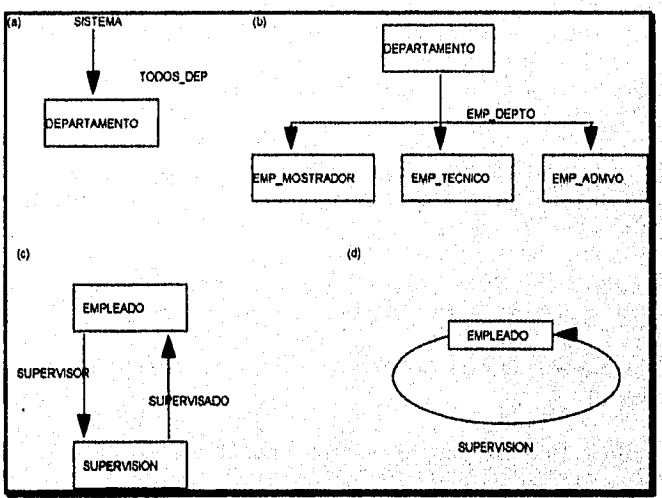


Figura 7.6 Tipos especiales de conjuntos. (a) Conjunto singular (propiedad-SISTEMA). (b) Conjunto multimiembro. (c) Conjunto recursivo SUPERVISION representado usando un tipo de registro enlace. (d) Representación de conjunto recursivo prohibido.

7.1.4 Representaciones de Instancias de Conjuntos Almacenados

Como se mostró en la Figura 7.5, una instancia de conjunto se representa comúnmente como un **anillo (lista circular enlazada)** que enlaza al registro propietario y a todos los

miembros del conjunto. A esta también se le llama **cadena circular**. La representación del anillo es simétrica con respecto a todos los registros; por tanto, para distinguir entre los registros propietario y miembros, el DBMS incluye un campo especial, llamado el **campo de tipo**, que tiene un valor distinto (asignado por el DBMS) para cada tipo de registro. Examinando el tipo de campo, el sistema puede decir si el registro es el propietario de la instancia o es uno de los miembros. Este tipo de campo está oculto al usuario y únicamente se usa por el DBMS.

Además del campo de tipo, el DBMS asigna automáticamente un **campo apuntador** a un tipo de registro para *cada tipo de conjunto en el cual él participa como propietario o miembro*. Un apuntador es usualmente llamado apuntador **NEXT** en un registro miembro y **FIRST** en un registro propietario debido a que estos apuntan al siguiente y primer registro miembro, respectivamente. El apuntador NEXT del *último registro miembro* en una ocurrencia del conjunto apunta de regreso al registro propietario. Si un registro del tipo miembro no participa en ninguna instancia del conjunto, su apuntador NEXT tiene un apuntador especial nil. Si un conjunto de ocurrencias tiene un propietario pero no registros miembros, el apuntador FIRST apunta de regreso al registro propietario o puede ser **nil**.

En general, un DBMS puede implementar conjuntos de varias formas. Sin embargo, la representación elegida debe permitir al DBMS hacer todas las operaciones siguientes:

- Dado un registro propietario, encontrar todos los registros miembros de la ocurrencia del conjunto.
- Dado un registro propietario, encontrar el primer, iésimo, o último miembro de registro de la ocurrencia del conjunto. Si no existe tal registro, dar una indicación de ese hecho.
- Dado un registro miembro, encontrar el siguiente (o previo) miembro de la ocurrencia del conjunto. Si no existe tal registro, dar una indicación de ese hecho.
- Dado un registro miembro, encontrar al registro propietario de la ocurrencia del conjunto.

7.1.5 Uso de Conjuntos para Representar Relaciones 1:1 y M:N

Un tipo de conjunto representa una relación 1:N entre dos tipos de registros. Esto significa que *un registro del tipo miembro puede aparecer en sólo una ocurrencia del conjunto*. En el modelo de red este argumento se lleva a cabo automáticamente por el DBMS.

Para representar una relación 1:1 entre dos tipos de registros usando un tipo de conjunto, debemos restringir a cada ocurrencia del conjunto a tener *un solo registro miembro*. El modelo de red no proporciona un forzamiento automático de este argumento, así el

programador debe checar que no se viole el argumento cada vez que se inserta un registro miembro en una ocurrencia del conjunto.

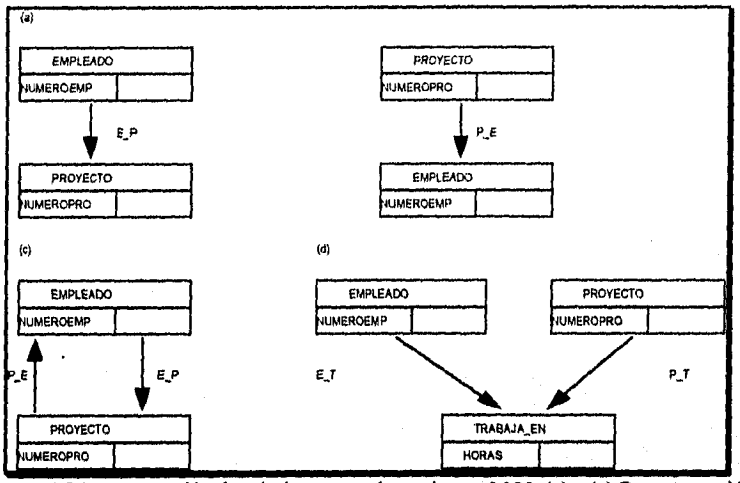


Figura 7.7 Representación de relaciones usando conjuntos M:N. (a) - (c) Representación incorrecta de relaciones M:N. (d) Representación correcta de una relación M:N usando tipo de registro enlace.

7.2 Argumentos del Modelo de Red

En esta sección discutiremos el "comportamiento" que se aplica a los miembros de conjuntos cuando se efectúan operaciones de inserción, borrado o actualización. Pueden especificarse varios argumentos sobre los miembros del conjunto. Estas se dividen usualmente en dos categorías principales, llamadas **opciones de inserción** y **opciones de retención** en terminología CODASYL. Estos argumentos se determinan durante el diseño de la base de datos sabiendo cómo se requiere que se *comporte* un conjunto cuando se inserten o borren miembros del registro. Los argumentos se especifican al DBMS cuando se declara la estructura de la base de datos, usando el lenguaje de definición de datos. No son posibles todas las combinaciones de argumentos. Primero discutiremos cada tipo de argumento y luego daremos las combinaciones permisibles.

7.2.1 Opciones (Argumentos) de Inserción Sobre Conjuntos

Los argumentos de inserción -u opciones, en terminología CODASYL- sobre los miembros de un conjunto especifican lo que sucede cuando se inserta un nuevo registro en la base de datos que es de tipo miembro. Un registro se inserta usando el comando STORE. Existen dos opciones:

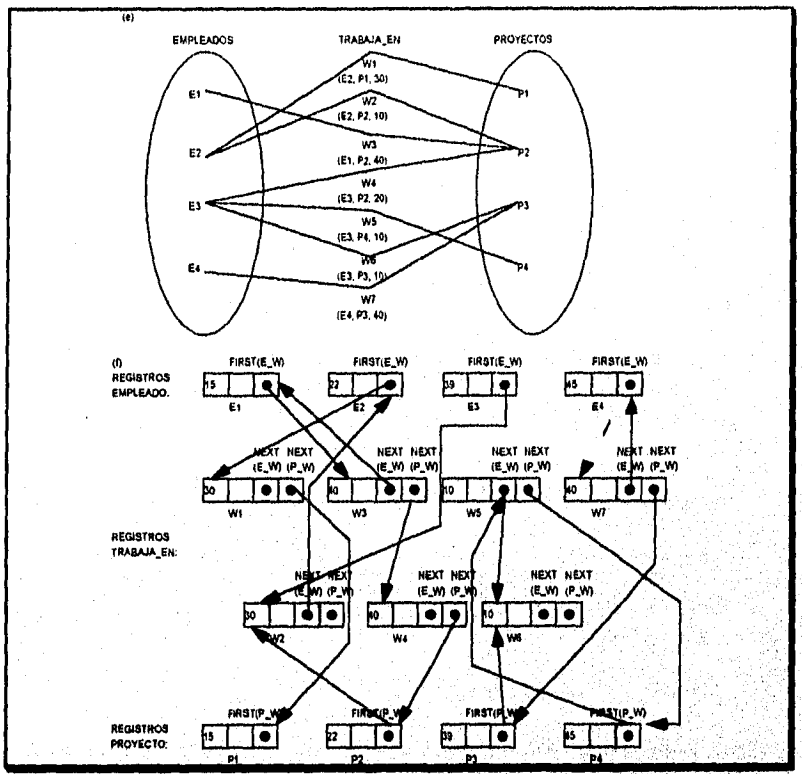


Figura 7.7 (e) Representación de algunas ocurrencias de relación M:N con "ocurrencias de enlace". (f) Algunas ocurrencias de los tipos de conjunto E_W y E_P y el tipo de registro de enlace TRABAJA_EN correspondiente a las instancias de relación M:N mostradas en la Figura 7.7(e).

- AUTOMATICO: El nuevo registro miembro *se conecta automáticamente* a una ocurrencia de conjunto apropiada cuando se inserta el registro.

- MANUAL: El nuevo registro no se conecta a ninguna ocurrencia del conjunto. Si se desea, el programador puede conectar explícitamente el registro a una ocurrencia de conjunto subsecuentemente, usando el comando CONNECT.

La opción de inserción AUTOMATIC se usa en situaciones donde deseamos insertar un miembro a una instancia automáticamente después del almacenamiento de ese registro en la base de datos. Se debe designar un criterio para *designar la instancia* de la cual se vuelve miembro cada registro nuevo.

7.2.2 Opciones (Argumentos) de Retención sobre Conjuntos

Los argumentos de retensión especifican si puede existir un registro miembro en la base de datos por sí mismo o si debe relacionarse a un propietario como miembro de alguna instancia. Existen tres opciones de retensión:

- OPCIONAL: Un miembro puede existir por sí mismo *sin ser* miembro de ninguna ocurrencia del conjunto. Puede conectarse y desconectarse a ocurrencias por medio de los comandos CONNECT y DISCONNECT del DML.

- MANDATORIO: *No puede* existir un miembro por sí mismo; *siempre* debe ser miembro en alguna ocurrencia del conjunto. Puede reconectarse en una operación sencilla de una ocurrencia a otra por medio del comando RECONNECT del DML.

- FIJO: Como en el MANDATORIO, *no puede* existir un miembro por sí mismo. Sin embargo, una vez que se inserta en una ocurrencia, es *fijo*; *no puede* reconectarse a otra ocurrencia.

En general, las opciones MANDATORY y FIXED se usan en situaciones donde un miembro no debe existir en la base de datos sin estar relacionado a un propietario a través de alguna ocurrencia.

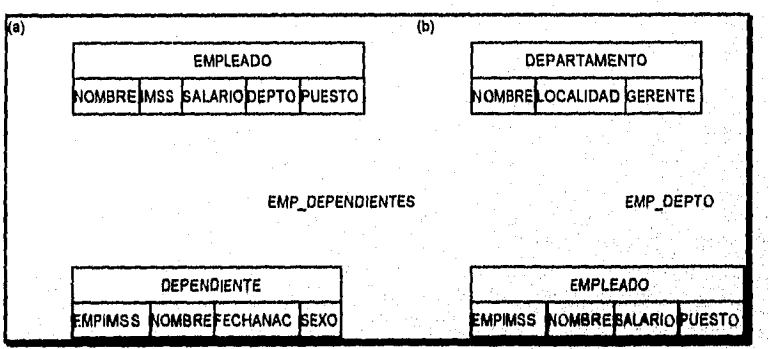


Figura 7.8 Diferentes opciones de conjunto. (a) Conjunto AUTOMATICO FIJO tipo EMP_DEPENDIENTES. (b) Conjunto AUTOMATICO MANDATORIO tipo EMP_DEPTO.

7.2.3 Combinaciones de Opciones de Inserción y Retensión

No todas las combinaciones de inserción y retención son útiles. Mientras que cualquier combinación de estas opciones es técnicamente válida, sólo tres combinaciones tienen sentido, y la mayoría de las implementaciones del modelo de red permiten sólo estas combinaciones "razonables": AUTOMATIC-FIXED, AUTOMATIC-MANDATORY, y MANUAL-OPTIONAL. También podemos imaginar aplicaciones donde puede ser muy

útil un conjunto AUTOMATIC-OPTIONAL -por ejemplo, cuando el miembro se conecta automáticamente a un propietario si se especifica un propietario en particular, de otra forma el nuevo registro miembro no se conecta a ninguna instancia. Estas combinaciones se globalizan en la Tabla 7.1.

Opción de Inserción \ Opción de Retensión	OPCIONAL	MANDATORIO	FIJO
MANUAL	El programa de aplicación está a cargo de insertar miembros a la ocurrencia. Puede CONNECT, DISCONNECT, RECONNECT.	No es muy útil.	No es muy útil.
AUTOMATICO	El DBMS inserta un nuevo miembro a la ocurrencia automáticamente. Puede CONNECT, DISCONNECT, RECONNECT.	El DBMS inserta un nuevo miembro a la ocurrencia automáticamente. Puede RECONNECT miembros a diferentes propietarios.	El DBMS inserta un nuevo miembro a la ocurrencia automáticamente. No puede RECONNECT miembros a un propietariodiferente.

7.2.4 Opciones de Ordenamiento de Conjuntos

Los miembros en una instancia pueden ordenarse de varias formas. El orden puede basarse sobre un campo o controlarse por el tiempo secuencia de inserción de nuevos miembros. Las opciones disponibles para ordenar pueden globalizarse como sigue:

- Sorteado por un campo de ordenamiento: Los valores de uno o más campos del miembro se usan para ordenar los registros miembro dentro de *cada ocurrencia* en orden ascendente o descendente.
- Default del Sistema: Un nuevo miembro se inserta en una posición arbitraria determinada por el sistema.
- Primero o último: *En el momento en que se inserta* un nuevo miembro se vuelve el primer o último registro en la ocurrencia.
- Próximo o anterior: El nuevo miembro se inserta después o antes del registro actual de la ocurrencia.

Las opciones deseadas de inserción, retensión, y ordenamiento se especifican en el lenguaje de definición de datos cuando se declara el tipo de conjunto.

7.3 Definición de Datos en el Modelo de Red

Después de diseñar el esquema de una base de datos de red, debemos declarar todos los tipos de registros, conjuntos, definiciones de elementos de datos, y argumentos al esquema para el DBMS. Para este propósito se usa el DDL.

7.3.1 Tipos de Registro y Declaraciones de Elementos de Datos

Las declaraciones DDL para los tipos de registro del esquema COMPAÑIA mostrados en la Figura 7.9 se muestran en la Figura 7.7(a). A cada tipo de registro se le asigna un nombre usando la cláusula **RECORD NAME IS**. Se especifica un formato (tipo de dato) para cada uno de sus elementos (campos), junto con cualquier argumento sobre los elementos. Los tipos de datos usuales dependen de los tipos definibles en el lenguaje de programación huésped. Asumiremos que están disponibles cadenas de caracteres, enteros, y números formateados.

Para especificar argumentos sobre campos llave, o en combinaciones de campos que no pueden tener el mismo valor en más de un registro de un tipo, se usa la cláusula **DUPLICATES ARE NOT ALLOWED**. Se dispone de argumentos adicionales sobre campos que incluyen argumentos sobre los valores que un campo numérico puede tomar, usando la cláusula **CHECK**.

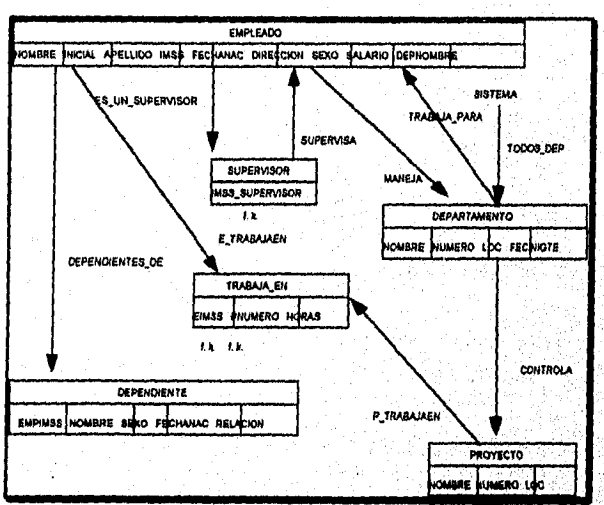


Figura 7.9 Esquema de de una red para la base de datos COMPAÑIA.

```
SCHEMA NAME IS COMPANY

RECORD NAME IS EMPLEADO
        DUPLICATES ARE NOT ALLOWED FOR IMSS
        DUPLICATES ARE NOT ALLOWED FOR NOMBRE, INIC, APELLIDO
                APELLIDO        TYPE IS     CHARACTER 15
                INIC            TYPE IS     CHARACTER 1
                APELLIDO        TYPE IS     CHARACTER 15
                IMSS            TYPE IS     CHARACTER 9
                FECHANAC        TYPE IS     CHARACTER 9
                DIRECCION       TYPE IS     CHARACTER 30
                SEXO            TYPE IS     CHARACTER 1
                SALARIO         TYPE IS     CHARACTER 10
                DEPNOMBRE       TYPE IS     CHARACTER 15

RECORD NAME IS DEPTO
        DUPLICATES ARE NOT ALLOWED FOR NOMBRE
        DUPLICATES ARE NOT ALLOWED FOR NUMERO
                NOMBRE          TYPE IS     CHARACTER 15
                NUMERO          TYPE IS     NUMERIC INTEGER
                LOC             TYPE IS     CHARACTER 15 VECTOR
                FECINIGTE       TYPE IS     CHARACTER 9

RECORD NAME IS PROYECTO
        DUPLICATES ARE NOT ALLOWED FOR NOMBRE
        DUPLICATES ARE NOT ALLOWED FOR NUMERO
                NOMBRE          TYPE IS     CHARACTER 15
                NUMERO          TYPE IS     NUMERIC INTEGER
                LOC             TYPE IS     CHARACTER 15

RECORD NAME IS TRABAJA_EN
        DUPLICATES ARE NOT ALLOWED FOR EIMSS, PNUMERO
                EIMSS           TYPE IS     CHARACTER 9
                PNUMERO         TYPE IS     NUMERIC INTEGER
                HORAS           TYPE IS     NUMERIC(4,1)

RECORD NAME IS SUPERVISOR
        DUPLICATES ARE NOT ALLOWED FOR IMSS_SUPERVISOR
                IMSS_SUPERVISOR TYPE IS     CHARACTER 9

RECORD NAME IS DEPENDIENTE
        DUPLICATES ARE NOT ALLOWED FOR IMSSEMP, NOMBRE
                IMSSEMP         TYPE IS     CHARACTER 9
                NOMBRE          TYPE IS     CHARACTER 15
                SEXO            TYPE IS     CHARACTER 1
                FECHANAC        TYPE IS     CHARACTER 9
                RELACION        TYPE IS     CHARACTER 10

        SET SELECTION IS BY APPLICATION

SET NAME IS TODOS_DEP
        OWNER IS SYSTEM
        ORDER IS SORTED BY DEFINED KEYS
        MEMBER IS DEPTO
        KEY IS ASCENDING NOMBRE

SET NAME IS TRABAJA_PARA
        OWNER IS DEPTO
        ORDER IS SORTED BY DEFINED KEYS
        MEMBER IS EMPLEADO
        INSERTION IS MANUAL
        RETENTION IS OPTIONAL
        KEY IS ASCENDING APELLIDO, NOMBRE, INIC
        CHECK IS DEPNOM IN EMPLEADO = NOMBRE IN DEPTO
```

```
SET NAME IS CONTROLA
        OWNER IS DEPTO
        ORDER IS SORTED BY DEFINED KEYS
        MEMBER IS PROYECTO
        INSERTION IS AUTOMATIC
        RETENTION IS MANDATORY
        KEY IS ASCENDING NOMBRE

SET NAME IS MANEJA
        OWNER IS EMPLEADO
        ORDER IS SYSTEM DEFAULT
        MEMBER IS DEPTO
        INSERTION IS AUTOMATIC
        RETENTION IS MANDATORY
        SET SELECTION IS BY APPLICATION

SET NAME IS P_TRABAJAEN
        OWNER IS PROYECTO
        ORDER IS SYSTEM DEFAULT
        DUPLICATES ARE NOT ALLOWED
        MEMBER IS TRABAJA_EN
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        KEY IS EIMSS
        SET SELECTION IS STRUCTURAL NUMERO IN PROYECTO = PNUMERO IN TRABAJA_EN

SET NAME IS E_TRABAJAEN
        OWNER IS EMPLEADO
        ORDER IS SYSTEM DEFAULT
        DUPLICATES ARE NOT ALLOWED
        MEMBER IS TRABAJA_EN
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        KEY IS PNUMERO
        SET SELECTION IS STRUCTURAL IMSS IN EMPLEADO = EIMSS IN TRABAJA_EN

SET NAME IS SUPERVISA
        OWNER IS SUPERVISOR
        ORDER IS BY DEFINED KEY
        DUPLICATES ARE NOT ALLOWED
        MEMBER IS EMPLEADO
        INSERTION IS MANUAL
        RETENTION IS OPTIONAL
        KEY IS APELLIDO, INIC, NOMBRE

SET NAME IS ES_UN_SUPERVISOR
        OWNER IS EMPLEADO
        ORDER IS SYSTEM DEFAULT
        DUPLICATES ARE NOT ALLOWED
        MEMBER IS SUPERVISOR
        INSERTION IS AUTOMATIC
        RETENTION IS MANDATORY
        KEY IS IMSS_SUPERVISOR
        SET SELECTION IS BY VALUE OF IMSS IN EMPLEADO
        CHECK IS IMSS_SUPERVISOR IN SUPERVISION = IMSS IN EMPLEADO

SET NAME IS DEPENDIENTES_DE
        OWNER IS EMPLEADO
        ORDER IS BY DEFINED KEY
        DUPLICATES ARE NOT ALLOWED
        MEMBER IS DEPENDIENTE
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        KEY IS ASCENDING NOMBRE
        SET SELECTION IS STRUCTURAL IMSS IN EMPLEADO = EIMSS IN DEPENDIENTE
```

Figura 7.10 (a) Declaraciones de tipo registro para el esquema de la Figura 7.9.

7.3.2 Declaraciones de Conjuntos y Opciones de Selección

A cada conjunto se le asigna un nombre usando la cláusula **SET NAME IS**. Las opciones de inserción y retención (argumentos), se especifican para cada conjunto usando las cláusulas **INSERTION IS** y **RETENTION IS**. Para este propósito se usa la cláusula **SET SELECTION**. Tres métodos comúnes de especificar SET SELECTION son los siguientes:

- SET SELECTION IS STRUCTURAL: Podemos especificar la selección de conjuntos por valores de dos campos que deben corresponder -un campo del tipo propietario y uno miembro. A esto se le llama argumento estructural en terminología de red.

- SET SELECTION BY APPLICATION: La ocurrencia se determina vía el programa de aplicación, el cual debe hacer que la ocurrencia deseada sea la actual antes de almacenar el nuevo registro. El nuevo miembro se conecta automáticamente a la ocurrencia actual. Almacenando un nuevo DEPARTAMENTO *automáticamente* conecta a su registro EMPLEADO gerente como propietario.

- SET SELECTION IS BY VALUE OF <nombre del campo> IN <nombre registro>: Una tercera opción es especificar un campo del tipo propietario cuyo valor se use para especificar una ocurrencia identificando al registro propietario del conjunto. El campo especificado en el propietario debe tener el argumento DUPLICATES ARE NOT ALLOWED de tal forma que identifique a un propietario único y por tanto una sola ocurrencia.

Otra opción para conjuntos es especificar cómo se ordenarán los registros miembro en una instancia. La cláusula **ORDER IS** se usa para este propósito, algunas veces junto con **KEY IS**. Las opciones de la cláusula ORDER IS incluyen a las siguientes:

- ORDER IS SORTED BY DEFINED KEYS: Usaremos la cláusula KEY IS para especificar uno o más campos del registro miembro; el sistema usa los valores de estos campos para ordenar los miembros dentro de cada instancia. La cláusula KEY IS también especifica si los registros deben ordenarse ASCENDENTE o DESCENDENTEMENTE.

- ORDER IS FIRST (o LAST): Un nuevo miembro se inserta al principio (o final) de la ocurrencia.

- ORDER IS BY SYSTEM DEFAULT: No se especifica ningún orden en particular sobre los miembros de una instancia.

- ORDER IS NEXT (o PRIOR): Un nuevo miembro se inserta inmediatamente después (o antes) del registro del conjunto actual.

Otra cláusula que trabaja junto con KEY IS es la cláusula **DUPLICATES ARE NOT ALLOWED**. Esta combinación especifica que a ningún par de registros miembro *dentro de una ocurrencia* se les permite tener los mismos valores para sus campos declarados como llaves.

Finalmente, consideremos la cláusula **CHECK**, la cual se usa para especificar argumento estructural entre el propietario y los registros miembro de una ocurrencia. Esto se usa con conjuntos MANUAL para especificar la condición de que algunos campos de un registro miembro deben tener los mismos valores que algunos campos del propietario. Si se hace el intento de conectar un miembro que no satisface la condición de CHECK, el sistema genera una condición de excepción y no conecta al miembro.

7.4 Diseño de Bases de Datos de Red Usando Mapeo ER-Red

Ahora mostraremos cómo puede mapearse un diseño de base de datos conceptual especificado como un esquema ER a un esquema de red. El esquema de red COMPAÑIA mostrado en la Figura 7.9, puede derivarse del esquema ER de la Figura 3.2 siguiendo el procedimiento de mapeo general.

Paso 1: Entidades Regulares: Para cada tipo de entidad regular E en el esquema ER, crear un registro R en el esquema de red. Todos los atributos simples (o compuestos) de E se incluyen como campos simples (o compuestos) de R. Todos los atributos multivaluados de E se incluyen como campos vector o grupos de repetición de R.

En nuestro ejemplo creamos los tipos de registro EMPLEADO, DEPARTAMENTO, y PROYECTO e incluyen a todos sus campos mostrados en la Figura 7.9 excepto los campos marcados por fk (llave externa) o un * (atributo de relación). Notemos que el campo LOCALIDADES del tipo de registro DEPARTAMENTO es un campo vector debido a que representa un atributo multivaluado.

Paso 2: Entidades Débiles: Para cada tipo de entidad débil WE con el propietario identificando al tipo de entidad IE, crear un registro tipo W para representar WE, haciendo a W el registro miembro en un tipo de conjunto que relaciona a W al registro que representa a IE como propietario.

En la Figura 7.9 se crea un registro DEPENDIENTE y se hace miembro del conjunto DEPENDE_DE, el cual es propiedad de EMPLEADO. Se duplicó la llave IMSS de EMPLEADO en DEPENDIENTE y se le llamó EMPIMSS.

PASO 3: Relaciones Uno a Uno y Uno a Muchos: Para cada relación binaria no recursiva 1:1 o 1:N R entre entidades E1 y E2, crear un conjunto que relacione a los registros S1 y S2 que represente a E1 y E2, respectivamente. Para una relación 1:1, se elige arbitrariamente a S1 o a S2 como propietario y al otro como miembro; sin embrago, es preferible elegir como miembro a un registro que represente una participación total en la relación. Otra opción para mapear una relación binaria 1:1 R entre E1 y E2 es crear un

registro sencillo S que carga a E1, E2 y R e incluir todos sus atributos; esto es útil si ambas participaciones de E1 y E2 en R son totales y E1 y E2 no participan en otros numerosos tipos de relaciones.

Para una relación 1:N, se elige como propietario al registro S1 que representa a la entidad E1 en el lado-1 de la relación, y como miembro al registro S2 que representa a la entidad E2 del lado N de la relación. Cualquier atributo de la relación R se incluye como campo en el registro *miembro* S2.

En nuestro ejemplo representamos la relación 1:1 MANEJA de la Figura 3.2 por el conjunto MANEJA, y elegimos a DEPARTAMENTO como registro miembro debido a que su participación es total. El atributo FechaIni de MANEJA se vuelve un campo FECINIGTE del registro miembro DEPARTAMENTO. Las dos relaciones no recursivas 1:N de la Figura 3.2, TRABAJA_PARA y CONTROLA, se representan por los dos conjuntos TRABAJA_PARA y CONTROLA en la Figura 7.9. Para el conjunto TRABAJA_EN, elegimos repetir un campo llave único NOMBRE del registro propietario DEPTO en el registro miembro EMPLEADO, y lo llamamos NOMDEP. Para el conjunto CONTROL declinamos repetir cualquier campo llave. En general, podría repetirse un campo único de un propietario en el miembro.

PASO 4: Relaciones Binarias Muchos a Muchos: Para cada relación binaria M:N R entre entidades E1 y E2, crear un registro de enlace X y hacerlo miembro en los dos conjuntos. Los conjuntos propietarios son S1 y S2 que representan a E1 y E2. Cualquier atributo de R se hace campo de X. El diseñador puede duplicar arbitrariamente los campos únicos (llave) del registro propietario como campos de X.

En la Figura 7.9 creamos el registro de enlace TRABAJA_EN para representar la relación M:N TRABAJA_EN, e incluimos a HORAS como su campo. Se crearon dos conjuntos E_TRABAJAEN y P_TRABAJAEN con TRABAJA_EN como registro miembro. Elegimos duplicar los campos llave únicos IMSS y NUMERO del registro propietario EMPLEADO y PROYECTO en TRABAJA_EN, llamándolos EIMSS y PNUMERO, respectivamente.

PASO 5: Relaciones Recursivas: Para cada relación binaria recursiva 1:1 o 1:N en la cual la entidad E participa en ambos roles, crear un registro de enlace D y dos conjuntos para relacionar a D con el registro X que representa a E. Uno o ambos conjuntos serán argumentados a tener instancias con un sólo registro miembro -uno en el caso de relación 1:N, y ambos en el caso de relación 1:1.

En la Figura 3.2 tenemos una relación recursiva 1:N SUPERVISION. Creamos el registro de enlace SUPERVISOR y los dos conjuntos ES_UN_SUPERVISOR y SUPERVISA. El conjunto ES_UN_SUPERVISOR está argumentado a tener registros miembro sencillos por los programas de actualización de la base de datos en sus instancias. Podemos imaginar a cada registro miembro SUPERVISOR del conjunto ES_UN_SUPERVISOR como representante de su registro propietario EMPLEADO *en un rol supervisor*. El

conjunto SUPERVISA se usa para relacionar al registro SUPERVISOR con todos los EMPLEADO que representan a los empleados que están bajo su supervisión directa.

PASO 6: Relaciones n-arias: Para cada relación n-aria R, n > 2, crear un registro de enlace X y hacerlo miembro en conjuntos tipo n. El propietario de cada conjunto es el registro que representa una de las entidades participantes en la relación R. Cualquier atributo de R se hace campo de X. El diseñador puede duplicar arbitrariamente los campos únicos (llave) de los registros propietario como campos de X.

Por ejemplo, consideremos la relación tipo PROVEE en el modelo ER, como se muestra en la Figura 7.11(a). Este puede mapear al registro PROVEE y a los tres conjuntos mostrados en la Figura 7.11(b), donde elegimos no duplicar ningún campo de los propietarios.

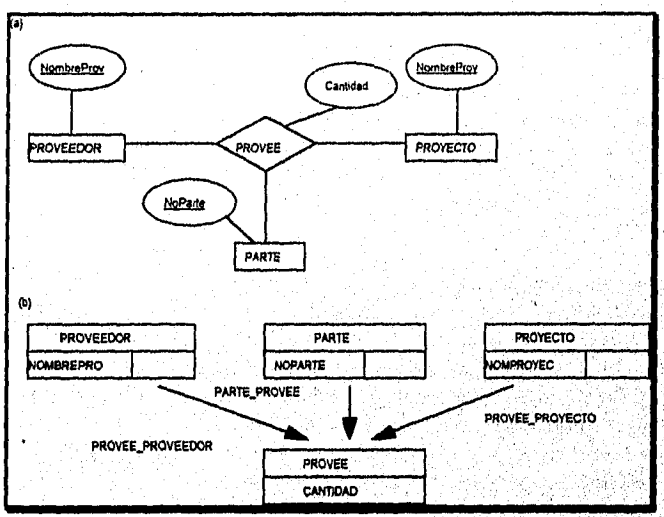


Figura 7.11 Mapeo de la relación n-aria (n=3) PROVEE del modelo ER al modelo de red. (a) El modelo ER. (b) El modelo de red.

7.5 Programación de una Base de Datos de Red

En esta sección discutiremos cómo escribir programas que manipulen a una base de datos de red -incluyendo tareas tales como la búsqueda y recuperación de registros; inserción, borrado y modificación; y conexión y desconexión de registros de las ocurrencias. Para este propósito se usa un **lenguaje manipulador de datos**. El DML asociado con el modelo de red consta de comandos de registro a la vez que están contenidos en un lenguaje de propósito general llamado el **lenguaje huésped**. En la práctica, los lenguajes huésped más comúnmente usados son el COBOL y el PL/1. En nuestros ejemplos, sin

embargo, escribiremos segmentos de programa en PASCAL aumentados con comandos DDL de red.

7.5.1 Conceptos Básicos para la Manipulación de Bases de Datos de Red

El sistema de base de datos y el lenguaje de programación huésped son dos sistemas de software independientes que se enlazan por una interface en común y se comunican sólo a través de esa interface. Debido a que los comandos DML son de un registro a la vez, es necesario identificar registros específicos de la base de datos como **registros actuales**. El DBMS conoce el número de registros actuales y ocurrencias de conjuntos por medio de un mecanismo conocido como **indicador de actualidad**. Además, el lenguaje de programación huésped necesita variables locales de programa para almacenar los registros de diferentes tipos de tal forma que sus contenidos puedan manipularse por el programa huésped. Al conjunto de estas variables locales se les refiere como **área de trabajo del usuario (UWA)**.

El Area de Trabajo del Usuario (UWA). Es un conjunto de variables de programa, declaradas en el lenguaje huésped, para comunicar el contenido de registros individuales entre el DBMS y el programa huésped. Para cada tipo de registro en el esquema, debe declararse una variable correspondiente con el mismo formato. Se acostumbra usar los mismos nombres de registro y los mismos campos en las variables UWA que en el esquema de la base de datos.

Indicadores de Actualidad. En el DML de red, las recuperaciones y actualizaciones se manejan moviéndose o **navegando** a través de los registros de la base de datos; por lo tanto, es crítico guardar un trazado de búsqueda. Los indicadores de actualidad son un medio de guardar la pista de los registros y ocurrencias más recientemente accesados en el DBMS. Juegan el rol de retenedores de posición de tal forma que podemos procesar nuevos registros empezando desde los más recientemente accesados hasta recuperar todos los registros que contengan la información que necesitamos. Cada indicador de actualidad puede visualizarse como un apuntador (o dirección) que apunta a un registro de la base de datos. En un DBMS de red, se usan varios indicadores de ocurrencia:

- **Tipo de registro actual:** Para cada tipo de registro, el DBMS guarda la posición del registro más recientemente accesado de ese tipo. Si aún no ha sido accesado ningún registro de ese tipo, el registro actual es indefinido.

- **Tipo de conjunto actual:** Para cada tipo de conjunto en el esquema, el DBMS guarda la posición de la ocurrencia más recientemente accesada de ese tipo de conjunto. Se especifica por un sólo registro de ese conjunto, el cual es el *propietario o uno de los miembros*. Por tanto, el conjunto actual apunta a un registro, aunque se use para guardar la pista de una ocurrencia. Si el programa no ha accesado a ningún registro de ese tipo de conjunto, el conjunto actual es indefinido.

embargo, escribiremos segmentos de programa en PASCAL aumentados con comandos DDL de red.

7.5.1 Conceptos Básicos para la Manipulación de Bases de Datos de Red

El sistema de base de datos y el lenguaje de programación huésped son dos sistemas de software independientes que se enlazan por una interface en común y se comunican sólo a través de esa interface. Debido a que los comandos DML son de un registro a la vez, es necesario identificar registros específicos de la base de datos como **registros actuales**. El DBMS conoce el número de registros actuales y ocurrencias de conjuntos por medio de un mecanismo conocido como **indicador de actualidad**. Además, el lenguaje de programación huésped necesita variables locales de programa para almacenar los registros de diferentes tipos de tal forma que sus contenidos puedan manipularse por el programa huésped. Al conjunto de estas variables locales se les refiere como **área de trabajo del usuario (UWA)**.

El Area de Trabajo del Usuario (UWA). Es un conjunto de variables de programa, declaradas en el lenguaje huésped, para comunicar el contenido de registros individuales entre el DBMS y el programa huésped. Para cada tipo de registro en el esquema, debe declararse una variable correspondiente con el mismo formato. Se acostumbra usar los mismos nombres de registro y los mismos campos en las variables UWA que en el esquema de la base de datos.

Indicadores de Actualidad. En el DML de red, las recuperaciones y actualizaciones se manejan moviéndose o **navegando** a través de los registros de la base de datos; por lo tanto, es crítico guardar un trazado de búsqueda. Los indicadores de actualidad son un medio de guardar la pista de los registros y ocurrencias más recientemente accesados en el DBMS. Juegan el rol de retenedores de posición de tal forma que podemos procesar nuevos registros empezando desde los más recientemente accesados hasta recuperar todos los registros que contengan la información que necesitamos. Cada indicador de actualidad puede visualizarse como un apuntador (o dirección) que apunta a un registro de la base de datos. En un DBMS de red, se usan varios indicadores de ocurrencia:

- **Tipo de registro actual:** Para cada tipo de registro, el DBMS guarda la posición del registro más recientemente accesado de ese tipo. Si aún no ha sido accesado ningún registro de ese tipo, el registro actual es indefinido.

- **Tipo de conjunto actual:** Para cada tipo de conjunto en el esquema, el DBMS guarda la posición de la ocurrencia más recientemente accesada de ese tipo de conjunto. Se especifica por un sólo registro de ese conjunto, el cual es el *propietario o uno de los miembros*. Por tanto, el conjunto actual apunta a un registro, aunque se use para guardar la pista de una ocurrencia. Si el programa no ha accesado a ningún registro de ese tipo de conjunto, el conjunto actual es indefinido.

- **Unidad de corrida actual:** Es un programa de acceso a la base de datos que se ejecuta en el sistema de cómputo. Para cada unidad de corrida, el CRU guarda pista del registro más recientemente accesado por el programa; este registro puede ser de *cualquier* tipo en la base de datos.

Cada vez que un programa ejecuta un comando DML, los indicadores de actualidad de registro y conjunto afectados por ese comando se actualizan por el DBMS. Muchos comandos DML afectan y dependen de los indicadores de actualidad.

Indicadores de Status. Varios **indicadores de status** retornan una indicación de éxito o falla después de ejecutar cada comando DML. El programa puede checar los valores de estos indicadores y tomar la acción apropiada -ya sea continuar la ejecución o transferir a una rutina de manejo de error.

A la variable principal de status la llamamos DB_STATUS y asumimos que está declarada implícitamente en el programa huésped. Después de cada comando DML, el valor de DB_STATUS indica si el comando fue exitoso o si ocurrió una excepción o error. La excepción más común que ocurre es el FIN_DE_CONJUNTO (EOS).

7.5.2 Lenguaje de Manipulación de Datos de Red (DML)

A los comandos para manipular a una base de datos de red se les llaman DML de red. Los comandos DDL pueden agruparse en comandos de navegación, de recuperación y de actualización. Los comandos de **navegación** se usan para colocar los indicadores de actualidad a registros y ocurrencias de conjuntos específicos en la base de datos. Los comandos de **recuperación** recuperan al registro actual de la unidad de corrida (CRU). Los comandos de **actualización** pueden dividirse en dos subgrupos -uno para la actualización de registros, y el otro para actualizar ocurrencias de conjuntos. Los comandos de actualización de registros se usan para almacenar nuevos registros, borrar registros no deseados, y modificar valores de campos, mientras que los comandos de actualización de conjuntos se usan para conectar o desconectar un miembro en una ocurrencia o para mover un miembro de una ocurrencia a otra.

En nuestros ejemplos a menudo necesitamos asignar valores a campos de variables UWA PASCAL. Usaremos la notación de asignación de PASCAL. Por ejemplo, para colocar los valores NOMBRE y APELLIDO de la variable UWA EMPLEADO a "John Smith", escribimos:

```
EMPLEADO.NOMBRE:='John'; EMPLEADO.APELLIDO:='Smith';
```

7.5.3 Comandos DML para Recuperación y Navegación

El comando DML para recuperar un registro es GET. Antes de hablar del comando GET, el programa debe especificar el registro que quiere recuperar como el CRU, usando los

comandos navegacionales FIND; primero discutiremos el uso de FIND en la localización de instancias de un registro y luego las variantes para procesar ocurrencias de conjuntos.

Comandos DML para Localizar Registros de Tipo. Existen dos variantes principales del comando FIND para localizar un registro de un cierto tipo y hacer a ese registro el CRU y registro actual. Como veremos, también pueden afectarse otros indicadores de actualidad. El formato de estos dos comandos es como sigue, donde las partes opcionales del comando se encierran entre corchetes, [...]:

- **FIND ANY** <nombre registro> [**USING** <lista de campos>]
- **FIND DUPLICATE** <nombre registro> [**USING** <lista de campos>]

Tabla 7.2 Sumario de Comandos DML de Red

(RECUPERACION)	
GET	RECUPERA LA UNIDAD ACTUAL DE CORRIDA (CRU) EN LA VARIABLE CORRESPONDIENTE UWA
(NAVEGACION)	
FIND	RESETEA LOS INDICADORES DE ACTUALIDAD; SIEMPRE COLOCA AL CRU; TAMBIEN COLOCA A LOS INDICADORES DE ACTUALIDAD DE REGISTROS Y CONJUNTOS INVOLUCRADOS. EXISTEN MUCHAS VARIANTES DE FIND.
(ACTUALIZACION DE REGISTROS)	
STORE	ALMACENA AL NUEVO REGISTRO EN LA BASE DE DATOS Y LO HACE EL CRU
ERASE	BORRA DE LA BASE DE DATOS AL REGISTRO QUE ESTE EN EL CRU
MODIFY	MODIFICA ALGUNOS CAMPOS DEL REGISTRO QUE ESTA EN EL CRU
(ACTUALIZA CONJUNTOS)	
CONNECT	CONECTA UN MIEMBRO (EL CRU) A UNA INSTANCIA
DISCONNECT	ELIMINA UN MIEMBRO (EL CRU) DE UNA INSTANCIA
RECONNECT	MUEVE UN MIEMBRO (EL CRU) DE UNA INSTANCIA A OTRA

Ahora ilustraremos el uso de estos comandos con ejemplos. Para recuperar el registro EMPLEADO cuyo nombre es John Smith e imprimir su salario, podemos escribir EX1:

```
EX1:  EMPLEADO.NOMBRE:='John'; EMPLEADO.APELLIDO:='Smith';
      $FIND ANY EMPLEADO USING NOMBRE, APELLIDO;
      if DB_STATUS = 0
            then begin
            $GET EMPLEADO;
            writeln(EMPLEADO.SALARIO)
            end
            else writeln('no se encontró registro');
```

El comando **FIND ANY** encuentra al *primer* registro en la base de datos del <nombre de registro> especificado de tal forma que los valores de campo del registro correspondan con los valores inicializados anteriormente en los campos UWA correspondientes especificados en la cláusula del comando **USING**.

La instrucción FIND no solo coloca al CRU sino que también coloca indicadores de actualidad adicionales -esto es, para aquéllos registros cuyo nombre se especifica en el comando y para cualquier conjunto en el cual ese registro participa como propietario o miembro. Sin embargo, el comando GET siempre recupera el CRU *el cual puede que no sea igual al tipo de registro actual.*

Si más de un registro satisface nuestra búsqueda y deseamos recuperar a todos ellos, debemos escribir un ciclo en el lenguaje de programación huésped. Por ejemplo, para recuperar todos los EMPLEADOS que trabajan en el departamento de Investigación e imprimir sus nombres, podemos escribir EX2.

```
EX2:  EMPLEADO.NOMDEP:='Investigación';
      $FIND ANY EMPLEADO USING NOMDEP;
      while DB_STATUS = 0 do
            begin
            $GET EMPLEADO;
            writeln(EMPLEADO.NOMBRE, ", EMPLEADO.APELLIDO);
            $FIND DUPLICATE EMPLEADO USING NOMDEP
            end;
```

El comando FIND DUPLICATE encuentra al registro siguiente (o duplicado), empezando a partir del registro actual, que satisface la búsqueda. No podemos usar FIND ANY, porque siempre localiza al primer registro que satisface la búsqueda. El sistema busca los registros EMPLEADO fisicamente en el orden en el cual están almacenados. Una vez que han sido checados todos los registros en el ciclo while, el sistema pone al DB_STATUS en la condición de excepción "no más registros encontrados" y el ciclo termina.

Comandos DML para Procesamiento de Conjuntos. Para el procesamiento de conjuntos, tenemos las siguientes variantes de FIND:

- FIND (FIRST | NEXT| PRIOR| LAST| ...) <nombre registro>
  WITHIN <nombre conjunto> [using <NOMBRES DE CAMPO>]

- FIND OWNER WITHIN <nombre conjunto>

Una vez que hemos establecido una ocurrencia del conjunto, podemos usar el comando FIND para localizar varios registros que participen en la ocurrencia. Podemos localizar al registro propietario o a uno de los miembros y hacer de ese registro el CRU. Para localizar al propietario usamos **FIND OWNER** y uno de **FIND FIRST**, **FIND NEXT**, **FIND LAST** o **FIND PRIOR** para localizar al primer miembro, siguiente, último o anterior de la instancia, respectivamente.

El siguiente query va a imprimir los nombres de los empleados que trabajan en el departamento de Investigación alfabéticamente por apellido, lo cual se muestra en EX3.

Se recupera primero al registro con DEPTO Investigación y luego a los miembros EMPLEADO propiedad de ese registro vía el conjunto TRABAJA_PARA.

```
EX3:  DEP.NOMBRE:='Investigación';
      $FIND ANY DEPTO USING NOMBRE;
      if DB_STATUS=0 then
            begin
            $FIND FIRST EMPLEADO WITHIN TRABAJA_PARA;
            while DB_STATUS=0 do
                  begin
                  $GET EMPLEADO;
                  writeln(EMPLEADO.APELLIDO,",EMPLEADO.NOMBRE);
                  FIND NEXT EMPLEADO WITHIN TRABAJA_PARA
            end
      end;
```

El siguiente ejemplo ilustra el uso de FIND OWNER. El query va a imprimir el nombre del proyecto, número, y horas a la semana en cada proyecto que trabajó John Smith (asumiendo que sólo hay una persona con este nombre). El comando FIND ANY coloca al CRU así como al registro actual de EMPLEADO y al de E_TRABAJAEN. Luego hacemos un ciclo a través de cada miembro de TRABAJA_EN en el conjunto de actualidad E_TRABAJAEN, y dentro de cada ciclo encontramos al PROYECTO que posee al registro TRABAJA_EN vía el conjunto P_TRABAJAEN, usando el comando FIND OWNER. Notemos que no tenemos que checar el DB_STATUS después del comando FIND OWNER, debido a que la opción de retensión para el conjunto P_TRABAJAEN es FIXED, así cada registro TRABAJA_EN debe pertenecer a una instancia de P_TRABAJAEN:

```
EX4:  EMPLEADO.NOMBRE:='John'; EMPLEADO.APELLIDO:='Smith';
      $FIND ANY EMPLEADO USING NOMBRE, APELLIDO;
      if DB_STATUS = 0 then
            begin
            $FIND FIRST TRABAJA_EN WITHIN E_TRABAJAEN;
            while DB_STATUS=0 do
                  begin
                  $GET TRABAJA_EN;
                  $FIND OWNER WITHIN P_TRABAJAEN;
                  $GET PROYECTO;
                  writeln(PROYECTO.NOMBRE,        PROYECTO.NUMERO,
                  TRABAJA_EN.HORAS);
                  $FIND NEXT TRABAJA_EN WITHIN E_TRABAJAEN
                  end
            end;
```

```
EX5:  PROYECTO.NOMBRE:='ProductoX';
      $FIND ANY PRYECTO USING NOMBRE;
      if DB_STATUS = 0 then
        begin
        TRABAJA_EN.HORAS:=40.0;
        $FIND FIRST TRABAJA_EN WITHIN P_TRABAJAEN USING HORAS;
        while DB_STATUS = 0 do
            begin
            $GET TRABAJA_EN;
            $FIND OWNER WITHIN E_TRABAJAEN; $GET EMPLEADO;
            writeln(EMPLEADO.NOMBRE, EMPLEADO.APELLIDO);
            $FIND NEXT TRABAJA_EN WITHIN P_TRABAJAEN USING
            HORAS
      end;
```

En EX5, la calificación USING HORAS en FIND FIRST y FIND NEXT especifica que sólo se encuentren a los registros de TRABAJA_EN en la instancia actual de P_TRABAJAEN cuyas HORAS correspondan al valor de TRABAJA_EN.HORAS del UWA, el cual es '40.0'. Notemos que se usa la cláusula USING con FIND NEXT para encontrar *al siguiente miembro dentro de la misma ocurrencia*; cuando procesamos registros de determinado tipo *sin importar al conjunto al que pertenecen*, usamos FIND DUPLICATE en vez de FIND NEXT.

Usando las Facilidades del Lenguaje de Programación Huésped. Debido a que el DML de red es un lenguaje de registro a la vez, necesitamos usar las facilidades del lenguaje de programación huésped cada que un query requiere de un conjunto de registros. También lo necesitamos para calcular funciones sobre conjuntos de registros, tales como COUNT y AVERAGE, los cuales deben implementarse explícitamente por el programador.

Un ejemplo final ilustra cómo calcular funciones tales como COUNT y AVERAGE. Supongamos que deseamos calcular el número de empleados quienes supervisan a cada departamento y su salario promedio; esto se muestra en EX6. Asumimos que ha sido declarada en alguna parte una función PASCAL convierte_a_real, la cual convierte el valor de la cadena salario a un número real. También debemos de tener declaradas las variables total_sal:real y no_de_supervisores:integer para acumular el total de salarios y el número de supervisores en cada departamento. En EX6, notemos cómo probar si un empleado es supervisor determinando si el registro EMPLEADO participa como propietario en alguna instancia del conjunto ES_UN_SUPERVISOR:

```
EX6:  $FIND FIRST DEPTO WITHIN TODOS_DEP;
      while DB_STATUS = 0 do
            begin
            $GET DEPTO;
            write(DEPTO.NOMBRE);
```

```
        total_sal:=0; no_de_seupervisores:=0;
        $FIND FIRST EMPLEADO WITHIN TRABAJA_PARA;
        while DB_STATUS=0 do
            begin
            $GET EMPLEADO;
            $FIND FIRST SUPERVISOR WITHIN ES_UN_SUPERVISOR;
            if DB_STATUS = 0 then
                begin
                total_sal:=total_sal+convert_to_real(EMPLEADO.SALA);
                no_de_supervisores:=no_de_supervisores+1;
                end;
            $FIND NEXT EMPLEADO WITHIN TRABAJA_PARA;
            end;
    writeln('número de supervisores = ', no_de_supervisores);
    writeln('salario promedio = ', total_sal/no_de_supervisres);
    writeln();
    $FIND NEXT DEPTO WITHIN TODOS_DEP
    end;
```

7.5.4 Comandos DML para la Actualización de la Base de Datos

Aquí, discutiremos primero los comandos para actualizar registros -STORE, ERASE y MODIFY. Estos se usan para insertar, borrar y modificar a un registro respectivamente. Después de esto, ilustramos los comandos que modifican la instancias, los cuales son los comandos CONNECT, DISCONNECT y RECONNECT.

El Comando STORE. Se usa para insertar un nuevo registro. Antes de hablar de STORE, debemos configurar la variable UWA del tipo de registro correspondiente de tal forma que sus valores de campo contengan los valores del nuevo registro. Por ejemplo, para insertar un nuevo registro EMPLEADO para John F. Smith, podemos usar EX7:

```
EX7:  EMPLEADO.NOMBRE:='John';
      EMPLEADO.APELLIDO:='Smith';
      EMPLEADO.INIC:='F';
      EMPLEADO.IMSS:='567342739';
      EMPLEADO.DIRECCION:='40 Walcott Road, Minneapolis, Minnesota 55433';
      EMPLEADO.FECHANAC:='10-ENE-55';
      EMPLEADO.SEXO:='M';
      EMPLEADO.SALARIO:='25000.00';
      EMPLEADO.NOMDEPTO:='';
      $STORE EMPLEADO
```

El resultado del comando STORE es la inserción del contenido actual del registro UWA del tipo especificado en la base de datos. El registro recién insertado se vuelve el CRU y el

registro actual para su tipo, así como el conjunto actual para cualquier tipo que tenga al registro como propietario o miembro.

Efectos de la Opción SET SELECTION sobre el Comando STORE. Las opciones AUTOMATIC SET SELECTION tienen diferentes efectos sobre la ejecución del comando STORE. Recordemos que, en un conjunto con opción de inserción AUTOMATIC, el nuevo miembro debe conectarse a una instancia al mismo tiempo que se inserta a la base de datos por el comando STORE. A continuación discutiremos brevemente tres de las opciones SET SELECTION -BY STRUCTURAL, BY APPLICATION y BY VALUE.

Primero ilustraremos la opción **BY STRUCTURAL**. Esta opción tiene el siguiente formato:

SET SELECTION IS STRUCTURAL
<elemento de datos> **IN** <registro miembro> = <elemento de datos> **IN** <registro propietario>

Esto permite que el DBMS *determine por sí mismo* la ocurrencia en la cual va a conectarse el registro recién ingresado; esto se ilustra por las declaraciones de los conjuntos P_TRABAJAEN y E_TRABAJAEN de la Figura 7.10(b). Por ejemplo, para relacionar el registro EMPLEADO con IMSS = '567342739', recién insertado en EX7, como un trabajador de 40 horas a la semana en el proyecto cuyo número es el 55, debemos crear y almacenar un nuevo registro de enlace TRABAJA_EN con los valores apropiados de EIMSS y PNUMERO, como se muestra en EX8. El comando STORE TRABAJA_EN de EX8 conecta automáticamente al registro recientemente ingresado TRABAJA_EN en la instancia E_TRABAJAEN propiedad del registro EMPLEADO con IMSS = '567342739' y a la instancia P_TRABAJAEN propiedad de PROYECTO con NUMERO = 55 localizando automáticamente a estos propietarios y a sus instancias. El registro recientemente ingresado también se vuelve el conjunto actual de ambos conjuntos. Si cualquiera de los propietarios no existe en la base de datos, el comando STORE generaría un error y el nuevo registro TRABAJA_EN no se insertaría en la base de datos.

```
EX8:  TRABAJA_EN.EIMSS:='567342739';
      TRABAJA_EN.PNUMERO:=55;
      TRABAJA_EN.HORAS:='40.0';
      $STORE TRABAJA_EN;
```

En el DDL de red, la opción **BY APPLICATION** tiene el siguiente formato:

SET SELECTION IS BY APPLICATION

La aplicación es responsable de seleccionar la ocurrencia adecuada *antes de* almacenar al nuevo miembro. Por ejemplo, para insertar un nuevo PROYECTO que es controlado por el departamento de Investigación, debemos explícitamente hacer del DEPTO Investigación

el conjunto de actualidad para CONTROLA *antes de* colocar el comando STORE PROJECT.

En el DDL de red, la opción **BY VALUE** tiene el siguiente formato:

SET SELECTION IS BY VALUE OF <elemento de dato> **IN** <registro propietario>

Se ilustra por la declaración del conjunto ES_UN_SUPERVISOR en la Figura 7.10(b). En este caso sólo colocamos el valor del campo especificado en la declaración SET SELECTION IS BY VALUE -IMSS de empleado para el conjunto ES_UN_SUPERVISOR- antes de usar el comando STORE. Este debe ser un campo llave del propietario, y el DBMS usa ese valor para encontrar al propietario (único) del registro. Por ejemplo, para insertar un nuevo SUPERVISOR al empleado correspondiente con IMSS='567342739', usamos EX9. Este proporciona el valor de EMPLEADO.IMSS dentro del programa, el cual usa el DBMS para seleccionar al propietario apropiado EMPLEADO y automáticamente conectar el nuevo SUPERVISOR a su instancia.

```
EX9:  SUPERVISOR.SUPERVISOR_IMSS:='567342739';
      EMPLEADO.IMSS:= '567342739';
      $STORE SUPERVISOR;
```

Los Comandos ERASE y ERASE ALL. Para borrar un registro de la base de datos, primero debemos hacer que el registro sea el CRU y luego usar el comando ERASE. Por ejemplo, para borrar el EMPLEADO insertado en EX7, podríamos usar EX10:

```
EX10: EMPLEADO.IMSS:='567342793';
      $FIND ANY EMPLEADO USING IMSS;
      if DB_STATUS = 0 then $ERASE EMPLEADO;
```

El efecto del comando ERASE se determina por la opción de retensión sobre cualquier miembro que sea *propiedad del registro que está siendo borrado*. Por ejemplo, el efecto del comando ERASE en EX10 depende de la retensión para cada conjunto que tenga a EMPLEADO como propietario. Si la retensión es OPTIONAL, los miembros se guardan en la base de datos pero se desconectan del propietario antes de borrarse. Si la retensión es FIXED, se borran todos los miembros junto con su propietario. Finalmente, si la retensión es MANDATORY y algunos miembros son propiedad del registro a borrar, se rechaza el comando y se genera un mensaje de error. No podemos borrar al propietario, porque los miembros no tendrían propietario, lo cual no se permite en conjuntos MANDATORIOS. Estas reglas se aplican recursivamente a cualquier registro adicional propiedad de otros registros cuyo borrado se efectúa automáticamente por un comando ERASE. Si no se usa con precaución, el borrado puede propagarse a través de la base de datos y ocasionar mucho daño.

Una variante del comando ERASE, **ERASE ALL**, permite al programador eliminar un registro y a todos los registros de su propiedad directa o indirectamente. Esto significa

que *todos* los miembros propiedad del registro se borran. Además, los miembros propiedad de cualquiera de los registros borrados también se eliminan, en cualquier número de repeticiones. Por ejemplo, EX11 borra el registro DEPTO Investigación, así como a todos los EMPLEADOS que son propiedad de ese DEPTO vía TRABAJA_PARA y cualquier PROYECTO que sea propiedad de ese DEPTO vía CONTROLA. Además, cualquier DEPENDIENTE, SUPERVISOR, DEPARTAMENTO, o TRABAJA_EN propiedad de los registros EMPLEADO o PROYECTO se borran *automáticamente*:

```
EX11: DEPTO.NOMBRE:='Investigación';
      $FIND ANY DEPTO USING NOMBRE;
      if DB_STATUS = 0 then $ERASE ALL DEPTO;
```

También podemos usar un ciclo para borrar un número de registros. Por ejemplo, supongamos que queremos borrar a todos los empleados que trabajan en el departamento de Investigación, pero no al registro DEPTO; para hacer esto podemos usar EX12. Notemos que el CRU y el tipo actual para el registro recién borrado apunta a una *posición "vacía"* donde solía estar el registro que fue borrado. Esto es por lo que la instrucción FIND NEXT en EX12 trabaja adecuadamente:

```
EX12: DEPTO.NOMBRE:='Investigación';
      $FIND ANY DEPTO USING NOMBRE;
      if DB_STATUS = 0 then
            begin
            $FIND FIRST EMPLEADO WITHIN TRABAJA_PARA;
            while DB_STATUS = 0 DO
                  begin
                        $ERASE EMPLEADO;
                        $FIND NEXT EMPLEADO WITHIN TRABAJA_PARA
                  end
            end;
```

Comando MODIFY. El último comando para actualizar registros es MODIFY, el cual cambia algunos valores de campo del registro. Para modificar valores de campo de un registro debemos seguir la siguiente secuencia:

- Hacer que el registro a modificar sea el CRU.
- Recuperar el registro a la variable UWA correspondiente.
- Modificar los campos deseados en la variable UWA.
- Emitir el comando MODIFY.

Por ejemplo, para dar a todos los empleados del DEPTO de Investigación un 10% de aumento, podemos usar EX13.

```
EX13: DEPTO.NOMBRE:='Investigación';
      $FIND ANY DEPTO USING NOMBRE;
      if DB_STATUS = 0 then
            begin
            $FIND FIRST EMPLEADO WITHIN TRABAJA_PARA;
            while DB_STATUS = 0 do
                  begin
                  $GET EMPLEADO;
                  EMPLEADO.SALARIO:=
                  convert_to_string(convert_to_real(EMPLEADO.SALARIO) * 1.1;
                  $MODIFY EMPLEADO;
                  $FIND NEXT EMPLEADO WITHIN TRABAJA_PARA
            end
      end;
```

Comandos para Actualizar Instancias. Ahora consideremos las tres operaciones de actualización -CONNECT, DISCONNECT y RECONNECT- las cuales se usan para insertar y eliminar miembros en instancias. El comando CONNECT inserta a un registro miembro a la instancia. El miembro debe ser la unidad actual de corrida y se conecta a la instancia que es el tipo de conjunto actual. Por ejemplo, para conectar el registro EMPLEADO con IMSS = '567342793' al conjunto TRABAJA_PARA propiedad de DEPTO Investigación, podemos usar EX14:

```
EX14: DEPTO.NOMBRE:='Investigación';
      $FIND ANY DEPTO USING NOMBRE;
      if DB_STATUS = 0 then
            begin
            EMPLEADO.IMSS:='567342793';
            $FIND ANY EMPLEADO USING IMSS;
            if DB_STATUS = 0 then
                  $CONNECT EMPLEADO TO TRABAJA_PARA;
            end;
```

El comando CONNECT puede usarse sólo con conjuntos MANUAL o AUTOMATIC OPTIONAL. Con otros conjuntos AUTOMATIC, el sistema conecta automáticamente un miembro a una instancia, gobernada por la opción especificada en SET SELECTION, tan pronto como se almacena el registro.

El comando DISCONNECT se usa para quitar a un miembro de una instancia sin conectarlo a otra instancia. Por lo tanto, sólo puede usarse en conjuntos OPTIONAL. Primero hacemos que el registro se desconecte del CRU antes de usar el comando

DISCONNECT. Por ejemplo, para quitar el EMPLEADO con IMSS='836483873' de la instancia SUPERVISA de la cual es miembro, usamos EX15:

```
EX15: EMPLEADO.IMSS:='836483873';
      $FIND ANY EMPLEADO USING IMSS;
      if DB_STATUS = 0 then
      $DISCONNECT EMPLEADO FROM SUPERVISA;
```

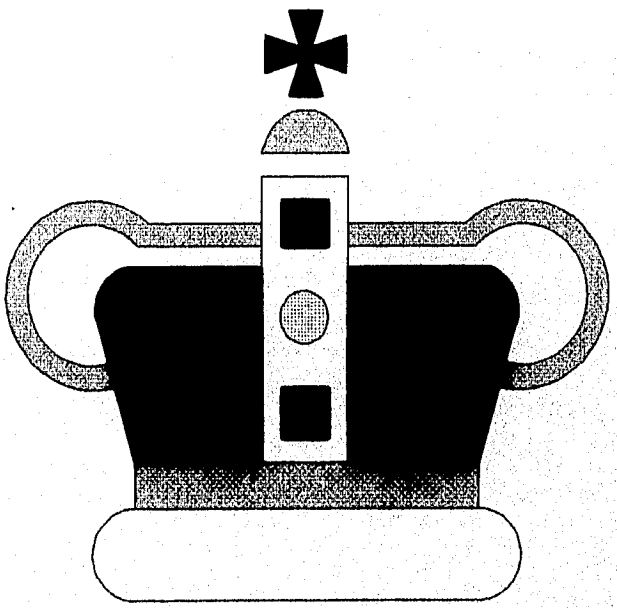
Finalmente, el comando RECONNECT puede usarse con conjuntos OPTIONAL y MANDATORY, pero no con FIXED. El comando RECONNECT mueve un registro miembro de una instancia a otra del *mismo tipo de conjunto*. No puede usarse en conjuntos FIXED porque un miembro no puede moverse de una instancia a otra bajo el argumento FIXED. Antes de usar el comando RECONNECT, la instancia a la cual se va a conectar el miembro debe ser del tipo actual, y el miembro a conectar debe ser el CRU. Para hacerlo, necesitamos usar una frase adicional con el comando FIND -llamada RETAINING CURRENCY- la cual discutiremos.

La Frase RETAINING CURRENCY. El comando RECONNECT y la frase RETAINING CURRENCY se ilustran en el contexto de EX16, el cual elimina al gerente actual del departamento de Investigación y asigna al empleado con IMSS='836483873' como su nuevo gerente. Notemos que el conjunto MANEJA se declara AUTOMATIC MANDATORY, así otro registro EMPLEADO posee actualmente al registro DEPTO Investigación en el conjunto MANEJA. Sin embargo, el registro DEPTO Investigación ya es miembro de una instancia de MANEJA diferente, así cuando se convierte en el CRU también se volverá el registro actual para el conjunto MANEJA:

```
EX16: EMPLEADO.IMSS:='836483873';
      $FIND ANY EMPLEADO USING IMSS;
      if DB_STATUS = 0
            begin
            DEPTO.NOMBRE='Investigación';
            $FIND ANY DEPTO USING NOMBRE RETAINING MANEJA
            CURRENCY;
            if DB_STATUS = 0 then $RECONNECT DEPTO WITHIN MANEJA
            end;
```

Para lograr que el registro se vuelva el CRU *sin cambiar al registro actual*, usamos la frase RETAINING CURRENCY junto al comando FIND. En EX16, usamos la frase RETAINING MANAGES CURRENCY junto al comando FIND. Esto cambia el CRU al DEPARTAMENTO Investigación pero deja al conjunto actual de MANEJA sin cambio; queda la instancia propiedad de EMPLEADO cuyo IMSS es 836483873.

CAPITULO 8



Modelo de Datos Jerárquico y el Sistema - IMS

8.1.1 Relaciones Padre-Hijo y Esquemas Jerárquicos

El modelo jerárquico emplea dos conceptos principales de estructuras de datos: relaciones de registros padres e hijos. Un **registro** es una colección de **valores de campos** que proporcionan información sobre una entidad o instancia de la relación. Los registros del mismo tipo se agrupan en **tipos de registros**. A un tipo de registro se le asigna un nombre, y su estructura se define por una colección de **campos** o **elementos de datos** nombrados. Cada campo tiene un cierto tipo de dato, tal como entero, real o cadena.

Un **tipo de relación padre hijo** (tipo **PCR**) es una relación 1:N entre dos tipos de registros. Al tipo de registro del lado 1 se le llama registro **padre**, y al del lado N se le llama **registro hijo**. Una **ocurrencia** (o **instancia**) **del tipo PCR** consta de *un registro* del tipo padre y *un número de registros* (cero o más) del tipo hijo.

Un **esquema de base de datos jerárquico** (tipo **PCR**) consiste de un número de esquemas jerárquicos. Cada **esquema jerárquico** (o **jerárquia**) consta de un número de registros y PCR.

Un esquema jerárquico se despliega como un **diagrama jerárquico**, en el cual los nombres de los registros se despliegan en cajas rectangulares y los PCR se despliegan como líneas que conectan al padre con el hijo. La Figura 8.1 muestra un diagrama jerárquico para un esquema con tres registros y dos PCR. Los tipos de registro son DEPARTAMENTO, EMPLEADO y PROYECTO. Los nombres de campo pueden desplegarse bajo cada nombre del registro, como se muestra en la Figura 8.1. En algunos diagramas, por brevedad, desplegaremos sólo los nombres de registros.

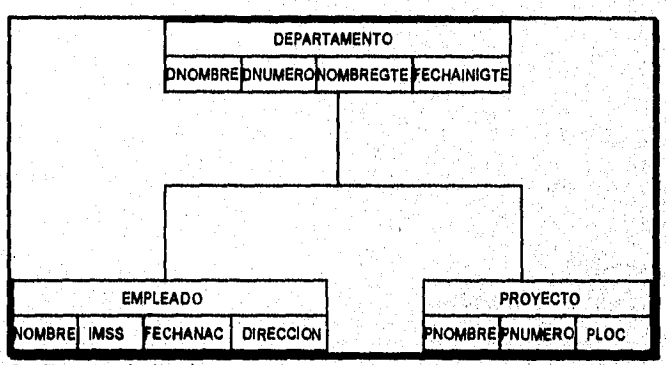


Figura 8.1 Esquema jerárquico.

En un esquema jerárquico nos referimos a un PCR listando al par (padre, hijo) entre paréntesis. Los dos tipos PCR en la Figura 8.1 son (DEPARTAMENTO, EMPLEADO) y (DEPARTAMENTO, PROYECTO). Notemos que los PCR *no* tienen nombre dentro del modelo jerárquico. En la Figura 8.1 cada *ocurrencia* del PCR (DEPARTAMENTO, EMPLEADO) relaciona a un departamento con los registros de los *muchos* (cero o más)

empleados que trabajan en ese departamento. Una *ocurrencia* PCR del tipo (DEPARTAMENTO, PROYECTO) relaciona a un departamento con los proyectos controlados por ese departamento. La Figura 8.2 muestra dos ocurrencias (o instancias) PCR para cada uno de estos dos tipos de PCR.

8.1.2 Propiedades de un Esquema Jerárquico

Un esquema jerárquico de registros PCR debe tener las siguientes propiedades:

1. Un registro del esquema jerárquico, llamado **raíz**, no participa como hijo en ningún PCR.

2. Todo registro excepto el raíz participa como hijo en sólo *un* PCR.

3. Un registro puede participar como padre de cualquier número (cero o más) de PCR.

4. Al registro que no participa como padre en ningún PCR se le llama **hoja** del esquema jerárquico.

5. Si un registro participa como padre de más de un PCR, entonces sus registros *hijo están ordenados.* En un diagrama jerárquico el orden se despliega, por convensión, de izquierda a derecha.

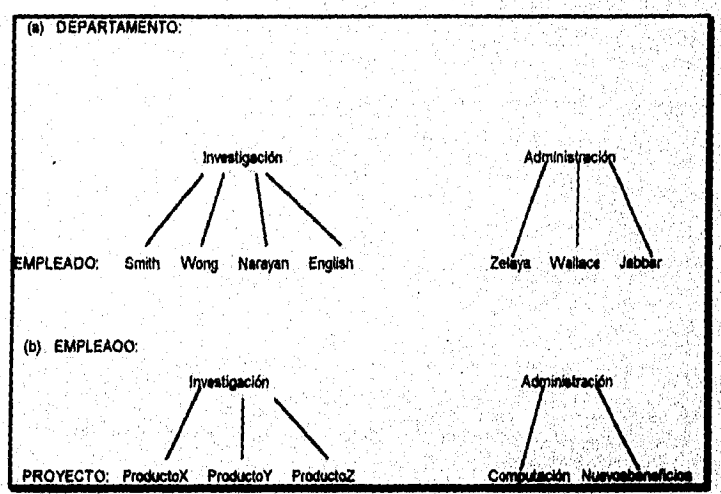


Figura 8.2 Ocurrencias de PCR. (a) Dos ocurrencias de PCR (DEPARTAMENTO, EMPLEADO). (b) Dos ocurrencias de PCR (DEPARTAMENTO, PROYECTO)

La definición de un esquema jerárquico define una **estructura de datos de árbol**. En terminología de estructuras de datos de árbol, un registro corresponde a un **nodo** del

árbol, y un PCR corresponde a una **punta** (o **arco**) del árbol. En los diagramas jerárquicos la convensión es que todas las puntas que emanan del mismo nodo padre se unen (Figura 8.1).

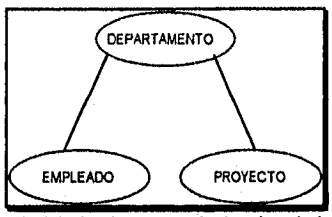


Figura 8.3 Representación de árbol del esquema jerárquico de la Figura 8.1.

Las propiedades anteriores de un esquema jerárquico significan que cada nodo excepto la raíz tienen un sólo nodo padre. Sin embargo, un nodo puede tener varios hijos, en este caso se ordenan de izquierda a derecha. En la Figura 8.1 EMPLEADO es el primer hijo de DEPARTAMENTO, y PROYECTO es el segundo. Las propiedades previamente identificadas también limitan los tipos de relaciones que pueden representarse en un esquema jerárquico. Las relaciones M:N entre registros *no pueden* representarse directamente, debido a que las relaciones padre-hijo son 1:N, y un registro *no puede participar como hijo* en dos o más relaciones padre-hijo distintas.

Dentro del modelo jerárquico puede manejarse una relación M:N permitiendo la *duplicación de instancias hijo*. Por ejemplo, consideremos una relación M:N entre EMPLEADO y PROYECTO, donde un proyecto puede tener distintos empleados trabajando en él, y un empleado puede trabajar en varios proyectos. Podemos representar la relación como un PCR (PROYECTO, EMPLEADO) como el mostrado en la Figura 8.4(a). En este caso un registro que describe al mismo empleado puede duplicarse apareciendo una vez debajo de *cada* proyecto en el que trabaja el empleado. Alternativamente, podemos representar la relación como un PCR (EMPLEADO, PROYECTO) mostrado en la Figura 8.4(b), en cuyo caso pueden duplicarse los registros proyecto.

EJEMPLO 1: Consideremos las siguientes instancias de la relación EMPLEADO-PROYECTO:

Proyecto	Empleados que Trabajan en el Proyecto
A	E1, E3, E5
B	E2, E4, E6
C	E1, E4
D	E2, E3, E4, E5

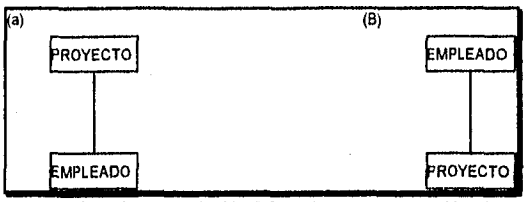


Figura 8.4 Representación de una relación M:N. (a) Representación de la relación M:N. (b) Representación alterna de la relación M:N.

Si estas instancias se almacenan usando el esquema jerárquico de la Figura 8.4(a), habrá cuatro ocurrencias del PCR (PROYECTO, EMPLEADO) -uno para cada proyecto. Los registros de empleado E1, E2, E3 y E5 aparecerán *dos veces cada uno* como hijo, debido a que cada uno de ellos trabaja en dos proyectos. El registro del empleado E4 aparecerá tres veces -una debajo de B, C y D. Algunos valores de campo en los registros empleados pueden ser dependientes de contexto; esto es, estos valores de campo dependen de EMPLEADO y PROYECTO. Tales datos pueden diferir en cada ocurrencia de empleado duplicado debido a que también depende del padre del proyecto. Un ejemplo es un campo que da el número de horas a la semana que un empleado trabaja en el proyecto. Sin embargo, la mayoría de los valores de campo en los registros empleado, tales como el nombre, número de seguro social, salario, se duplicarían bajo cada proyecto en el que trabajara el empleado.

Para evitar tal duplicación, se usa una técnica en la que varios esquemas jerárquicos pueden especificarse sobre el mismo esquema de la base de datos.

8.1.3 Arboles de Ocurrencia Jerárquica

Para un esquema jerárquico dado, existen muchas ocurrencias jerárquicas en la base de datos. Cada **ocurrencia jerárquica**, también llamada **árbol de ocurrencia**, es una estructura de **árbol** cuya raíz es un registro sencillo del tipo raíz. El árbol de ocurrencia contiene también a todos los hijos del tipo raíz, todos los hijos dentro del PCR de cada hijo de la raíz, y así sucesivamente, para todos los registros del tipo hoja.

Por ejemplo, consideremos el diagrama jerárquico mostrado en la Figura 8.5, el cual representa parte de la base de datos COMPAÑIA. La Figura 8.6 muestra un árbol de ocurrencia jerárquica de este esquema. En el árbol de ocurrencia, cada nodo es un registro, y cada arco representa una relación padre-hijo entre dos registros. En ambas Figuras 8.5 y 8.6 usamos los caracteres **D, E, P, T, S y W** para representar **indicadores de tipo** para los registros DEPARTAMENTO, EMPLEADO, PROYECTO, DEPENDIENTE, SUPERVISA y TRABAJADOR, respectivamente.

En una estructura de árbol, a la raíz se dice tener **nivel** cero. El nivel de un nodo no raíz es uno más que el de su nodo padre, como se muestra en las Figuras 8.5 y 8.6. Un **descendiente** D de un nodo N es un nodo conectado a N vía uno o más arcos tales que el

nivel de D es mayor que el nivel de N. Un nodo N y todos sus nodos descendientes forman un **sub-árbol** de nodo N. Un **árbol de ocurrencia** puede definirse como el sub-árbol de un registro cuyo tipo es de raiz.

La raíz de un árbol de ocurrencia es un sólo registro del tipo raiz. Puede haber distintos números de ocurrencias de cada registro no raíz, cada uno de los cuales debe tener un padre en el árbol de ocurrencia; esto es, cada ocurrencia debe participar en un PCR. Notemos que cada nodo no raíz, junto con todos sus descendientes, forman un **sub-árbol**, el cual, tomado solo, satisface la estructura de un árbol de ocurrencia para una porción del diagrama jerárquico. Notemos también, que el nivel de un registro en un árbol de ocurrencia es el nismo que el nivel de su registro en el diagrama jerárquico.

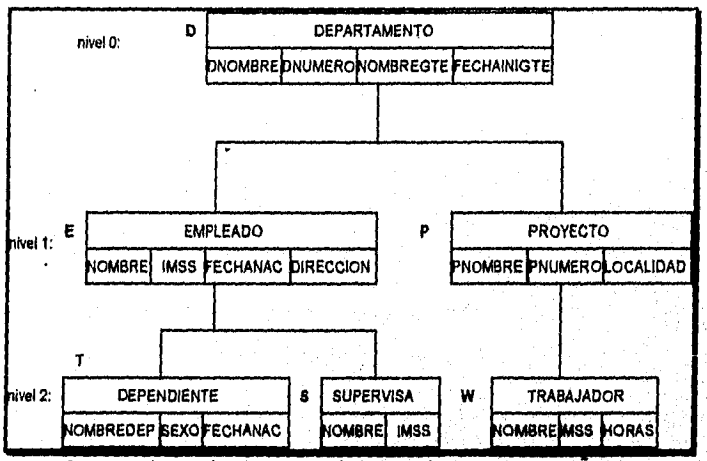


Figura 8.5 Esquema jerárquico para parte de la base de datos COMPAÑIA.

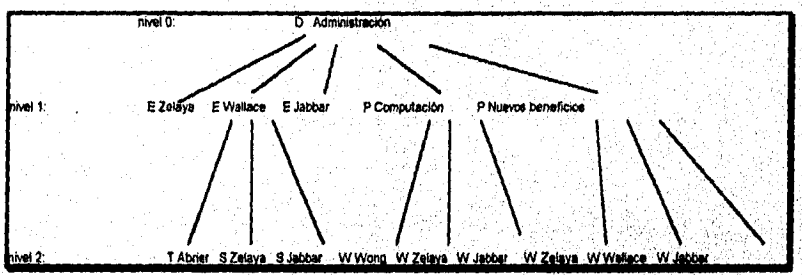


Figura 8.6 Ocurrencia jerárquica (o árbol de ocurrencia) del esquema jerárquico en la Figura 8.5

8.1.4 Forma Linealizada de una Ocurrencia Jerárquica

Una estructura de almacenamiento particularmente sencilla es el **registro jerárquico**, el cual es un ordenamiento lineal de registros en un árbol de ocurrencia en el *preorden transversal* del árbol. Este orden produce una secuencia de ocurrencias registro conocidos como la **secuencia jerárquica** (o **secuencia jerárquica de registros**) del árbol de ocurrencia; puede obtenerse aplicando el siguiente procedimiento recursivo a la raíz de un árbol de ocurrencia:

```
procedure Pre_orden_transverso(registro raíz);
        begin
        output (registro raíz);
        for cada hijo del registro raíz en el orden de izquierda a derecha do
                Pre_orden_transverso(registro hijo);
        end;
```

El procedimiento anterior, cuando se aplica al orden de ocurrencia de la Figura 8.6 da como resultado la secuencia jerárquica mostrada en la Figura 8.7. Si usamos la secuencia jerárquica para implementar árboles de ocurrencia, necesitamos almacenar un indicador de registro con cada árbol debido a los diferentes registros y número de variables de registros hijo en cada relación padre-hijo. Conforme pasa secuencialmente a través de ellos el sistema necesita examinar el tipo de cada registro. Estos indicadores de tipo son estructuras de implementación y no son vistos por el usuario del DBMS jerárquico.

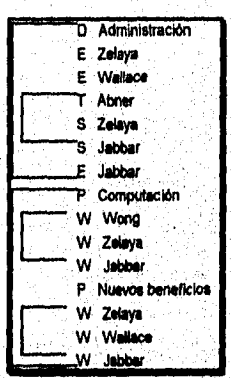


Figura 8.7 Secuencia jerárquica para el árbol de ocurrencia de la Figura 8.6.

La secuencia jerárquica es deseable debido a que los hijos siguen a su padre en el almacenamiento. Sin embargo, los hijos se colocan de manera colectiva después del padre sólo si los registros son hojas en el árbol de ocurrencia. De otro modo, los sub-árboles de cada hijo se colocan después de su padre en el orden de izquierda a derecha.

La secuencia jerárquica también es importante debido a que algunos lenguajes jerárquicos de manipulación de datos, la usan como referencia para definir operaciones de bases de datos jerárquicas.

Una **trayectoria jerárquica** es una secuencia de nodos N_1, N_2...N_i, donde N_1 es la raiz de un árbol y N_j es el hijo de N_{j-1} para j = 1, 2, 3... i. Una trayectoria jerárquica está **completa** si N_i es una hoja del árbol. Una **escoba** es un conjunto de trayectorias jerárquicas que resultan de la trayectoria jerárquica N_1, N_2, ... N_i junto con todas las trayectorias jerárquicas en el sub-árbol de N_i. Por ejemplo, (DEPARTAMENTO, EMPLEADO, SUPERVISA) es una trayectoria completa en el esquema jerárquico de la Figura 8.5. En el árbol de ocurrencia de la Figura 8.6, (Administración, Wallace) es una trayectoria, y (Administración, Wallace, {Abner, Zelaya, Jabbar} es una escoba.

Ahora podemos definir una **ocurrencia de base de datos jerárquica** como una secuencia de todos los árboles que son ocurrencias de un esquema jerárquico. Por ejemplo, una ocurrencia de base de datos jerárquica del esquema de la Figura 8.5 consistiría en un número de árboles similares al mostrado en la Figura 8.6. Habría un árbol para cada DEPARTAMENTO, y se ordenarían como primero, segundo, ... último árbol.

8.2 Relaciones Virtuales Padre-Hijo

El modelo jerárquico tiene problemas para modelar cierto tipo de relaciones. Estas incluyen las siguientes relaciones y situaciones:

1. Relaciones M:N

2. El caso donde un registro participa como hijo en más de un PCR.

3. Relaciones N-arias con más de dos tipos de registros participantes.

Como vimos anteriormente, el caso 1 puede representarse como un PCR al costo de duplicar los registros hijo. El caso 2 puede representarse de manera similar, con más duplicación de registros. El caso 3 presenta un problema porque el PCR es una relación binaria.

El duplicado de registros, además de desperdiciar espacio de almacenamiento, ocasiona problemas con el mantenimiento de la consistencia debido al duplicado del mismo registro.

Un **registro virtual** (o **apuntador**) VC es un registro con la propiedad de que cada uno de sus registros contiene un apuntador a un registro de otro tipo VP. VC juega el rol de "hijo virtual" y VP de "padre virtual" en una "relación virtual padre-hijo". Cada ocurrencia c de VC apunta exactamente a una ocurrencia p de VP. En vez de duplicar al registro p que es un árbol de ocurrencia, incluimos al registro virtual c que contiene un apuntador a p. Varios registros virtuales pueden apuntar a p, pero en la base de datos se almacena una sola copia de p.

La Figura 8.8 muestra la relación M:N entre EMPLEADO y PROYECTO representada con registros virtuales EPOINTER y PPOINTER. Comparemos esto con la Figura 8.6, donde la misma relación se representó sin registros virtuales. La Figura 8.6 muestra los árboles y apuntadores de las instancias datos en Ejemplo 1 cuando se usa el esquema jerárquico mostrado en la Figura 8.5(a). En la Figura 8.6 sólo hay una copia de cada EMPLEADO, sin embargo, varios registros virtuales pueden apuntar al mismo EMPLEADO. Por lo tanto, la información almacenada en EMPLEADO no se duplica. En el apuntador virtual se incluye la información que depende de padre e hijo -tal como las horas a la semana que trabaja un empleado en un proyecto, *tal dato es popularmente* conocido entre los usuarios de bases de datos jerárquicas como **dato intersección**

Notemos que la relación entre EMPLEADO y EPOINTER en la Figura 8.8(a) es una relación 1:N y por lo tanto califica como PCR. A tal relación se le llama **relación virtual padre-hijo**. Al EMPLEADO se le llama **padre virtual** de EPOINTER, y por consiguiente, a EPOINTER se le llama **hijo virtual** de EMPLEADO. Un PCR se implementa usualmente usando la secuencia jerárquica, mientras que un VPCR se implementa estableciendo un apuntador (físico conteniendo una dirección, o lógico conectando una llave) de un hijo virtual a su padre virtual.

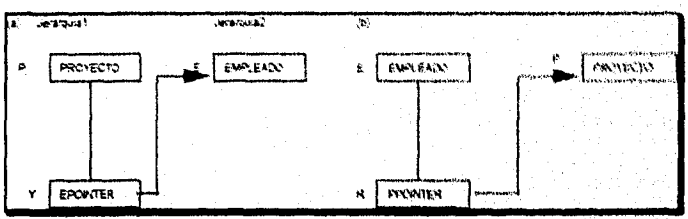


Figura 8.8 Representación de una relación M:N usando VPCR. (a) Representación de la relación M:N, con padre virtual EMPLEADO. (b) Representación alterna de la relación M:N, con padre virtual PROYECTO.

La Figura 8.10 muestra un esquema de base de datos jerárquica de la base COMPAÑIA que usa algunos VPCR y no tiene redundancia en sus registros. El esquema de la base de datos jerárquica se conforma de dos esquemas -uno con raíz DEPARTAMENTO, y el otro con raíz EMPLEADO. Para representar las relaciones sin redundancia se incluyen cuatro VPCR, todos con padre virtual EMPLEADO.

Algo a considerar acerca de los VPCR es que pueden implementarse de diferentes maneras. Como se discutió anteriormente, una opción es tener un apuntador del hijo virtual al padre virtual. Una segunda opción, es tener además del apuntador hijo-padre, un enlace de regreso del padre virtual a una lista enlazada de hijos virtuales. Al apuntador del padre virtual al primer hijo virtual se le llama **apuntador a hijo virtual**, mientras que a un apuntador de un hijo virtual al siguiente se le llama **apuntador virtual gemelo**. En este caso, el modelo jerárquico se vuelve *muy similar* al de red. Este enlace de regreso facilita la recuperación de todos los hijos virtuales de un padre virtual en particular.

La Figura 8.8 muestra la relación M:N entre EMPLEADO y PROYECTO representada con registros virtuales EPOINTER y PPOINTER. Comparemos esto con la Figura 8.4, donde la misma relación se representó sin registros virtuales. La Figura 8.9 muestra los árboles y apuntadores de las instancias dadas en Ejemplo 1 cuando se usó el esquema jerárquico mostrado en la Figura 8.8(a). En la Figura 8.9 sólo hay una copia de cada EMPLEADO; sin embargo, varios registros virtuales pueden apuntar al mismo EMPLEADO. Por lo tanto, la información almacenada en EMPLEADO no se duplica. En el apuntador virtual se incluye la información que depende de padre e hijo -tal como las horas a la semana que trabaja un empleado en un proyecto; tal dato es popularmente conocido entre los usuarios de bases de datos jerárquicas como **dato intersección**.

Notemos que la relación entre EMPLEADO y EPOINTER en la Figura 8.8(a) es una relación 1:N y por lo tanto califica como PCR. A tal relación se le llama **relación virtual padre-hijo**. Al EMPLEADO se le llama **padre virtual** de EPOINTER; y por consiguiente, a EPOINTER se le llama **hijo virtual** de EMPLEADO. Un PCR se implementa usualmente usando la secuencia jerárquica, mientras que un VPCR se implementa estableciendo un apuntador (físico conteniendo una dirección, o lógico conectando una llave) de un hijo virtual a su padre virtual.

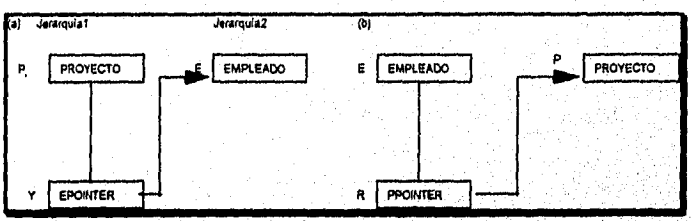


Figura 8.8 Representación de una relación M:N usando VPCR. (a) Representación de la relación M:N, con padre virtual EMPLEADO. (b) Representación alterna de la relación M:N, con padre virtual PROYECTO.

La Figura 8.10 muestra un esquema de base de datos jerárquica de la base COMPAÑIA que usa algunos VPCR y no tiene redundancia en sus registros. El esquema de la base de datos jerárquica se conforma de dos esquemas -uno con raíz DEPARTAMENTO, y el otro con raíz EMPLEADO. Para representar las relaciones sin redundancia se incluyen cuatro VPCR, todos con padre virtual EMPLEADO.

Algo a considerar acerca de los VPCR es que pueden implementarse de diferentes maneras. Como se discutió anteriormente, una opción es tener un apuntador del hijo virtual al padre virtual. Una segunda opción, es tener además del apuntador hijo-padre, un enlace de regreso del padre virtual a una lista enlazada de hijos virtuales. Al apuntador del padre virtual al primer hijo virtual se le llama **apuntador a hijo virtual**, mientras que a un apuntador de un hijo virtual al siguiente se le llama **apuntador virtual gemelo**. En este caso, el modelo jerárquico se vuelve *muy similar* al de red. Este enlace de regreso facilita la recuperación de todos los hijos virtuales de un padre virtual en particular.

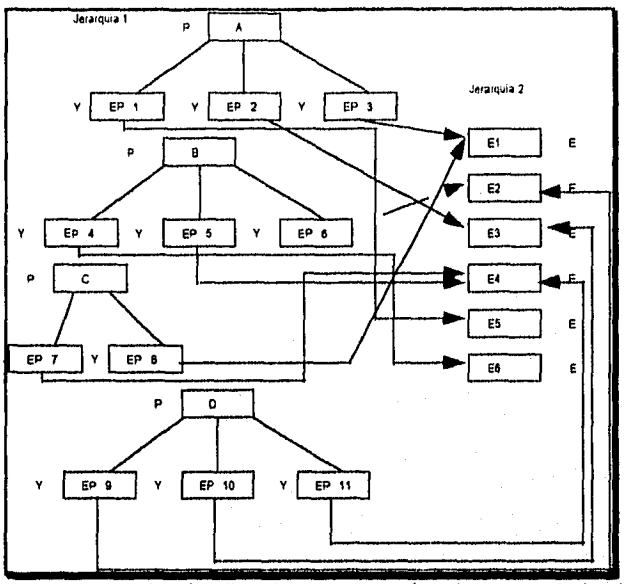


Figura 8.9 Las ocurrencias del Ejemplo 1 corresponden al esquema jerárquico en la Figura 8.8(a).

8.3 Argumentos de Integridad en el Modelo Jerárquico

Cada vez que se especifica un esquema jerárquico existe un número de **argumentos inherentes**. Estos incluyen los siguientes:

1. Ninguna ocurrencia excepto los registros raíz pueden existir sin relacionarse a un padre. Esto tiene las siguientes implicaciones.

   a. No puede insertarse un hijo a menos que se enlace a un padre.

   b. Puede borrase un hijo independientemente de su padre; sin embargo, el borrado de un padre resulta automáticamente en el borrado de todos sus hijos y descendientes.

   c. Las reglas anteriores no se aplican a registros virtuales padre e hijo. La regla aquí es que un apuntador en un registro virtual hijo debe apuntar a una ocurrencia del registro virtual padre. No se debe permitir el borrado de un registro mientras que existan apuntadores de registros virtuales hijo a él, haciéndolo un padre virtual.

2. Si un registro hijo tiene a dos o más padres del *mismo* tipo de registro, el hijo debe duplicase una vez debajo de cada padre.

3. Un hijo que tiene dos o más padres de *diferentes* tipos de registro puede hacerlo sólo teniendo al menos un padre real, con los demás representados como padres virtuales.

Cualesquiera otros argumentos que no son implícitos en un esquema jerárquico deben reforzarse explícitamente por los programadores en los programas de actualización a la base de datos. Por ejemplo, si se actualiza un registro duplicado, es responsabilidad del programa de actualización asegurarse que todas las copias se actualicen de la misma forma.

8.4 Definición de Datos en el Modelo Jerárquico

En esta sección daremos una definición de un lenguaje de definición de datos jerárquico (HDDL), el cual no es el lenguaje de ningún DBMS jerárquico en específico pero se usó para ilustrar los conceptos de lenguaje de base de datos jerárquica. El HDDL demuestra cómo puede definirse un esquema de base de datos jerárquico. Para definir un esquema de base de datos jerárquico, debemos definir los campos de cada tipo de registro, el tipo de cada campo, y cualquier argumento de llave sobre los campos. Además, debemos de especificar a un registro raíz como tal; y para cada registro no raíz, debemos especificar su padre (real) en un PCR. Así como cualquier VPCR.

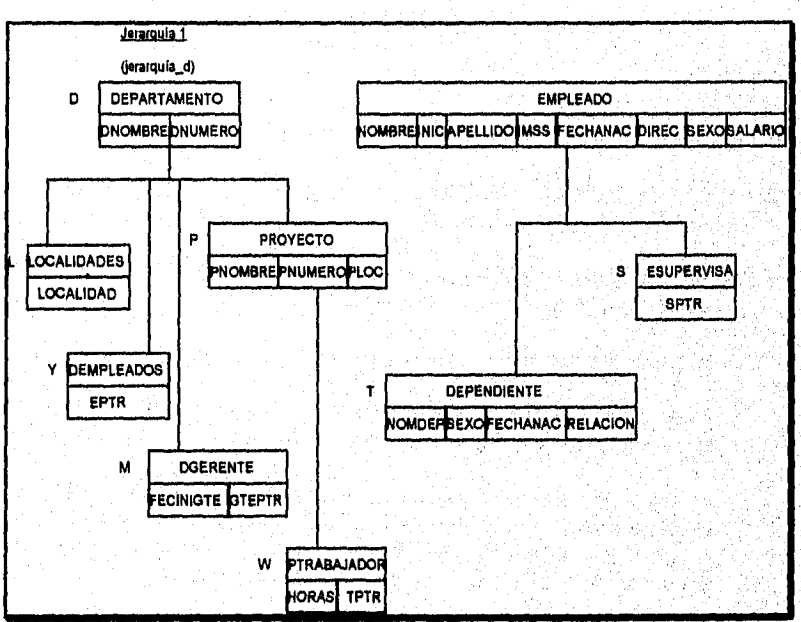


Figura 8.10 Esquema jerárquico para la base de datos COMPAÑIA, usando VPCR entre dos jerarquías para eliminar instancias.

En la Figura 8.11, a cada registro se le declara como raíz o padre único. Los elementos del registro se listan después junto con sus tipos de datos. Para los elementos de tipo *apuntador* debemos de especificar un padre. Los elementos declarados bajo la cláusula KEY están retenidos a tener valores únicos en cada registro. Cada cláusula KEY especifica una llave separada; y si una sola cláusula KEY lista a más de un campo, la combinación de estos valores de campo deben ser únicas para cada registro.

La cláusuala CHILD NUMBER especifica el orden de izquierda a derecha de un hijo debajo de su padre (real). La Figura 8.11 corresponde al orden mostrado en la Figura 8.10 y en la secuencia jerárquica es necesario especificar el orden de los sub-árboles hijo de cada uno de ellos bajo un padre.

La cláusula ORDER BY especifica el orden de registros individuales del mismo tipo en la secuencia jerárquica. Para un registro raiz, este especifica el orden de los árboles de ocurrencia. Por ejemplo, EMPLEADO se ordena alfabéticamente por APELLIDO, NOMBRE, asi los árboles de ocurrencia de estos registros están ordenados alfabéticamente por estos campos. Para registros no raíz, la cláusula ORDER BY especifica cómo deben ordenarse los registros *dentro de cada padre*, especificando un campo llamado **llave de secuencia**.

8.5 Diseño de Bases de Datos Jerárquicas Usando Mapeo ER-Jerárquico

En el modelo jerárquico, sólo las relaciones tipo 1:N pueden representarse en una jerarquía particular como relaciones padre-hijo PCR. Además, un registro puede tener al menos una padre (real); por lo tanto, las relaciones M:N son difíciles de representar. Formas posibles de representar relaciones M:N en una base de datos jerárquicas incluyen las siguientes:

- Representar a la relación M:N como si fuera 1:N. En este caso, las instancias del lado N se duplican debido a que cada registro puede relacionarse con varios padres. Esta representación guarda a todos los registros en una sola jerarquía *al costo de duplicar las instancias*. Los programas que actualizan a la base de datos deben de mantener la consistencia de las copias.

- Crear más de una jerarquía y tener relaciones virtuales padre-hijo (PCR) (apuntadores lógicos) de un registro que aparece en una jerarquía a la raiz de otra.

```
SCHEMA NAME = COMPAÑIA

HIERARCHIES = HIERARCHY1, HIERARCHY2

RECORD
        NAME = EMPLEADO
        TYPE = ROOT OF HIERARCHY2
        DATAITEMS =
                NOMBRE          CHARACTER 15
                INIC            CHARACTER 1
                APELLIDO        CHARACTER 15
                IMSS            CHARACTER 9
                FECHANAC        CHARACTER 9
                DIRECCION       CHARACTER 30
                SEXO            CHARACTER 1
                SALARIO         CHARACTER 10
        KEY = IMSS
        ORDER BY APELLIDO, NOMBRE
```

Figura 8.11 Declaraciones para el esquema jerárquico en la Figura 8.10

```
RECORD
        NAME = DEPTO
        TYPE = ROOT OF HIERARCHY1
        DATAITEMS =
                DNOMBRE         CHARACTER 15
                DNUMERO         INTEGER
        KEY = DNOMBRE
        KEY = DNUMERO
        ORDER BY DNOMBRE

RECORD
        NAME = DLOCS
        PARENT = DEPTO
        CHILD NUMBER = 1
        DATA ITEMS =
                LOC     CHARACTER 15

RECORD
        NOMBRE = GTEDEPTO
        PARENT = DEPTO
        CHILD NUMBER = 3
        DATA ITEMS =
                FECINIGTE       CHARACTER 9
                GTEPTR          POINTER WITH VIRTUAL PARENT = EMPLEADO

RECORD
        NAME = PROJECT
        PARENT = DEPTO
        CHILD NUMBER = 4
        DATA ITEMS =
                PNOMBRE         CHARACTER 15
                PNUMERO         INTEGER
                PLOC            CHARACTER 15
        KEY = PNOMBRE
        KEY = PNUMERO
        ORDER BY PNOMBRE

RECORD
        NAME = PTRABAJADOR
        PARENT = PROYECTO
        CHILD NUMBER = 1
        DATA ITEMS =
                HORAS   CHARACTER 4
                TPTR    POINTER WITH VIRTUAL PARENT = EMPLEADO
```

```
RECORD
        NAME=DEMPLEADO
        PARENT = DEPTO
        CHILD NUMBER = 2
        DATA ITEMS =
                EPTR      POINTER WITH VIRTUAL PARENT = EMPLEADO

RECORD
        NAME = DEPENDIENTE
        PARENT = EMPLEADO
        CHILD NUMBER = 1
        DATA ITEMS =
                NOMDEP          CHARACTER 15
                SEXO            CHARACTER 1
                FECHANAC        CHARACTER 9
                RELACION        CHARACTER 10
        ORDER BY DESC FECHANAC
```

Figura 8.11 (continuación) Declaraciones para el esquema jerárquico en la Figura 8.10

Debido a que se consideraron varias opciones, no hay método estandar para mapear un esquema ER a esquema jerárquico. En las Figuras 8.12(a) y (b) ilustraremos las dos posibilidades discutidas anteriormente, que pueden usarse para representar el esquema ER mostrado en la Figura 3.2. En la Figura 8.10 se muestra una tercer alternativa.
La Figura 8.12(a) muestra una sola jerarquía que puede usarse para representar el esquema ER de la Figura 3.2. Elegimos a DEPTO como la raíz de la jerarquía. Las relaciones 1:N TRABAJA_PARA y CONTROLA y la relación 1:1 MANEJA se representan en el primer nivel de la jerarquía por los registros EMPLEADO, PROYECTO, y GTEDEPTO, respectivamente. Sin embargo, para limitar la redundancia, sólo guardamos los atributos de un empleado quien es gerente en el registro DEPGTE. Los EMPLEADOS en un árbol jerárquico propiedad de un DEPTO en particular representarán a los empleados que trabajan para ese departamento. Similarmente, los registros PROYECTO repesentarán a los proyectos controlados por ese departamento, y el registro DEPGTE representará a todos los emplados que manejan el departamento. Un empleado que es gerente se representa dos veces -una como instancia de EMPLEADO, y la segunda como instancia de DEPGTE. Los programas de aplicación son responsables de mantener estas copias en forma consistente.

La relación 1:N DEPENDE_DE se representa por el registro DEPENDIENTE como subordinado de EMPLEADO. La relación M:N TRABAJA_EN se representa como subordinada de PROYECTO, pero sólo se incluyen ENOMBRE y EIMSS en TRABAJA_EN, junto con el atributo de la relación HORAS. Un empleado que trabaja en diferentes proyectos se almacenará en varias copias de instancias TRABAJA_EN con valores idénticos de ENOMBRE y EIMSS. Para limitar la redundiancia, el resto de la información de empleados no se duplica en TRABAJA_EN. Notemos que cada registro de TRABAJA_EN representa a uno de los empleados que trabajan en un proyecto en particular. Alternativamente, podríamos representar a TRABAJA_EN como subordinado de EMPLEADO; en este caso, cada registro TRABAJA_EN representará a uno de los proyectos en los que un EMPLEADO trabaja, y sus campos serían PNOMBRE, PNUMERO y HORAS, como se mostró por la caja punteada TRABAJA_EN en la Figura 8.12(a). En el último caso, la información de PROYECTO se duplicaría en varias copias de registros TRABAJA_EN.

Finalmente, la relación SUPERVISION se representa como un subordinado de EMPLEADO. Podríamos elegir representarla en el rol de supervisor o de supervisado. En la Figura 8.12(a) cada registro SUPERVISOR representa al supervisor propiedad de EMPLEADO en la jerarquía, así la relación jerárquica representa el rol del supervisor; cada empleado tiene un hijo SUPERVISOR representando a su supervisor directo. Alternativamente, podríamos representar el rol de un supervisado en la relación jerárquica; entonces cada registro EMPLEADO que representa a un supervisor se relacionaría con muchos supervisados directos como registros hijo. Esto se muestra en líneas punteadas en la caja SUPERVISA de la Figura 8.12(a). En cualquier caso, la información de EMPLEADO se repite en el registro de SUPERVISOR o SUPERVISA, así que sólo se incluyen unos cuantos atributos de EMPLEADO.

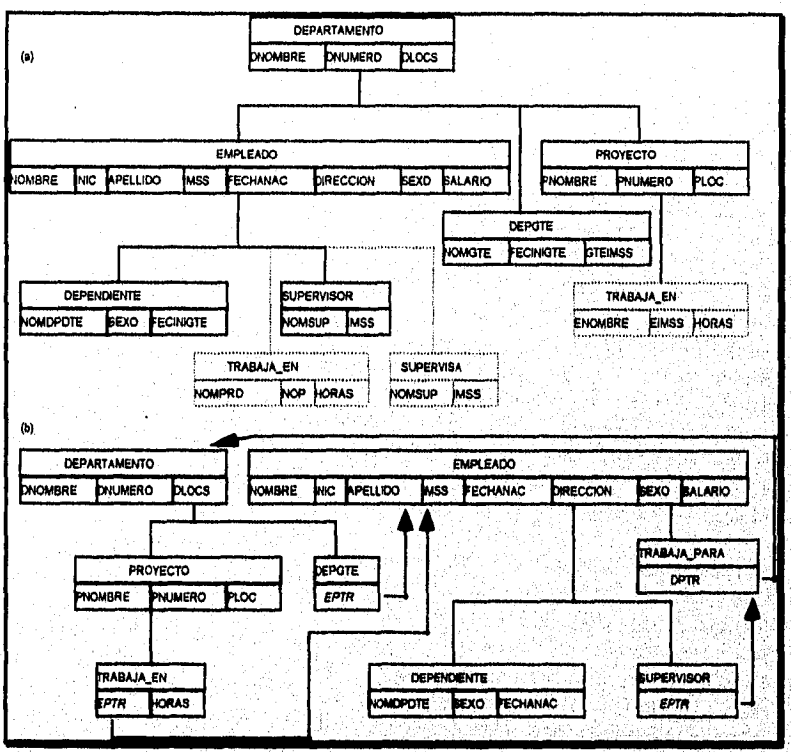


Figura 8.12 Mapeo del esquema ER de la Figura 3.2 para un modelo jerárquico. (a) Un esquema de base de datos jerárquica para la base de datos COMPAÑIA con una sola jerarquía. (b) Otro esquema jerárquico para la misma base de datos con dos jerarquías y cuatro VPCR.

En la Figura 8.12(a) existe una repetición excesiva de la información de empleado en los registros EMPLEADO, DEPGTE, TRABAJA_EN y SUPERVISOR, debido a que todos

ellos representan empleados en distintos roles. De alguna manera podemos limitar la repetición limitando el esquema ER de la Figura 3.2 con dos o más jerarquías. En la Figura 8.12(b) se usaron dos jerarquías. La primera tiene a DEPTO como registro raíz y representando los tipos de relación CONTROLA, MANEJA y TRABAJA_EN. La segunda jerarquía tiene a EMPLEADO como registro raíz y representa a las relaciones DEPENDE_DE y SUPERVISION. Usando relaciones y apuntadores virtuales padre-hijo en los registros TRABAJA_EN, GTEDEP, y SUPERVISOR, no duplicamos ninguna información del empleado. Cada apuntador apunta a un registro EMPLEADO, pero EMPLEADO se almacena una sola vez como raíz de la segunda jerarquía. La información TRABAJA_PARA se almacena una sola vez como raíz de la segunda jerarquía. La relación TRABAJA_PARA se representa por un hijo del registro EMPLEADO llamado TRABAJA_PARA con padre virtual DEPTO.

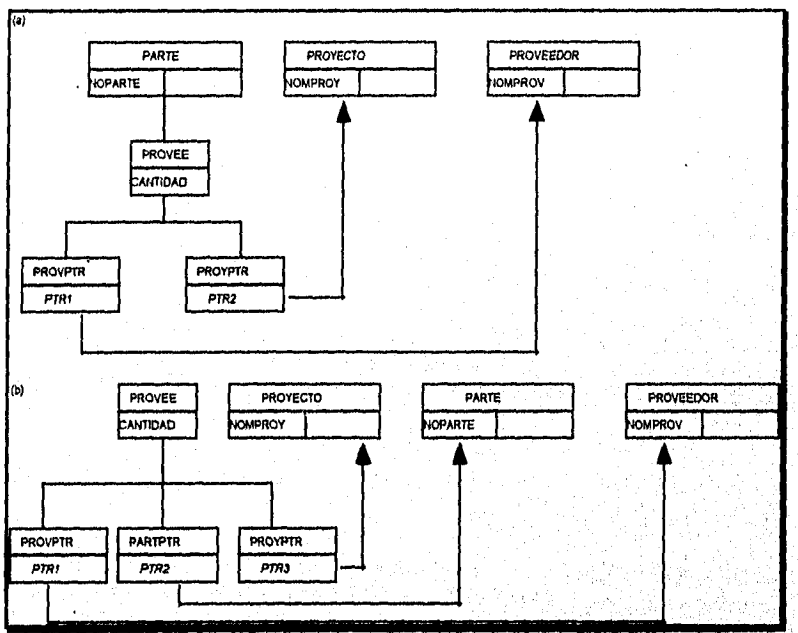


Figura 8.13 Mapeo de una relación n-aria (n = 3) tipo PROVEE de ER a jerárquica. (a) Una opción de representar una relación ternaria. (b) Representación de una relación ternaria usando tres VPCR.

Finalmente, consideremos el mapeo de relaciones n-arias, n > 2. La Figura 8.13 muestra dos opciones para mapear la relación ternaria PROVEE de la Figura 3.16(a). Debido al argumento, derivado de IMS, de que un registro puede tener al menos un padre virtual, no podemos colocar PROVEE debajo de PARTE, e incluir dos apuntadores a dos padres virtuales PROYECTO y PROVEEDOR. La opción en la Figura 8.13(a) crea *dos registros apuntadores* debajo de PROVEE con padres virtuales PROYECTO y PROVEEDOR.

Otra opción, mostrada en la Figura 8.13(b), tiene a PROVEE como raíz de la jerarquía y crea tres tipos de registro apuntador debajo de él para apuntar a los registros participantes como padres virtuales. Esta opción es la más flexible.

El modelo jerárquico se considera generalmente inferior en su capacidad de modelaje al modelo relacional y de red, por las siguientes razones:

- Los tipos de relación M:N sólo pueden representarse agregando registros redundantes o usando relaciones virtuales padre-hijo y registros apuntador.
- Todas las relaciones 1:N en una jerarquía deben mantenerse en la misma dirección.
- Un registro en una jerarquía debe tener al menos un propietario.
- Un registro puede tener como máximo dos padres -uno real y uno virtual. (Esta limitante es específica a IMS)

8.6 Lenguaje de Manipulación de Datos para el Modelo Jerárquico

Ahora discutiremos el **HDML** (Lenguaje de Manipulación de Datos Jerárquico), el cual es un lenguaje de registro a la vez para la manipulación de bases de datos jerárquicas. Los comandos del lenguaje deben alojarse en un lenguaje de programación de propósito general, llamado lenguaje **huésped**. A pesar de que es más común tener COBOL o PL/1 como lenguaje huésped, usaremos PASCAL. Notemos que HDML *no es* un lenguaje para un DBMS jerárquico en particular; se introdujo para ilustrar los conceptos de manipulación de bases de datos jerárquicas. Empezaremos introduciendo los conceptos de área de trabajo del usuario para comunicación con el sistema, junto con algunos conceptos de circulación.

8.6.1 Area de Trabajo del Usuario (UWA) y Conceptos de Circulación para el Uso de Comandos HDML

En un lenguaje de registro a la vez, una operación de recuperación recupera registros en **variables de programa**. En nuestros ejemplos, los registros se recuperan en variables de PASCAL. El programa puede entonces referir a las variables para accesar los valores de los registros. Asumimos que ha sido declarado un registro para cada tipo del esquema de la Figura 8.10. Las variables en PASCAL mostradas en la Figura 8.14, usan los mismos *nombres de campo* que los del esquema de la base de datos de la Figura 8.10, mientras que los *nombres de los registros* están prefijados con una P_. Estas variables existen en lo que se llama **área de trabajo del usuario**. Notemos que es posible tener estas variables declaradas automáticamente refiriendo al esquema de la Figura 8.11. Inicialmente, los valores de estas variables son indefinidos.

El HDML se basa en el concepto de **secuencia jerárquica** definida anteriormente. Después de cada comando, al último registro accesado se le llama el **registro actual de la**

base de datos. El DBMS mantiene un apuntador al registro actual. Los comandos subsecuentes proceden *del registro actual* y pueden definir a un nuevo registro actual, dependiendo del tipo de comando. Por tanto, los comandos HDML recorren la base de datos recuperando los registros requeridos por el query. Originalmente, el registro actual es un "registro imaginario" localizado antes de la raíz del primer árbol de ocurrencia en la base de datos.

```
var P_EMPLEADO:        record
                           NOMBRE: packed array [1..15] of char;
                           INIC: char;
                           APELLIDO: packed array [1..15] of char;
                           IMSS: packed array [1..9] of char;
                           FECHANAC: packed array [1..9] of char;
                           DIRECCION: packed array [1..30] of char;
                           SEXO: char;
                           SALARIO: packed array [1..10] of char;
                           end;

P_DEPTO :              record
                           DNOMBRE: packed array [1..15] of char;
                           DNUMERO: integer;
                           end;

P_DLOC:                record
                           LOC: packed array [1..15] of char;
                           end;

P_DGTE:                record
                           FECINIGTE: packed array [1..9] of char;
                           GTEPTR: apuntador de base de datos a EMPLEADO
                           end;

P_PROYECTO:            record
                           PNOMBRE: packed array [1..15] of char;
                           PNUMERO: integer;
                           PLOC: packed array [1..15] of char;
                           end;

P_PTRABAJADOR:         record
                           HORAS: packed array [1..4] of char;
                           TPTR: apuntador de base de datos a EMPLEADO
                           end;

P_DEMPLEADOS:          record
                           EPTR:    apuntador de base de datos a EMPLEADO
                           end;

P_ESUPERVISA:          record
                           SPTR:    apuntador de base de datos a EMPLEADO
                           end;

P_DEPENDIENTE:         record
                           NOMDEP: packed array [1..15] of char;
                           SEXO: char;
                           FECHANAC: packed array [1..9] of char;
                           RELACION: packed array [1..10] of char;
                           end;
```

Figura 8.14 Varibles PASCAL en el UWA correspondientes a parte del esquema jerárquico de la Figura 8.10.

Si una base de datos tiene más de un esquema jerárquico en él, y estas jerarquías se procesan en conjunto, el sistema IMS permite la definición de una vista para crear un

esquema hecho de trozos que incluyan los tipos de registros deseados conectados por VPCR. Tal vista es tratada como un **esquema jerárquico** sencillo y tiene su propio **registro actual**. *Cada esquema jerárquico* tiene su propio **registro de jerarquía actual**. Los comandos HDML refieren implícitamente a estos tres tipos de **indicadores de actualidad**:

- Actualidad de la base de datos.
- Actualidad de jerarquía para cada esquema jerárquico.
- Actualidad de registro para cada tipo.

La programación de un registro a la vez requiere interacción continua entre el usuario del programa y el DBMS. Al final de cada comando debe comunicarse el **status de información** de vuelta al programa. Esto se logra a través de una variable llamada **DB_STATUS**, cuyo valor se coloca por el software del DBMS después de que se ejecuta cada comando. Asumiremos que un valor de 0 especifica que el último comando fue ejecutado exitosamente.

Los comandos HDML pueden categorizarse como comandos de recuperación, actualización, y retensión de actualidad. Los **comandos de recuperación** recuperan a uno o más registros en las variables correspondientes de programa y pueden cambiar algunos indicadores de actualidad. Los **comandos de actualización** se usan para insertar, borrar y modificar registros. Los **comandos de retensión de actualidad** se usan para marcar al registro actual de tal forma que pueda actualizarse o borrarse por un comando subsecuente. En la Tabla 8.1 se resumen los comandos HDML.

Ahora discutiremos cada uno de estos comandos e ilustraremos nuestra discusión con ejemplos basados en el esquema mostrado en la Figura 8.10. En los segmentos de programa, los *comandos HDML están prefijados con un signo $* para distinguirlos de las instrucciones en PASCAL. Las instrucciones en PASCAL -tales como *if,then, while*, y *for*- se escriben en minúsculas.

8.6.2 El Comando GET

El comando HDML para recuperar un registro es el comando GET. Existen muchas variantes de GET; la estructura de dos de estas variantes es como sigue, con las partes opcionales entre paréntesis [...]:

- GET FIRST <nombre del registro> [WHERE <condición>]
- GET NEXT <nombre del registro> [WHERE <condición>]

Tabla 8.1 Resumen de Comandos HDML

RECUPERACION	
GET	RECUPERA UN REGISTRO A LA VARIABLE DE PROGRAMA CORRESPONDIENTE Y LA HACE EL REGISTRO ACTUAL. LAS VARIANTES INCLUYEN GET FIRST, GET NEXT, GET NEXT WITHIN PARENT, Y GET PATH.
ACTUALIZACION DE REGISTROS	
INSERT	ALMACENA UN NUEVO REGISTRO EN LA BASE DE DATOS Y LO HACE EL REGISTRO ACTUAL.
DELETE	BORRA AL REGISTRO ACTUAL (Y A SU SUB-ARBOL) DE LA BASE DE DATOS
REPLACE	MODIFICA ALGUNOS CAMPOS DEL REGISTRO ACTUAL.
RETENSION DE ACTUALIDAD	
GET HOLD	RECUPERA UN REGISTRO Y LO RETIENE COMO EL ACTUAL DE TAL FORMA QUE SUBSECUENTEMENTE PUEDA SER BORRADO O REEMPLAZADO.

La variante más sencilla es **GET FIRST**, la cual siempre inicia la búsqueda al *inicio de la secuencia jerárquica* hasta encontrar el primer registro de <nombre del registro> que satisface <condición>. Este registro también se convierte en el registro actual, jerarquía actual, y recupera a la variable UWA correspondiente. Por ejemplo, para recuperar al "primer" EMPLEADO en la secuencia jerárquica cuyo nombre es John Smith, escribimos EX1:

EX1: **$GET FIRST** EMPLEADO **WHERE** NOMBRE = 'John' **AND** APELLIDO = 'Smith'

El DBMS utiliza la condición después de WHERE para buscar al primer registro en el orden de la secuencia jerárquica que satisfaga la condición y sea del tipo especificado. El valor de DB_STATUS se pone en 0 si el registro es *encontrado exitosamente*; de otro modo, DB_STATUS se pone en algún otro valor -digamos, 1- que indica *no encontrado*. Mediante diferentes valores de DB_STATUS se indican otros errores o excepciones.

Si más de un registro satisface la condición WHERE y deseamos recuperar a todos ellos, debemos escribir un ciclo en el programa huésped y usar el comando GET NEXT. Suponemos que GET NEXT inicia su búsqueda desde el *registro actual del tipo especificado* en *GET NEXT* y busca hacia adelante en la secuencia jerárquica para encontrar otro registro que satisface la condición WHERE. Por ejemplo, para recuperar los registros de todos los EMPLEADOS cuyo salario es menor que $20000 y obtener un listado de sus nombres, podemos escribir el segmento de programa mostrado en EX2:

```
EX2:  $GET FIRST EMPLEADO WHERE SALARIO < '20000';
      while DB_STATUS = 0 do
            begin
            writeln(P_EMEPLEADO.NOMBRE, P_EMPLEADO.APELLIDO);
            $GET NEXT EMPLEADO WHERE SALARIO < '20000'
            end;
```

En EX2, el ciclo while continúa hasta que no más registros de la base de datos satisfacen la condición WHERE; por lo tanto, la búsqueda llega hasta el último registro en la base de datos (secuencia jerárquica). Cuando ya no se encuentran más registros, DB_STATUS se vuelve diferente de cero, "fin de la base de datos", y finaliza el ciclo while. Notemos que la condición WHERE en los comandos GET es opcional. Si no hay condición, se recupera el siguiente registro en la secuencia jerárquica. Por ejemplo, para recuperar todos los EMPLEADOS en la base de datos, podemos usar EX3:

```
EX3:  $GET FIRST EMPLEADO
      while DB_STATUS = 0 do
            begin
            writeln(P_EMPLEADO.NOMBRE, P_EMPLEADO.APELLIDO);
            $GET NEXT EMPLEADO
            end;
```

8.6.3 Los Comandos GET PATH y GET NEXT WITHIN PARENT

Hasta ahora hemos considerado la recuperación de registros aislados usando el comando GET. Pero cuando se tiene que localizar un registro en las profundidades de la jerarquía, puede usarse la recuperación sobre una serie de condiciones sobre los registros junto a la trayectoria jerárquica completa. Para lograr esto, introducimos el comando GET PATH:

GET (FIRST| NEXT) PATH <trayectoria jerárquica> [WHERE <condición>]

Aquí, <trayectoria jerárquica> es una lista de los registros que inician desde la raíz junto con una trayectoria en el esquema jerárquico, y <condición> es una expresión Booleana que especifica las condiciones sobre los registros individuales junto con la trayectoria. Debido a que pueden especificarse varios registros, los nombres de los campos son prefijados con los nombres de los registros en <condición>. Por ejemplo, consideremos el siguiente query: "Listar el apellido y fechas de nacimiento de todos los pares empleado-dependiente, donde ambos tienen el nombre John". Esto se muestra en EX4:

```
EX4:  $GET FIRST PATH EMPLEADO, DEPENDIENTE
      WHERE EMPLEADO.NOMBRE = 'John' AND DEPENDIENTE.NOMBRE =
'John'
      while DB_STATUS = 0 do
            begin
                  writeln(P_EMPLEADO.NOMBRE, P_EMPLEADO.FECHANAC,
                        P_DEPENDIENTE.FECHANAC);
                  $GET NEXT PATH EMPLEADO, DEPENDIENTE
                  WHERE EMPLEADO.NOMBRE = 'John' AND
                  DEPENDIENTE.NOMBRE = 'John'
            end;
```

Suponemos que un comando GET PATH recupera *todos los registros junto con la trayectoria especificada* en las variables UWA, y el último registro junto a la trayectoria se vuelve el registro actual. Además, todos los registros junto a la trayectoria se vuelven *los registros actuales de sus tipos respectivos.*

Otro query común es encontrar todos los registros de un tipo dado que tienen al *mismo registro padre.* En este caso necesitamos el comando GET NEXT WITHIN PARENT, el cual puede usarse para hacer ciclos a través de los hijos de un padre y tiene el siguiente formato:

GET NEXT <nombre registro hijo>
WITHIN [VIRTUAL] PARENT [<nombre del registro padre]
[WHERE <condición>]

Este comando recupera al siguiente registro hijo buscando hacia adelante del *registro hijo actual* para el siguiente hijo *propiedad del padre actual.* Si no se encuentran más hijos, DB_STATUS se coloca en un valor diferente de cero para indicar que "ya no existen más registros del hijo especificado que tienen al mismo padre que el padre actual". El <nombre del registro padre> es *opcional*, y el default es el padre inmediato de <nombre registro hijo>. Por ejemplo, para recuperar los nombres de todos los proyectos controlados por el departamento 'Investigación', podemos escribir el segmento de programa mostrado en EX5:

```
EX5:  $GET FIRST PATH DEPTO, PROYECTO
      WHERE DNOMBRE = 'Investigación';
      while DB_STATUS = 0 do
            begin
            writeln(P_PROYECTO.PNOMBRE);
            $GET NEXT PROYECTO WITHIN PADRE
            end;
```

En EX5, podemos escribir "WITHIN PARENT DEPTO" en lugar de sólo "WITHIN PARENT" en el comando GET NEXT con el mismo efecto, debido a que DEPTO es el padre inmediato de PROYECTO. Sin embargo, si queremos recuperar todos los registros propiedad de un padre que *no es el padre inmediato* -por ejemplo, todos los registros PTRABAJADOR propiedad del mismo DEPTO- debemos entonces especificar a DEPTO como el padre en la cláusula "WITHIN PARENT".

Notemos que hay dos métodos principales para establecer explícitamente a un padre como el registro actual:

- Si usamos GET FIRST o GET NEXT, el registro recuperado se convierte en el padre actual.

- Si usamos GET PATH, se establece una trayectoria jerárquica de los registros padre actuales. Como se mostró en EX5, este también puede recuperar al *primer hijo*, de tal forma que puedan usarse los comandos GET NEXT WITHIN PARENT.

Podemos reescribir EX4 sin el comando GET PATH usando un ciclo para encontrar EMPLEADOS con NOMBRE = 'John' y un ciclo anidado usando GET NEXT WITHIN PARENT para encontrar cualquier DEPENDIENTE de cada EMPLEADO con DEPNAME = 'John'. Sin embargo, el comando GET PATH nos permite hacer esto más directamente y con un *menor número de llamadas* al DBMS.

Para localizar al padre real o virtual del registro actual *de un hijo determinado* puede usarse otra variante del comando GET:

GET [VIRTUAL] PARENT <nombre del registro>
OF <registro hijo>

Por ejemplo, para recuperar los nombres y horas a la semana de cada empleado que trabaja en "ProyectoX", podemos usar el comando GET PARENT, como en EX6:

```
EX6:  $GET FIRST PATH DEPTO, PROYECTO, PTRABAJADOR
      WHERE PNOMBRE = 'ProyectoX'
      while DB_STATUS = 0 do
            begin
            $GET VIRTUAL PARENT EMPLEADO OF PTRABAJADOR;
            if DB_STATUS = 0 then
                  writeln(P_EMPLEADO.APELLIDO, P_EMPLEADO.NOMBRE,
                  P_PTRABAJADOR.HORAS);
            else writeln("error --EMPLEADO no tiene padre virtual");
            $GET NEXT PTRABAJADOR WITHIN PARENT PROYECTO
            end;
```

Notemos que se puede usar una condición WHERE con el comando GET NEXT WITHIN PARENT. Por ejemplo, para recuperar los nombres de los empleados que trabajan más de cinco horas a la semana en el 'ProyectoX', podemos modificar el comando GET NEXT PTRABAJADOR WITHIN PARENT en EX6 a:

```
$GET NEXT PTRABAJADOR WITHIN PARENT PROYECTO WHERE HORAS >
'5.0'
```

También debemos modificar el comando GET FIRST PATH apropiadamente. Tal como se nos permita recorrer un VPCR de hijo a padre, como se ilustró en EX6, también podemos viajar de padre a hijo, usando la siguiente modificación del comando:

GET NEXT <registro virtual hijo>
WITHIN PARENT <nombre padre virtual>

8.6.4 Cálculo de Funciones Agregadas

Usando las facilidades del programa huésped, deben implementarse explícitamente por el programador las funciones agregadas tales como COUNT y AVERAGE. Por ejemplo, para calcular el número de empleados que trabajan en cada deprtamento y su salario promedio, podemos escribir EX7:

```
EX7:  $GET FIRST PATH DEPTO, DEMPLEADOS;
      while DB_STATUS = 0  do
            begin
            total_sal:=0; no_de_empleados:=0;
            writeln(P_DEPTO.NOMBRE);
            while DB_STATUS = 0 do
                  begin
                  $GET VIRTUAL PARENT EMPLEADO;
                  total_sal:=total_sal + conv_sal(P_EMPLEADO.SALARIO);
                  no_de_empleados:= no_de_emepleados + 1;
                  $GET NEXT DEMPLEADOS WITHIN PARENT DEPTO
                  end;
            writeln('no de empleados =', no_de_empleados, 'salario promedio =',
            total_sal/no_de_empleados);
            $GET NEXT PATH DEPTO, DEMPLEADOS
            end;
```

8.6.5 Comandos HDML para Actualización

Los comandos HDML para actualizar una base de datos jerárquica se muestran en la Tabla 8.1. El comando INSERT se usa para ingresar un nuevo registro. Antes de insertar un registro de algún tipo en particular, debemos colocar primero los valores de los campos del nuevo registro en el área de trabajo del usuario. Por ejemplo, supongamos que queremos insertar un nuevo EMPLEADO para John F. Smith; podemos usar el segmento de programa en EX8:

```
EX8:  P_EMPLEADO.NOMBRE:='John';
      P_EMPLEADO.APELLIDO:='Smith';
      P_EMPLEADO.INIC:='F';
      P_EMPLEADO.IMSS:='567342739';
      P_EMPLEADO.DIRECCION:='40 NW 80 Blvd. Gainesville, Florida, 32607';
      P_EMPLEADO.FECHANAC:='10-ENE-55';
      P_EMPLEADO.SEXO:='M';
      P_EMPLEADO.SALARIO:='30000.00';
      $INSERT EMPLEADO FROM P_EMPLEADO;
```

El registro recién insertado también se convierte en el registro actual de la base de datos, su esquema jerárquico, y su tipo de registro. Si es un registro raíz, como en EX8, se crea

una nueva ocurrencia jerárquica con el nuevo registro como raíz. El registro se inserta en la secuencia jerárquica en el orden especificado por cualquier campo ORDER BY en la definición del esquema. Por ejemplo, el nuevo EMPLEADO en EX8 se inserta en orden alfabético de su valor combinado de APELLIDO, NOMBRE, de acuerdo a la definición del esquema en la Figura 8.11. Si no se especifican campos de ordenamiento en la definición del registro raíz de un esquema jerárquico, un nuevo registro raíz se inserta siguiendo al árbol de ocurrencia que contenía al registro actual de la base de datos antes de la inserción.

Para insertar un registro hijo, debemos hacer que su padre, o uno de sus registros hermanos, sea el *registro actual* del esquema jerárquico antes de emitir el comando INSERT. También se debe colocar cualquier apuntador virtual padre antes de insertar el registro. Para hacerlo, necesitamos un comando SET VIRTUAL PARENT, el cual coloca el campo apuntador en la variable de programa al registro virtual del padre actual. El registro se inserta después de encontrar un lugar apropiado para él en la secuencia jerárquica. Por ejemplo, supongamos que queremos relacionar al EMPLEADO insertado en EX8 como un trabajado de 40 horas a la semana en el proyecto cuyo número es 55; podemos usar EX9:

```
EX9:  $GET FIRST EMPLEADO WHERE IMSS = '567342739';
      if DB_STATUS = 0 then
            begin
            P_TRABAJADOR.TPTR:= SET VIRTUAL PARENT;
            P_PTRABAJADOR.HORAS:='40.0';
            $GET FIRST PROYECTO WHERE PNUMERO = 55;
            if DB_STATUS = 0 then $INSERT PTRABAJADOR FROM
            P_PTRABAJADOR;
            end;
```

Para borrar un registro de la base de datos, primero lo hacemos el registro actual y luego usamos el comando **DELETE**. **GET HOLD** se usa para hacer al registro el actual, donde la instrucción HOLD indica al DBMS que el programa borrará o actualizará al registro recién recuperado. Por ejemplo, para borrar todos los EMPLEADOS barones, podemos usar EX10, el cual también lista los nombres de los empleados borrados *antes* de borrar sus registros:

```
EX10: $GET HOLD FIRST EMPLEADO WHERE SEXO = 'M';
      while DB_STATUS = 0 do
            begin
            writeln(P_EMPLEADO.APELLIDO, P_EMPLEADO.NOMBRE);
            $DELETE EMPLEADO;
            $GET HOLD NEXT EMPLEADO WHERE SEX = 'M';
            end;
```

Notemos que el borrar un registro significa automáticamente borrar a todos sus descendientes -todos los registros en el sub-árbol. Sin embargo, los hijos virtuales en otras jerarquías no son borrados. En efecto, antes de borrar un registro, el DBMS debe asegurarse que no hay hijos virtuales que apunten a él. Después de un comando DELETE exitoso, el registro actual se convierte en la secuencia jerárquica "posición vacía" correspondiente al registro recién vaciado. Desde esa posición continúan las operaciones subsecuentes.

Para modificar los valores de campo de un registro, seguimos los siguientes pasos:

1. Hacer del registro a modificar el registro actual, y recuperarlo a la variable de programa UWA usando el comando GET HOLD.

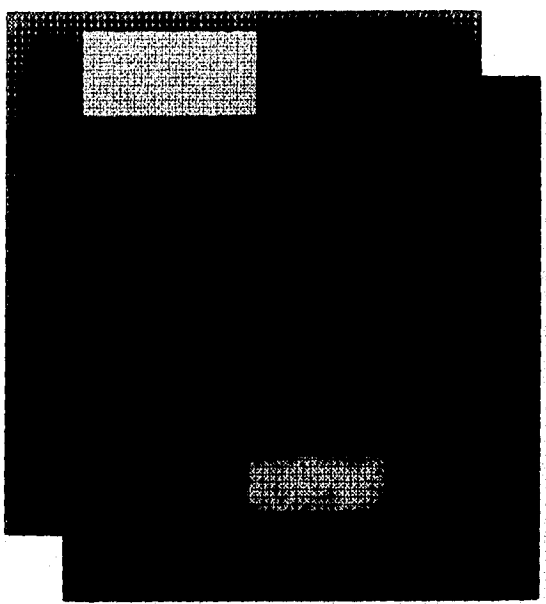
2. Modificar los campos deseados en la variable UWA.

3. Usar el comando **REPLACE**.

Por ejemplo, para dar un 10% de aumento a todos los empleados de 'Investigación', podemos usar el programa mostrado en EX11:

```
EX11: $GET FIRST PATH DEPTO, EMPLEADOS
      WHERE DNOMBRE = 'Investigación'
      while DB_STATUS = 0 do
            begin
            $GET HOLD VIRTUAL PARENT EMPLEADO OF DEMPLEADOS;
            P_EMPLEADO.SALARIO:=P_EMPLEADO.SALARIO * 1.1;
            $REPLACE EMPLEADO FROM P_EMPLEADO;
            $GET NEXT DEMPLEADOS WITHIN PARENT DEPTO
            end;
```

CAPITULO 9



Dependencias Funcionales y Normalización de Bases de Datos Relacionales

En los Capítulos anteriores, presentamos varios aspectos del modelo relacional, incluyendo ejemplos de bases de datos relacionales. Cada *esquema de relación* consta de un número de atributos, y el *esquema relacional* se compone de un número de esquemas de relación. Hasta ahora, hemos asumido que los atributos se agrupan para formar un esquema de relación usando el sentido común del diseñador o mapeando un esquema especificado en el modelo Entidad-Relación a esquema relacional. Sin embargo, no tenemos ningún parámetro formal del por qué un agrupamiento de atributos en un esquema de relación puede ser mejor que otro. No existía otro parámetro de la propiedad o calidad del diseño, que la intuición del diseñador.

En este capítulo discutiremos algo de la teoría que ha sido implementada para intentar elegir "buenos" esquemas de relación -esto es, parametrizar formalmente por qué un conjunto de agrupamiento de atributos es mejor que otro. Existen dos niveles sobre los cuales podemos discutir la "bondad" de los esquemas de relación. El primero es a **nivel lógico**, el cual se refiere a la forma en que los usuarios interpretan los esquemas de relación y el significado de sus atributos. Contar con buenos esquemas de relación a este nivel ayuda a los usuarios a entender claramente el significado de las tuplas en las relaciones, y por lo tanto formular los queries correctamente. El segundo es a **nivel de manipulación** (o **almacenamiento**), el cual se refiere a la forma en que las tuplas de una relación se almacenan y actualizan. Este nivel se aplica sólo a esquemas de relaciones -los cuales estarán físicamente almacenados como archivos- mientras que a nivel lógico nos interesamos en los esquemas de relaciones y vistas (relaciones virtuales). La teoría del diseño de bases de datos relacionales desarrollada en este capítulo se aplica principalmente a relaciones base, a pesar de que algunos criterios de propiedad también aplican a vistas.

Iniciaremos discutiendo informalmente algunos criterios de esquemas de relación buenos y malos. Después, definiremos el concepto de dependencia funcional, la cual es la herramienta principal en esquemas de relación para parametrizar formalmente la propiedad de atributos de agrupamiento, también se estudian y analizan las propiedades de las dependencias funcionales. Mostraremos cómo pueden usarse las dependencias funcionales para agrupar atributos en esquemas de relación que estén de forma normalizada. Un esquema de relación es una forma normalizada cuando satisface ciertas características funcionales. Pueden definirse formas normalizadas de esquemas de relación, llevando al mejoramiento progresivo de los agrupamientos.

9.1 Guías Informales para el Diseño de Esquemas de Relación

En esta sección discutiremos cuatro *parámetros informales* de calidad para el diseño de esquemas de relación:

- Semántica de los atributos.
- Reducción de los valores redundantes en las tuplas.
- Reducción de los valores nulos en las tuplas.

- Prohibición de tuplas falsas.

Como veremos, estos parámetros no son siempre independientes el uno del otro.

9.1.1 Semántica de los Atributos de la Relación

Cada vez que agrupamos atributos para formar un esquema de relación, suponemos que se asocia un cierto significado con los atributos. Anteriormente discutimos como puede interpretarse cada relación como un conjunto de hechos o instrucciones. Este significado, o **semántica**, especifica cómo interpretar los valores de atributos almacenados en una tupla de la relación -en otras palabras, cómo se relacionan los valores de los atributos de una tupla con otra. Mientras más sencillo sea explicar la semántica de la relación, mejor será el diseño del esquema de relación.

Para ilustrar esto, consideremos una versión simplificada del esquema de base de datos relacional COMPAÑIA, mostrado en la Figura 9.1. Las relaciones ejemplo pobladas de este esquema se muestran en la Figura 9.2. El significado del esquema de relación EMPLEADO es sencillo: cada tupla representa a un empleado, con valores para el nombre del empleado (ENOMBRE), número de seguro social (IMSS), fecha de nacimiento (FNAC), y dirección (DIR), y el número de departamento para el cual el empleado trabaja (DNUMERO). El atributo DNUMERO es una llave externa que representa una *relación implícita* entre EMPLEADO y DEPARTAMENTO. La semántica de los esquemas DEPARTAMENTO y PROYECTO son también directos; cada tupla DEPARTAMENTO representa a una entidad departamento, y cada tupla PROYECTO representa a una entidad proyecto. El atributo DGTEIMSS de DEPERTAMENTO relaciona a un departamento con el empleado que es su gerente, mientras que DNUM de PROYECTO relaciona a un proyecto con su departamento de control; ambos atributos son llaves externas.

La semántica de los otros dos esquemas de relación en la Figura 9.1 son ligeramente más complejos. Cada tupla en DEP_LOCS da un número de departamento (DNUMERO) y *una de* las localidades del departamento (DLOC). Cada tupla en TRABAJA_EN da el número de seguro social de un empleado (IMSS), el número de proyecto *de alguno* de los proyectos en que trabaja el empleado (PNUMERO), y el número de horas a la semana que ese empleado trabaja en cada proyecto (HORAS). Sin embrago, ambos esquemas tienen un significado bien definido, sin ambiguedades. El esquema DEP_LOCS representa un atributo multivaluado de DEPARTAMENTO, mientras que TRABAJA_EN representa una relación M:N entre EMPLEADO y PROYECTO. Por lo tanto, todos los esquemas de relación de la Figura 9.1 pueden considerarse buenos desde el punto de vista de tener una semántica clara. Para el diseño de un esquema de relación podemos establecer la siguiente guía:

GUIA 1: Diseñar un esquema de relación de tal forma que sea sencillo explicar su significado. No combinar atributos de distintos tipos de entidades y de relaciones en una sola relación. Intuitivamente, si un esquema de relación corresponde a un tipo de entidad o

de relación, el significado tiende a ser claro. De otro modo, tiende a ser una mezcla de varias entidades y relaciones y por lo tanto semánticamente poco claro.

Los esquemas de relación en las Figuras 9.3 (a) y (b) también tienen semántica clara. Una tupla en el esquema de relación EMP_DEP de la Figura 9.3(a) representa a un sólo empleado pero incluye información adicional -llamada, el nombre del departamento para el cual trabaja el empleado (DNOMBRE) y el número de seguro social del gerente del departamento (GTEIMSS). Para la relación EMP_PROY de la Figura 9.3(b), cada tupla relaciona un empleado a un proyecto pero también incluye el nombre del empleado (ENOMBRE), el nombre del proyecto (PNOMBRE), y la localidad del proyecto (PLOC). A pesar de que no existe ninguna contradicción con estas dos relaciones, se consideran diseños pobres porque violan la primer guía *mezclando atributos de distintas entidades del mundo real*; EMP_DEP mezcla atributos de empleados y departamentos, y EMP_PROY mezcla atrubutos de empleados y proyectos. Pueden ser utilizadas como vistas, pero ocasionarán problemas si se utilizan como relaciones base.

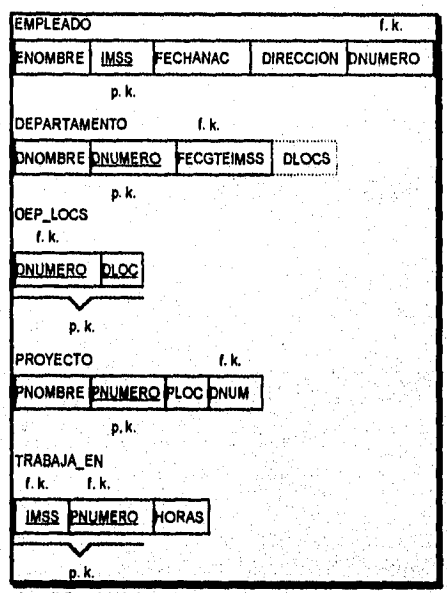


Figura 9.1 Versión simplificada del esquema de la base de datos relacional COMPAÑIA.

EMPLEADO

ENOMBRE	IMSS	FECHANAC	DIRECCION	DNUMERO
Smith, John B.	123456789	09-ENE-55	731 Fondren	5
Wong, Franklin T	333445555	08-DIC-45	638 Voss	5
Zelaya, Alicia J.	999887777	19-JUL-58	3321 Castle	4
Wallace, Jennifer	987654321	20-JUN-31	291 Berry	4
Narayan, Ramesh	666884444	15-SEP-52	975 Fire Oak	5

DEPARTAMENTO

DNOMBRE	DNUMERO	FECGTEIMSS
Investigación	5	333445555
Administración	4	987654321
Oficinas Centrales	1	888665555

DEP_LOCS

DNUMERO	DLOCS
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

TRABAJA_EN

IMSS	PNUMERO	HORAS
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
888665555	20	15.0

PROYECTO

PNOMBRE	PNUMERO	PLOC	DNUM
ProductoX	1	Bellaire	5
ProductoY	2	Sugarland	5
ProductoZ	3	Houston	5
Computación	10	Stafford	4
Reorganización	20	Houston	1
Nuevos Beneficios	30	Stafford	4

Figura 9.2 Ejemplo de relaciónes del esquema de la Figura 9.1

9.1.2 Información Redundante en Tuplas y Anomalías de Actualización

Una de las metas del diseño de esquemas es minimizar el espacio de almacenamiento que ocupan las relaciones base (archivos). El agrupamiento de atributos en esquemas de relación tiene un efecto significativo en el espacio de almacenamiento. Por ejemplo, comparemos el espacio utilizado por las dos relaciones base EMPLEADO y DEPARTAMENTO de la Figura 9.2 con el espacio de la relación base EMP_DEP de la Figura 9.4, la cual es resultado de aplicar la operación JOIN-NATURAL a EMPLEADO y

DEPARTAMENTO. En EMP_DEP, los valores de atributos pertienetes a un departamento en particular (DNUMERO, DNOMBRE, DGTEIMSS) se repiten para cada empleado que trabaja para ese departamento. En contraste, cada información del departamento aparece sólo en la relación DEPARTAMENTO de la Figura 9.2. En la relación EMPLEADO sólo se repite el número de departamento (DNUMERO) para cada empleado que trabaja en ese departamento. Comentarios similares aplican a la relación EMP_PROY (Figura 9.4), la cual incrementa a la relación TRABAJA_EN con atributos adicionales de EMPLEADO y PROYECTO.

Otro serio problema cuando se usan las relaciones de la Figura 9.4 como base son las **anomalías de actualización**. Estas pueden clasificarse en anomalías de inserción, de borrado, y de modificación.

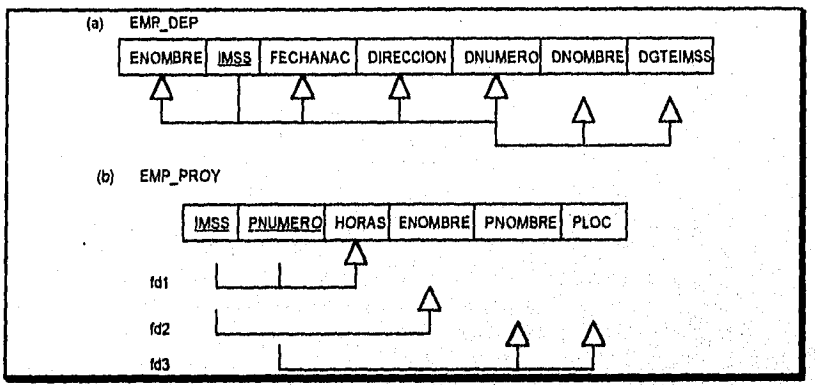


Figura 9.3 Dos esquemas de relación y sus dependencias funcionales. (a) Esquema de relación EMP_DEP. (b) Esquema de relación EMP_PROY.

EMP_DEP

ENOMBRE	IMSS	FECHANC	DIRECCN	DNUMERO	DNOMBRE	DGTEIMSS
Smith, John	123456789	09-ENE-55	731 Fondre	5	Investig.	333445555
Wong, Fran	333445555	08-DEC-45	638 Voss	5	Investig	333445555
Borg, James	888665555	10-NOV-27	450 Stone	1	Oficinas	888665555
Jabbar, Ah	987987987	29-MAR-59	980 Dallas	4	Admon	987654321
English, Joy	453453453	31-JUL-62	5631 Rice	5	Investig	333445555

EMP_PROY

IMSS	PNUMERO	HORAS	ENOMBRE	PNOMBRE	PLOC
123456789	1	32.5	Smith, John	ProductoX	Bellaire
123456789	2	7.5	Smith, John	ProductoY	Sugarland
666884444	3	40.0	Narayan, Ram	ProductoZ	Houston
453453453	1	20.0	English, Joy	ProductoX	Bellaire
888665555	20	null	Borg, James	Reorganizac	Houston

Figura 9.4 Ejemplo de relaciones para los esquemas de la Figura 9.3 que resultan de aplicar JOIN-NATURAL a las relaciones de la Figura 9.2.

Anomalías de Inserción. Estas pueden clasificarse en dos tipos, ilustrados por los siguientes ejemplos basados en la relación EMP_DEP:

- Para insertar una nueva tupla empleado en EMP_DEP, debemos incluir ya sea los valores de atributo para el departamento en que trabaja el empleado, o nulo (si el empleado no trabaja para ningun departamento aún). Por ejemplo, para insertar una nueva tupla para un empleado que trabaja en el departamento número 5, debemos ingresar los valores de atributos del departamento 5 correctamente de tal forma que sean *consistentes* con los valores del departamento 5 en otras tuplas de EMP_DEP. En el diseño de la Figura 9.2 no debemos preocuparnos de esta consistencia porque sólo ingresamos el número de departamento en la tupla de empleado; todos los valores de los atributos del departamento 5 están registrados una sola vez en la base de datos, como una sola tupla en la relación DEPARTAMENTO.

- Es difícil insertar un nuevo departamento que no tiene empleados a la relación EMP_DEP. La única forma de lograrlo es colocar valores nulos en los atributos de empleado. Esto ocasiona problemas ya que IMSS es la llave primaria de EMP_DEP, y cada tupla se supone representa a una entidad empleado -no a un departamento. Sin embrago, cuando el primer empleado se asigna a ese departamento, ya no necesitaremos a la tupla con valores nulos. Este problema no ocurre con el diseño de la Figura 9.2, porque un departamento se ingresa en la relación DEPARTAMENTO con o sin empleados que trabajen en él, y cada vez que se le asigna un empleado a ese departamento se inserta la tupla correspondiente en EMPLEADO.

Anomalías de Borrado. Este problema se relaciona con la segunda situación de anomalía de inserción discutida anteriormente. Si eliminamos a un empleado de EMP_DEP que resulta ser el último empleado de un departamento en particular, la información concerniente a ese departamento se pierde de la base de datos. Este problema no ocurre en la base de datos de la Figura 9.2 debido a que las tuplas de DEPARTAMENTO se almacenan independientemente.

Anomalías de Modificación. En EMP_DEP, si cambiamos el valor de uno de los atributos de un departamento en particular -digamos, el gerente del departamento 5- debemos actualizar las tuplas de todos los empleados que trabajan en ese departamento; de otro modo, la base de datos se volverá inconsistente. Si fallamos en la actualización de algunas tuplas, el mismo departamento será mostrado con dos valores diferentes de gerente en tuplas de empleado diferentes, lo cual no debe ocurrir.

Basados en las tres anomalías anteriores, podemos establecer la siguiente guía:

GUIA 2: Diseñar los esquemas de la relación base de tal forma que no ocurran anomalías de inserción, eliminación o modificación en las relaciones. Si cualquiera de las anomalías

está presente, anotémoslas claramente de tal forma que los programas que actualizan a la base de datos operen correctamente.

La segunda guía es consistente con y, de alguna manera, un reestablecimiento de la primera. También podemos ver la necesidad de un acercamiento más formal para evaluar si el diseño cumple con estas guías. Es importante notar que estas guías pueden *tener que violarse* para *mejorar el desempeño* de ciertos queries. Por ejemplo, si un query importante recupera información concerniente al departamento de un empleado, junto con los atributos del empleado, el esquema EMP_DEP puede usarse como una relación base. Sin embrago, las anomalías en EMP_DEP deben anotarse y entenderse, de tal forma que, cada que se actualice a la relación base, no terminemos con inconsistencias. En general, es aconsejable usar relaciones libres de anomalías y especificar vistas que incluyan JOINs para colocar juntos atributos que son referenciados con frecuencia en queries importantes. Esto reduce el número de términos JOIN especificados en el query, facilitando la escritura del query, y en muchos casos mejorando el desempeño.

9.1.3 Valores Nulos en las Tuplas

En algunos diseños de esquemas podemos agrupar varios atributos juntos en una sola relación "gruesa". Si muchos de los atributos no aplican a todas las tuplas en la relación, finalizaremos con muchos nulos en esas tuplas. Esto puede desperdiciar espacio a nivel de almacenamiento y también puede conducir a problemas con el entendimiento del significado de los atributos y la especificación de operaciones JOIN a nivel lógico. Otro problema con los nulos es cómo contabilizarlos cuando se apliquen operaciones agregadas tales como COUNT o SUM. Sin embargo, los nulos pueden tener diferentes interpretaciones, tales como:

- El atributo *no aplica* a esta tupla.
- El valor del atributo para esta tupla es *desconocido*.
- El valor es *conocido pero ausente*; esto es, no ha sido registrado aún.

Tener la misma representación para todos los nulos compromete los diferentes significados que puedan tener. Por lo tanto, podemos establecer otra guía como sigue:

GUIA 3: Tanto como sea posible, evitar la colocación de atributos en una relación base cuyos valores puedan ser nulos. Si los nulos son inevitables, asegurarse de que apliquen únicamente en casos excepcionales y no a una mayoría de tuplas en la relación.

Por ejemplo, si sólo el 10% de los empleados tienen oficinas independientes, hay poca justificación para incluir un atributo NUM_OFIC en la relación EMPLEADO; por el contrario, puede crearse una relación EMP_OFIC(IMSS, NUM_OFIC) para incluir tuplas de empleados con oficinas individuales.

9.1.4 Tuplas Falsas

Consideremos los dos esquemas de relación EMP_LOCS y EMP_PROY1 de la Figura 9.5(a), las cuales pueden usarse en lugar de la relación EMP_PROY de la Figura 9.3(b). Una tupla en EMP_LOCS significa que todos los empleados cuyo nombre es ENOMBRE trabajan en *algunos proyectos* cuya localidad es PLOCALIDAD. Una tupla en EMP_PROY1 significa que todos los empleados cuyo número de seguro social es IMSS trabajan HORAS a la semana en el proyecto cuyo nombre, número, y localidad son PNOMBRE, PNUMERO y PLOC. La Figura 9.5(b) muestra extensiones de relación de EMP_LOCS y EMP_PROY1 correspondientes a la relación EMP_PROY de la Figura 9.4, las cuales se obtienen aplicando las operaciones PROJECT (π) apropiadas a EMP_PROY.

Supongamos que empleamos EMP_PROY1 y EMP_LOCS como las relaciones de la base de datos en lugar de EMP_PROY. Esto produce un esquema de diseño particularmente malo, debido a que no podemos recuperar la información que estaba originalmente en EMP_PROY a partir de EMP_PROY1 y EMP_LOCS. Si intentamos una operación JOIN NATURAL sobre EMP_PROY1 y EMP_LOCS, obtendremos mucho más tuplas de las que tenía EMP_PROY. En la Figura 9.6 se muestra el resultado de aplicar el join únicamente a las tuplas *sobre* las líneas punteadas de la Figura 9.5(b) para reducir el tamaño de la relación resultante. A las tuplas adicionales que no se encontraban en EMP_PROY se les llaman **tuplas falsas** debido a que representan información falsa o *incorrecta* que no es válida. En la Figura 9.6 las tuplas falsas están marcadas con asteriscos (*).

Descomponer EMP_PROY en EMP_LOCS y EMP_PROY1 es malo porque cuando los unamos de regreso en un JOIN NATURAL, no obtendremos la información original correctamente. Esto se debe a que PLOC es el atributo que relaciona a EMP_LOCS y EMP_PROY1. Ahora podemos establecer informalmente otra guía de diseño:

GUIA 4: Diseñar esquemas de relación de tal forma que puedan unirse con igualdad de condiciones sobre atributos que son ya sea llaves primarias o llaves externas de manera que garanticen que no se generarán tuplas falsas.

(a) EMP_LOCS

ENOMBRE	PLOC

EMP_PROY1

IMSS	PNUMERO	HORAS	PNOMBRE	PLOC

p. k.

(b)

EMP_LOCS

ENOMBRE	PLOC
Smith, John B	Bellaire
Smith, John B	Sugarland
Narayan, Ramesh K	Houston
English, Joyce A	Bellaire
Wong, Franklin T	Sugarland
Wong, Franklin T	Houston
Wong, Franklin T	Stafford
Zelaya, Alicia J	Stafford
Jabbar, Ahmad V	Stafford
Wallace, Jeunifer S	Stafford
Wallace, Jennifer S.	Houston
Borg, James E	Houston

EMP_PROY1

IMSS	PNUMERO	HORAS	PNOMBRE	PLOC
123456789	1	32.5	ProductoX	Bellaire
123456789	2	7.5	ProductoY	Sugarland
666884444	3	40.0	ProductoZ	Houston
999887777	10	10.0	Computación	Stafford
987987987	10	35.0	Computación	Stafford
987654321	20	15.0	Reorganización	Houston
888665555	20	null	Reorganización	Houston

Figura 9.5 Representación alterna de EMP_PROY. (a) Representación de EMP_PROY de la Figura 9.3(b) por dos esquemas de relación: EMP_LOCS y EMP_PROY1. (b) Resultado de proyectar la relación EMP_PROY de la Figura 9.4 en los atributos de EMP_PROY1 y EMP_LOCS.

IMSS	PNUMERO	HORAS	PNOMBRE	PLOC	ENOMBRE
123456789	1	32.5	ProductoX	Bellaire	Smith, John B
*123456789	1	32.5	ProductoX	Bellaire	English, Joyce
123456789	2	7.5	ProductoY	Sugarland	Smith, John B
666884444	3	40.0	ProductoZ	Houston	Narayan, Ram
*666884444	3	40.0	ProductoZ	Houston	Wong, Frank
*333445555	20	10.0	Reorganizac	Houston	Narayan, Ram
*333445555	20	10.0	Reorganizac	Houston	Wong, Frank

Figura 9.6 Resultado de aplicar la operación JOIN NATURAL a TEMP_PROY1 y EMP_LOCS, con las tuplas falsas marcadas con *.

9.1.5 Discusión

En las Secciones anteriores, discutimos informalmente situaciones que llevaron a esquemas de relación problemáticos, y propusimos guías informales para un buen diseño relacional. En el resto de este capítulo presentaremos los conceptos formales y la teoría que puede usarse para definir con mayor precisión los conceptos de la "bondad" y "desventajas" de esquemas de relación *individuales*. Especificaremos varias formas de normalización para los esquemas de relación. Las formas de normalización definidas en este capítulo se basan en el concepto de dependencia funcional, la cual describiremos a continuación.

9.2 Dependencias Funcionales

El concepto más importante en el diseño de esquemas relacionales es el de dependencia funcional. En esta sección definiremos formalmente este concepto, y posteriormente veremos cómo conduce a esquemas de relación en formas normalizadas.

9.2.1 Definición de Dependencia Funcional

Una dependencia funcional es un argumento entre dos conjuntos de atributos de la base de datos. Supongamos que nuestro esquema relacional de la base de datos tiene n atributos A_1, A_2, ..., A_n; imaginemos a la base de datos completa descrita por un sólo esquema de relación **universal** $R=\{A_1, A_2, ..., A_n\}$. Esto no implica que almacenaremos a la base de datos como una sola tabla universal; usaremos este concepto sólo en el desarrollo de la teoría formal de las dependencias de datos.

Una **dependencia funcional**, denotada por $X \rightarrow Y$, entre dos conjuntos de atributos X y Y que son subconjuntos de R especifica un argumento sobre las posibles tuplas que pueden formar una instancia de relación r de R. El argumento establece que, para cualquier par de tuplas t1 y t2 en r tal que $t1[X] = t2[X]$, debemos tener que $t1[Y] = t2[Y]$. Esto significa que los valores del componente Y de una tupla en r dependen de, o están determinadas por, los valores del componente X; o **alternativamente**, los valores del componente X de una tupla únicamente (o **funcionalmente**) **determinan** los valores del componente Y. También decimos que existe una dependencia funcional de X a Y o que Y es **funcionalmente dependiente** de X. Abreviaremos la dependencia funcional como **FD**. Al conjunto de atributos de X se le llama el **lado izquierdo** de la FD, y a Y se le llama el **lado derecho**.

Así, X determina funcionalmente a Y en un esquema de relación R si y sólo si, cada par de tuplas de r(R) coinciden con el valor de X, necesariamente deben coincidir con el valor de Y. Notemos que:

- Si un argumento en R establece que no puede existir más de una tupla con un valor de X dado en cualquier instancia de relación r(R) -esto es, X es un **candidato a llave** de R- esto implica que $X \rightarrow Y$ para cualquier subconjunto de atributos Y de R.

- Si X $\rightarrow$ Y en R, esto no indica si o no Y $\rightarrow$ X en R.

Una dependencia funcional es una propiedad del significado o **semántica** de los atributos. Usemos nuestro conocimiento de la semántica de los atributos de R -esto es, cómo se relacionan unos con otros- para especificar las dependencias funcionales que deben retenerse sobre *todos* los estados de la relación (extensiones) r de R. Cada vez que la semántica de dos conjuntos de atributos en R indica que debe retenerse una dependencia funcional, especificamos a la dependencia como un argumento. A las extensiones de relación r(R) que satisfacen los argumentos de dependencia funcional se les llaman **extensiones legales** (o **estados legales de la relación**) de R, debido a que obedecen los argumentos de dependencia funcional. Por lo tanto, el uso principal de las dependencias funcionales es para la descripción de un esquema de relación R especificando argumentos sobre sus atributos que deben retenerse *en cualquier momento*.

Consideremos el esquema de relación EMP_PROY de la Figura 9.3(b); de la semántica de los atributos, sabemos que deben retenerse las siguientes dependencias funcionales:

(a) IMSS $\rightarrow$ ENOMBRE

(b) PNUMERO $\rightarrow$ {PNOMBRE, PLOC}

(c) {IMSS, PNUMERO} $\rightarrow$ HORAS

Estas dependencias funcionales especifican que (a) el valor del número de seguro social de un empleado (IMSS) determina de manera única el nombre del empleado (ENOMBRE), (b) el valor de un número de proyecto (PNUMERO) determina únicamente el nombre del proyecto (PNOMBRE) y su localidad (PLOC), y (c) una combinación de IMSS y PNUMERO determinan únicamente el número de horas que el empleado trabaja en el proyecto a la semana (HORAS). Alternativamente, decimos que ENOMBRE está funcionalmente determinado (o funcionalmente dependiente de) por IMSS, o "dado un valor de IMSS, conoceremos el valor de ENOMBRE", etc.

La Figura 9.3 también introduce una notación diagramal para desplegar dependencias funcionales. Cada FD se despliega como una línea horizontal. Los atributos del lado izquierdo de la FD se conectan por líneas verticales a la línea horizontal que representa la FD. El lado derecho de los atributos de la FD se conecta a la línea horizontal mediante flechas que apuntan a los atributos, como se mostró en las Figuras 9.3(a) y (b).

Una dependencia funcional es una *propiedad del esquema de relación* (intensión) R, y no de un estado legal de la relación en particular (extensión) r de R. Por lo tanto, una FD *no puede* ser inferida automáticamente desde una extensión de relación r pero debe definirse explícitamente por alguien que conozca la semántica de los atributos de R. Por ejemplo, la Figura 9.7 muestra una instancia en particular de la relación del esquema de relación ENSEÑA. A pesar de que a primera instancia podamos estar tentados a decir que TEXTO $\rightarrow$ CURSO, no podemos confirmarlo a menos que sepamos que es verdad para todos los

posibles estados del esquema de relación ENSEÑA. Sin embrago, es suficiente demostrar un sólo contraejemplo para desaprobar una dependencia funcional. Por ejemplo, debido a que "Smith" enseña ambos cursos "Estructuras de Datos" y "Administración de Datos", podemos concluir que PROFESOR no determina funcionalmente el CURSO. Esto lo notamos como PROFESOR *→ CURSO. De la Figura 9.7 podemos decir que CURSO * → TEXTO.

ENSEÑA

PROFESOR	CURSO	TEXTO
Smith	Estructuras de Datos	Bartram
Smith	Administración de Datos	Al-Nour
Hall	Compiladores	Hoffman
Brown	Estructuras de Datos	Augenthaler

Figura 9.7 Relación ENSEÑA.

9.2.2 Reglas de Inferencia para Dependencias Funcionales

Denotamos por F al conjunto de dependencias funcionales que son específicas al esquema de relación R. Típicamente, el diseñador del esquema especifica las dependencias funcionales que son *semánticamente obvias*; usualmente, sin embargo, otras dependencias funcionales numerosas retienen *todas* las instancias de relación legales que satisfacen a las dependencias en F. Al conjunto de tales dependencias funcionales se le llama la **conclusión** de F y se denota como F+. Por ejemplo, supongamos que especificamos el siguiente conjunto F de dependencias funcionales obvias en el esquema de relación de la Figura 9.3(a):

F = {IMSS → {ENOMBRE, FECHANAC, DIRECCION, DNUMERO},
DNUMERO → {DNOMBRE, DGTEIMSS} }

Podemos **inferir** las siguientes dependencias funcionales de F:

IMSS → {DNOMBRE, DGTEIMSS}
IMSS → IMSS
DNUMERO → DNOMBRE

Una FD X → Y es **inferida de** un conjunto de dependencias F **especificadas sobre** R si X → Y retiene en cada estado de relación r que es una **extensión legal de** R; esto es, cada que r satisface todas las dependencias en F, X → Y también se retiene en r. La conclusión F+ de F es el conjunto de todas las dependencias funcionales que pueden inferirse de F. Para determinar una forma semántica de inferir dependencias, debemos descubrir un conjunto de **reglas de inferencia** que pueden usarse para inferir nuevas dependencias a partir de un conjunto de dependencias dado. La notación F ⊨ X → Y denota que la dependencia funcional X → Y es inferida de un conjunto de dependencias funcionales F.

En la siguiente discusión, usaremos una notación abreviada cuando discutamos las dependencias funcionales. Por conveniencia concatenaremos variables de atributos y ahorraremos las comas. Por lo tanto, la FD{X,Y} → Z se abrevia como XY → Z, y la FD{X,Y,Z} → {U,V} se abrevia como XYZ → UV. Las siguientes reglas de inferencia son bien conocidas para dependencias funcionales:

(IR1) (Regla reflexiva) Si $X \supseteq Y$, entonces $X \rightarrow Y$
(IR2) (Regla de aumento) $\{X \rightarrow Y\} \models XZ \rightarrow YZ$
(IR3) (Regla transitiva) $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$
(IR4) Regla de descomposición (o proyectiva)) $\{X \rightarrow YZ\} \models X \rightarrow Y$
(IR5) (Regla de unión (o aditiva)) $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$
(IR6) (Regla pseudotransitiva) $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$

La regla reflexiva (IR1) establece que un conjunto de atributos siempre se determina a si misma, lo cual es obvio. La regla de aumento (IR2) establece que agregar el mismo conjunto de atributos a ambos lados (derecha e izquierda) de una dependencia resulta en otra dependencia válida. De acuerdo a IR3, las dependencias funcionales son transitivas. La regla de descomposición (IR4), establece que podemos eliminar atributos del lado derecho de una dependencia; aplicando esta regla repetidamente podemos descomponer la FD X → {A1, A2, ..., An} en el conjunto de dependencias {X → A1, X → A2, ..., X → An}. La regla de unión (IR5) nos permite hacer lo opuesto; podemos combinar un conjunto de dependencias {X → A1, X → A2, ..., X → An} en una sola FD X → {A1, A2, ..., An}.

9.3 Formas de Normalización Basadas en Llaves Primarias

En esta sección discutiremos el proceso de normalización y definiremos las tres primeras formas de normalización de esquemas relacionales. Las definiciones de la segunda y tercera forma de normalización presentadas aquí se basan en las dependencias funcionales y llaves primarias de un esquema de relación. Discutiremos cómo fueron desarrolladas estas formas normalizadas y la intuición detrás de ellas. Se presentarán definiciones generales de estas formas normalizadas, las cuales toman en cuanta a todos los *candidatos a llave* de una relación en lugar de sólo a la *llave primaria*.

9.3.1 Introducción a la Normalización

El proceso de normalización, como fue propuesto por Codd (1972), lleva a un esquema de relación a través de una serie de pruebas para "certificar" si o no pertenecen a cierta **forma normalizada**. Inicialmente, Codd propuso tres formas de normalización, a las cuales llamaremos primera, segunda y tercera formas de normalización. Posteriormente fue propuesta una definición más fuerte de 3NF por Boyce y Codd y se le conoce como la forma de normalización Boyce-Codd. Todas estas formas normalizadas se basan en las dependencias funcionales entre los atributos de una relación. Posteriormente, fueron

propuestas una cuarta y quinta formas de normalización (4NF) y (5NF), basados en los conceptos de dependencias multivaluadas y dependencias join, respectivamente.

La **normalización de datos** puede observarse como un proceso durante el cual se descomponen esquemas de relaciones no satisfactorios rompiendo sus atributos en esquemas de relación más pequeños que poseen propiedades deseables. Un objetivo del proceso original de normalización es asegurar que no ocurran las anomalías de actualización. Las formas de normalización proporcionan a los diseñadores de las bases de datos:

- Un marco formal para analizar esquemas de relación basados en sus llaves y en las dependencias funcionales entre sus atributos.

- Una serie de pruebas que pueden llevarse a cabo sobre esquemas de relación individuales de tal forma que la base de datos relacional pueda **normalizarse** a cualquier grado. Cuando falla alguna prueba, la relación que viola la prueba debe descomponerse en relaciones que cumplan individualmente las pruebas de normalización.

Las formas de normalización, cuando se consideran *aisladas* de otros factores, no garantizan un buen diseño de la base de datos. Generalmente no es suficiente checar independientemente que cada relación en la base de datos está, digamos, en BCNF o 3NF. Por el contrario, el proceso de normalización a través de la descomposición debe confirmar la existencia de propiedades adicionales a los esquemas relacionales, tomados en conjunto. Dos de estas propiedades son:

- La pérdida de la propiedad join o no aditiva, lo cual garantiza que el problema de la tupla falsa no ocurra.

- La propiedad de preservación de dependencia, lo cual asegura que todas las dependencias funcionales se presenten en algunas de las relaciones resultantes individuales.

Aplazaremos la presentación del concepto y técnicas formales que garanticen las dos propiedades anteriores. En esta sección nos concentraremos en una *discusión intuitiva* del proceso de normalización. Notemos que las formas de normalización mencionadas en esta sección no son las únicas posibles. Basados en tipos de argumentos adicionales pueden definirse formas de normalización adicionales para cumplir otros criterios. Las formas de normalización hasta BCNF se definen considerando únicamente la dependencia funcional y los argumentos de llave, mientras que la 4NF considera un argumento adicional llamado dependencia join. La utilidad práctica de formas de normalización se vuelve cuestionable cuando los argumentos sobre los cuales se basan son difíciles de entender o detectar por los diseñadores y usuarios de la base de datos quienes deben descubrir estos argumentos.

Otro punto que vale la pena hacer notar es que los diseñadores de la base de datos *no necesitan* normalizar a la forma de normalización más alta posible. Por razones de desempeño las relaciones pueden dejarse en formas de normalización inferiores.

Antes de continuar, recordemos las definiciones de llave de un esquema de relación. Una **superllave** de un esquema de relación R = {A1, A2, ..., An} es un conjunto de atributos S $\subseteq$ R con la propiedad de que ningun par de tuplas t1 y t2 en ningun estado legal de la relación r de R tendrán t1[S] = t2[S]. Una **llave** K es una superllave con la propiedad adicional que la eliminación de cualquier atributo de K ocasionara que K no sea más una superllave. La diferencia entre una llave y una superllave es que la llave tiene que ser "mínima"; esto es, si tenemos una llave K = {A1, A2, ... Ak}, entonces K - Ai no es una llave para $1 \leq i \leq k$. En la Figura 9.1 {IMSS} es una llave de EMPLEADO, mientras que {IMSS, ENOMBRE}, {IMSS, ENOMBRE, FECHANAC}, etc. son todas superllaves.

Si un esquema de relación tiene más de una llave "mínima", a cada una se le llama **candidato de llave**. Uno de los candidatos a llave es designado *arbitrariamente* para ser la **llave primaria**, y a las demás se les llaman llaves secundarias. Cada esquema de relación debe tener una llave primaria. En la Figura 9.1 {IMSS} es el único candidato a llave para EMPLEADO, así que también es una llave primaria.

A un atributo del esquema de relación R se le llama **atributo primo** de R si es un miembro de *cualquier llave* de R. A un atributo se le llama **no primo** si no es un atributo primo -esto es, si no es miembro de cualquier candidato a llave. En la Figura 9.1 IMSS y PNUMERO son atributos primos de TRABAJA_EN, mientras que los demás atributos de TRABAJA_EN son no primos.

Ahora presentaremos las tres primeras formas de normalización: 1NF, 2NF y 3NF. Estas fueron propuestas por Codd (1972) como consecuencia de adquirir el estado deseable de relaciones 3NF progresando a través de los estados intermedios de 1NF y 2NF si es necesario.

9.3.2 Primer Forma de Normalización (1NF)

La **primer forma de normalización** se considera parte de la definición formal de una relación; históricamente, fue definida para deshabilitar los atributos multivaluados, atributos compuestos, y sus combinaciones. Establece que los dominios de los atributos deben incluir sólo *valores atómicos* (sencillos, indivisibles) y que el valor de cualquier atributo en una tupla debe ser un *valor sencillo* del dominio del atributo. Por lo tanto, 1NF deshabilita el tener conjuntos de valores, una tupla de valores, o una combinación de ambos como un valor de atributo para *una sola tupla*. En otras palabras, 1NF deshabilita las "relaciones dentro de las relaciones" o las "relaciones como atributos de tuplas". Los únicos valores de atributos permitidos por 1NF son los **valores atómicos (o indivisibles)**.

Consideremos el esquema de la relación DEPARTAMENTO mostrado en la Figura 9.1, cuya llave primaria es DNUMERO, y supongamos que la extendemos incluyendo al

atributo DLOC mostrado con líneas punteadas. Supongamos que cada departamento puede tener cierto *número* de localidades. En la Figura 9.8 se muestra el esquema DEPARTAMENTO y un ejemplo de extensión. Como podemos ver, este no está en 1NF porque DLOC no es un atributo atomico, como se ilustró por la primer tupla de la Figura 9.8(b). Existen dos formas de poder ver al atributo DLOC:

- El dominio de DLOC contiene valores atómicos, pero algunas tuplas pueden tener conjuntos de estos valores. En este caso, IMSS *→ DLOC.
- El dominio de DLOC contiene conjuntos de valores y por lo tanto no es atómico. En este caso, IMSS → DLOC, porque cada conjunto se considera un miembro sencillo del dominio del atributo.

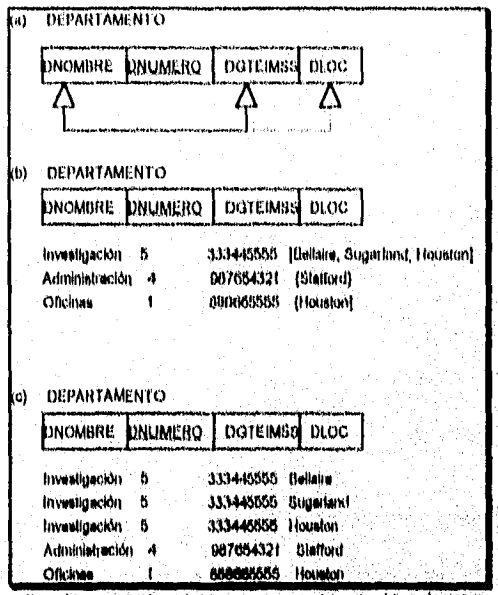


Figura 9.8 Normalización a 1NF. (a) Un esquema de relación sin 1NF. (b) Ejemplo de instancia de relación. (c) Relación 1NF con redundancia.

En cualquier caso, la relación DEPARTAMENTO de la Figura 9.8 no es 1NF; en efecto, no califica aún como relación, de acuerdo a nuestra definición. Para normalizarla a relaciones 1NF, debemos romper sus atributos en las dos relaciones DEPARTAMENTO y DEP_LOC mostradas en la Figura 9.2. La idea es eliminar al atributo DLOC que viola 1NF y colocarlo en una relación independiente DEP_LOCS junto con la llave primaria DNUMERO de DEPARTAMENTO. La llave primaria de esta relación es la combinación {DNUMERO, DLOC}, como se mostró en la Figura 9.2. Para *cada localidad de departamento* existe una tupla distinta en DEP_LOC. El atributo DLOC es eliminado de la

atributo DLOC mostrado con líneas punteadas. Supongamos que cada departamento puede tener cierto *número* de localidades. En la Figura 9.8 se muestra el esquema DEPARTAMENTO y un ejemplo de extensión. Como podemos ver, este no está en 1NF porque DLOC no es un atributo atómico, como se ilustró por la primer tupla de la Figura 9.8(b). Existen dos formas de poder ver al atributo DLOC:

- El dominio de DLOC contiene valores atómicos, pero algunas tuplas pueden tener conjuntos de estos valores. En este caso, IMSS *→ DLOC.
- El dominio de DLOC contiene conjuntos de valores y por lo tanto no es atómico. En este caso, IMSS → DLOC, porque cada conjunto se considera un miembro sencillo del dominio del atributo.

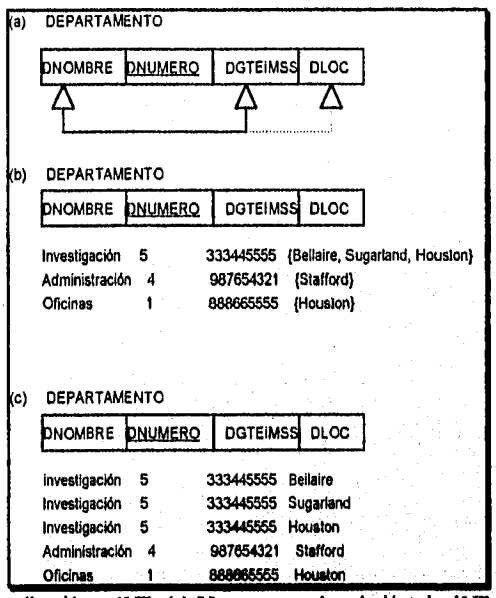


Figura 9.8 Normalización a 1NF. (a) Un esquema de relación sin 1NF. (b) Ejemplo de instancia de relación. (c) Relación 1NF con redundancia.

En cualquier caso, la relación DEPARTAMENTO de la Figura 9.8 no es 1NF; en efecto, no califica aún como relación, de acuerdo a nuestra definición. Para normalizarla a relaciones 1NF, debemos romper sus atributos en las dos relaciones DEPARTAMENTO y DEP_LOC mostradas en la Figura 9.2. La idea es eliminar al atributo DLOC que viola 1NF y colocarlo en una relación independiente DEP_LOCS junto con la llave primaria DNUMERO de DEPARTAMENTO. La llave primaria de esta relación es la combinación {DNUMERO, DLOC}, como se mostró en la Figura 9.2. Para *cada localidad de departamento* existe una tupla distinta en DEP_LOC. El atributo DLOC es eliminado de la

relación DEPARTAMENTO de la Figura 9.8, descomponiendo la relación no 1NF en las dos relaciones 1NF DEPARTAMENTO y DLOC de la Figura 9.2.

Notemos que una segunda forma de normalizar en 1NF es tener una tupla en la relación original DEPARTAMENTO para cada localidad de DEPARTAMENTO, como se mostró en la Figura 9.8(c). En este caso, la llave primaria se convierte en la combinación {DNUMERO, DLOC}, y existe redundancia en las tuplas. La primer solución es superior porque no sufre de este problema de redundancia. En efecto, si elegimos la segunda solución, será descompuesta posteriormente durante subsecuentes pasos de normalización a la primer solución.

La primer forma de normalización también deshabilita atributos compuestos que son por sí mismo multivaluados. A estas se les llaman **relaciones anidadas** porque cada tupla puede tener a una relación *dentro de ella*. La Figura 9.9 muestra cómo puede desplegarse la relación EMP_PROY si se permite el anidamiento. Cada tupla representa a una entidad empleado, y una relación PROYS(PNUMERO, HORAS) *dentro de cada tupla* representa los proyectos de los empleados y las horas a la semana que trabaja cada empleado en cada proyecto. El esquema de la relación EMP_PROY puede representarse como sigue:

EMP_PROY(IMSS, ENOMBRE, {PROYS(PNUMERO, HORAS)})

El juego de brazos {} identifica a los atributos de PROYS como multivaluados, y listamos a los atributos componenetes que forman a PROYS entre paréntesis (). Interesantemente, recientes investigaciones en el modelo relacional intentan permitir y formalizar relaciones anidadas, las cuales fueron deshabilitadas anteriormente en 1NF.

Notemos que IMSS es la llave primaria de la relación EMP_PROY de las Figuras 9.9(a) y (b), mientras que PNUMERO es la llave primaria **parcial** de cada relación anidada; esto es, dentro de cada tupla, la relación anidada debe tener valores únicos de PNUMERO. Para normalizar esto a 1NF, eliminamos los atributos de la relación anidada en una nueva relación y **propagamos la llave primaria** a ella; la llave primaria de la nueva relación combinará la llave parcial con la llave primaria de la relación original. La descomposición y propagación de la llave primaria lleva a los esquemas mostrados en la Figura 9.9(c).

Este procedimiento puede aplicarse recursivamente a una relación con multiple nivel de anidamiento para **desligar** la relación a un conjunto de relaciones 1NF. Esto es útil en la conversión de esquemas jerárquicos a relaciones 1NF. Las relaciones restringidas a 1NF conducen a los problemas asociados con dependencias multivaluadas 4NF.

9.3.3 Segunda Forma de Normalización (2NF)

La segunda forma de normalización se basa en el concepto de dependencia funcional completa. Una dependencia funcional $X \rightarrow Y$ es una **dependencia funcional completa** si por la eliminación de cualquier atributo A de X significa que la dependencia no es más retenida; esto es, para cualquier atributo $A \in X$, $(X - \{A\}) \not\rightarrow Y$. Una dependencia

funcional X → Y es una **dependencia parcial** si algun atributo A ∈ X puede eliminarse de X y la dependencia continuar; esto es, para algun A ∈ X, (X - {A}) → Y. En la Figura 9.3(b), {IMSS, PNUMERO} → HORAS es una dependencia completa. Sin embargo, la dependencia {IMSS, PNUMERO} → ENOMBRE es parcial porque se retiene IMSS → ENOMBRE.

Un esquema de relación está en 2NF si cada atributo no primo A en R es *completa y funcionalmente dependiente* de la llave primaria de R. La relación EMP_PROY de la Figura 9.3(b) está en 1NF pero no en 2NF. El atributo no primo ENOMBRE viola la 2NF debido a fd2, como lo hacen los atributos no primos PNOMBRE y PLOC debido a fd3. Las dependencias funcionales fd1, fd2 y fd3 de la Figura 9.3(b) por lo tanto conducen a la descomposición de EMP_PROY en los tres esquemas de relación EP1, EP2 y EP3 mostrados en la Figura 9.10(a), cada uno de los cuales está en 2NF. Podemos ver que las relaciones EP1, EP2 y EP3 están desprovistos de las anomalías de actualización de las cuales sufre la Figura 9.3(b).

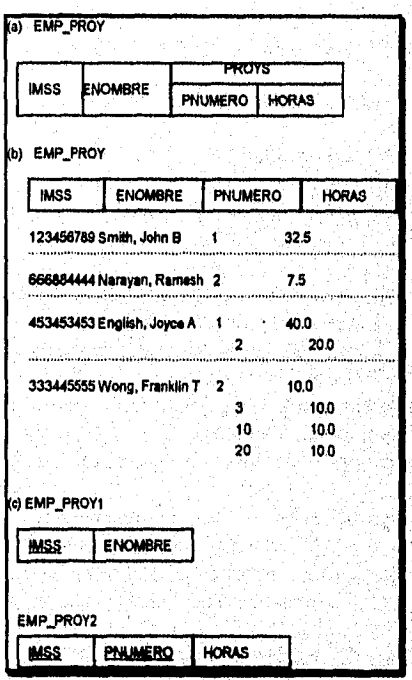


Figura 9.9 Normalización de relaciones anidadas a 1NF. (a) Esquema de la relación EMP_PROY con una "relación anidada" **PROYS** dentro de **EMP_PROY**. (b) Ejemplo extensión de la relación **EMP_PROY** con relaciones anidades dentro de cada tupla. (c) Descomposción de **EMP_PROY** a relaciones 1NF mediante la migración de la llave primaria.

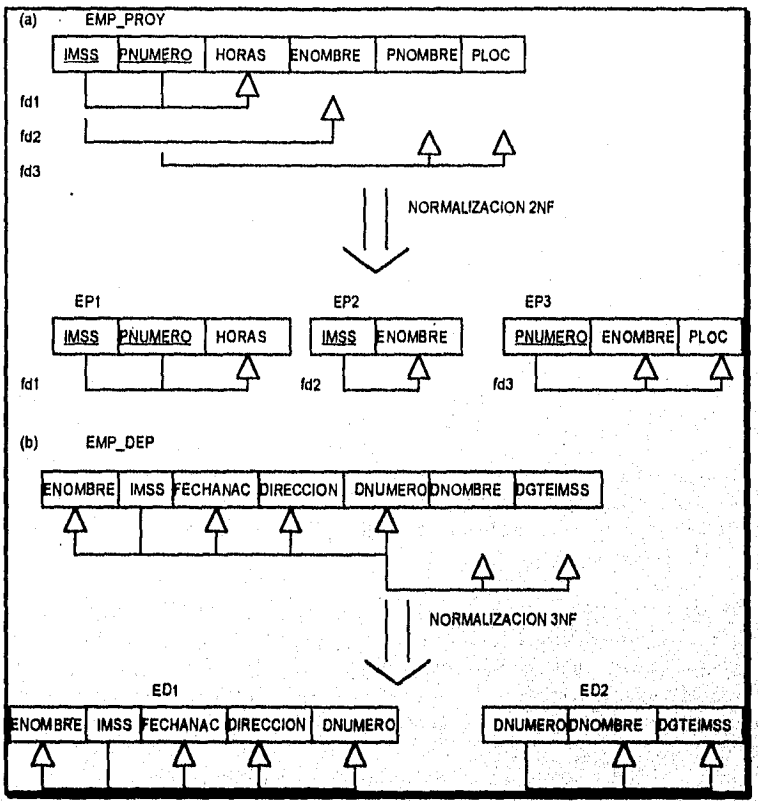


Figura 9.10 Proceso de normalización. (a) Normalización de EMP_PROY a relaciones 2NF. (b) Normalización de EMP_DEP a relaciones 3NF.

9.3.4 Tercera Forma de Normalización (3NF)

La tercera forma de normalización se basa en el concepto de dependencia transitiva. Una dependencia funcional $X \rightarrow Y$ en un esquema de relación R es una **dependencia transitiva** si existe un conjunto de atributos Z que *no son subconjunto* de ninguna llave de R, y se retienen ambas $X \rightarrow Y$ y $Z \rightarrow Y$. La dependencia IMSS $\rightarrow$ DGTEIMSS es transitiva a través de DNUMERO en EMP_DEP de la Figura 9.3(a) porque ambas dependencias IMSS $\rightarrow$ DNUMERO y DNUMERO $\rightarrow$ DGTEIMSS se retienen y DNUMERO no es subconjunto de la llave de EMP_DEP. Intuitivamente, podemos ver que la dependencia de DGTEIMSS sobre DNUMERO es indeseable en EMP_DEP ya que DNUMERO no es una llave de EMP_DEP.

De acuerdo a la definición original de Codd, un esquema de relaciones R está en 3NF si está en 2NF y ningun atributo no primo de R está transitoriamente dependiente en la llave primaria. El esquema de la relación EMP_DEP de la Figura 9.3(a) está en 2NF, ya que no existe ninguna dependencia parcial sobre alguna llave. Sin embargo, EMP_DEP no está en 3NF debido a la dependencia transitoria de DGTEIMSS (y también de DNOMBRE) sobre IMSS vía DNUMERO. Podemos normalizar a EMP_DEP descomponiéndolo en los dos esquemas de relaciones 3NF ED1 y ED2 mostrados en la Figura 9.10(b). Intuitivamente, veremos que ED1 y ED2 representan hechos de entidades independientes acerca de empleados y departamentos. Una operación JOIN NATURAL sobre ED1 y ED2 recuperará a la relación original EMP_DEP sin generar tuplas falsas.

9.4 Definiciones Generales de Segunda y Tercer Formas de Normalización

En general, deseamos diseñar nuestros esquemas de relación de tal forma que no tengan ninguna dependencia parcial o transitoria, debido a que estos tipos de dependencias ocasionan las anomalías de actualización ya discutidas. De acuerdo a las definiciones de formas de normalización de segundo y tercer nivel, los pasos para normalización a relaciones 3NF deshabilitan a las dependencias parciales y transitivas de la *llave primaria*. Estas definiciones, sin embargo, no toman en cuenta, si es que existen, a otros candidatos a llave de la relación.

En esta sección daremos las definiciones más generales de 2NF y 3NF que toman en cuenta a *todos* los candidatos a llave. Notemos que esto no afecta la definición de 1NF, ya que es independiente de llaves y dependencias funcionales. Utilizaremos las *definiciones generales* de atributos primos, dependencias parciales y funcionales, y dependencias transitivas dadas anteriormente que consideran a todos los candidatos a llave de una relación.

9.4.1 Definición General de la Segunda Forma de Normalización (2NF)

Un esquema de relación R está en **segunda forma de normalización (2NF)** si cada atributo no primo A en R no es parcialmente dependiente de *ninguna llave* de R. Consideremos el esquema de relación **LOTES** mostrado en la **Figura 9.11(a)**, el cual describe parcelas de tierra para su venta en varios condados de un estado. Supongamos que existen dos candidatos a llave: **ID_PROP** y **{NOM_CONDADO, #LOTE}**; esto es, los **#LOTE** son únicos dentro de cada condado pero los **ID_PROP** son únicos a través de los condados de todo el estado.

Basado en los dos candidatos a llave, sabemos que se retienen las dependencias funcionales **fd1** y **fd2** de la **Figura 9.11(a)**. Eligiremos a **ID_PROP** como la llave primaria, de tal forma que aparece subrayada en la **Figura 9.11(a)**. Supongamos que se retienen las siguientes dos dependencias funcionales en **LOTES**:

fd3: NOM_CONDADO → PCTJE_IMP
fd4: AREA → PRECIO

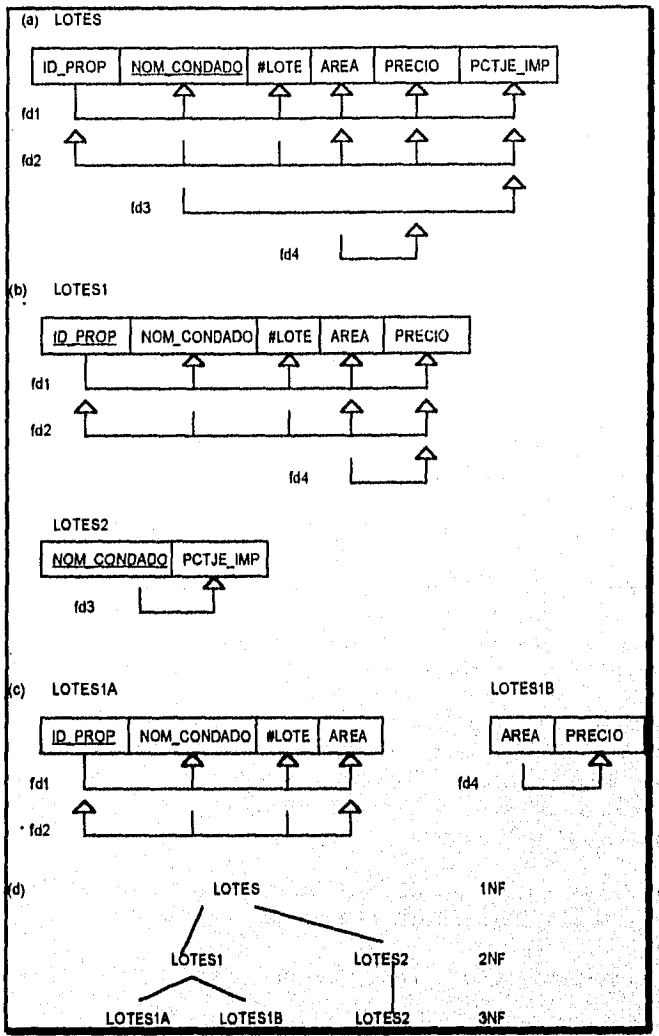


Figura 9.11 Normalización a 2NF y 3NF. (a) El esquema de relación LOTES y sus dependencias funcionales fd1 a fd4. (b) Descomposición de LOTES en LOTES1 y LOTES2. (c) Descomposición de LOTES1 en relaciones 3NF LOTES1A y LOTES1B. (d) Resúmen de la normalización de LOTES.

En palabras, la dependencia fd3 establece que el porcentaje de impuesto es fijo para un condado dado (no varía de lote a lote dentro del mismo condado), mientras que fd4

establece que el precio de un lote se determina por su área sin importar el condado en que se encuentre (supongamos que este es el precio del lote para propósitosde impuestos).

El esquema de la relación LOTES viola la definición general de 2NF porque PCTJE_IMP es parcialmente dependiente del candidato a llave {NOM_CONDADO, #LOTE}, debido a fd3. Para normalizar a LOTES a 2NF, lo descompondremos en las dos relaciones LOTES1 y LOTES2, mostradas en la Figura 9.11(b). Construiremos LOTES1 eliminando al atributo PCTJE_IMP que viola 2NF de LOTES y colocándolo con NOM_CONDADO (el lado izquierdo de fd3 que ocasiona la dependencia parcial) en otra relación LOTES2. LOTES1 y LOTES2 están en 2NF. Notemos que fd4 no viola la 2NF y es llevada a LOTES1.

9.4.2 Definición General de la Tercera Forma de Normalización (3NF)

Un esquema de relación R está en 3NF, cada vez que una dependencia funcional $X \rightarrow A$ se retiene en R, ya sea que (a) X es superllave de R, o (b) A es un atributo primo de R. De acuerdo a esta definición, LOTES2 está en 3NF (Figura 9.11(b)). Sin embrago, fd4 en LOTES1 viola la 3NF porque AREA no es una superllave de LOTES1 y PRECIO no es un atributo primo. Para normalizarla a 3NF, descompondremos a LOTES1 en los esquemas de relación LOTES1A y LOTES1B mostrados en la Figura 9.11(c). Construiremos LOTES1A eliminando al atributo PRECIO de LOTES1 que viola la 3NF y colocándolo con AREA (el lado izquierdo de fd4 que ocasiona la dependencia transitiva) a otra relación LOTES1B. Ambas relaciones LOTES1A y LOTES1B están en 3NF. Vale la pena notar dos puntos acerca de la definición general de 3NF:

- Esta definición puede aplicarse *directamente* para probar que si un esquema de relación está en 3NF; no necesita pasar por 2NF en primera instancia. Si aplicamos la definición de 3NF a LOTES con las dependencias fd1 a fd4, encontraremos que fd3 y fd4 violan la 3NF. Por lo tanto podríamos descomponer a LOTES en LOTES1A, LOTES1B y LOTES2 directamente.

- LOTES1 viola la 3NF porque PRECIO es transitoriamente dependiente de cada uno de los candidatos a llave de LOTES1 vía el atributo no primo AREA.

9.4.3 Interpretación de la Definición General de 3NF

Un esquema de relación R viola la definición general de 3NF si una dependencia funcional $X \rightarrow A$ que se retiene en R viola *ambas* condiciones (a) y (b) de la 3NF. Violar (b) implica que A es un atributo no primo. Violar (a) implica que X no es un superconjunto de ninguna llave de R; por lo tanto, X podría ser no primo o podría ser un subconjunto apropiado de una llave de R. Si X es no primo tendremos una dependencia transitoria que viola la 3NF, mientras que si X es un subconjunto propio de una llave de R tendremos una dependencia funcional que viola la 3NF (y también la 2NF). Por lo tanto, podemos establecer una **definición general alterna de la 3NF** como sigue: Un esquema de relación R está en 3NF si cada atributo no primo de R es:

- Completamente dependiente funcional de cada llave de R, y
- No transitoriamente dependiente de cada llave de R.

9.5 Forma de Normalización Boyce-Codd (BCNF)

La forma de normalización Boyce-Codd es más estricta que la 3NF, significa que cada relación en BCNF está también en 3NF; sin embargo, una relación en 3NF *no está necesariamente* en BCNF. Intuitivamente, podemos ver la necesidad de una forma de normalización más fuerte que 3NF regresando al esquema de la relación LOTES de la Figura 9.11(a) con sus cuatro dependencias funcionales fd1 a fd4. Supongamos que tenemos miles de lotes en la relación pero que sólo son de dos condados: Marion y Libertad. También supongamos que el tamaño de los lotes en Marion son únicamente de 0.5, 0.6, 0.7, 0.8, 0.9 y 1.0 acres, mientras que en Libertad los tamaños están restringidos a 1.1, 1.2, ..., 1.9, 2.0 acres. Ante tal situación tendríamos la siguiente dependencia funcional fd5: AREA → NOM_CONDADO. Si agregamos esta a las demás dependencias, el esquema de relación LOTES1A sigue en 3NF debido a que NOM_CONDADO es un atributo primo.

La relación área contra condado representada por fd5 puede representarse mediante 16 tuplas en una relación separada R(AREA, NOM_CONDADO), ya que sólo existen 16 valores de área posibles. Esta representación reduce la redundancia de repetir la misma información en las miles de tuplas LOTES1A. BCNF es una *forma de normalización más fuerte* que deshabilitaría a LOTES1 y sugeriría la necesidad de descomponerla.

Esta definición de Boyce-Codd difiere ligeramente de la definición de 3NF. Un esquema de relación está en **BCNF** si cualquier dependencia funcional X → A se retiene en R, entonces X es una superllave de R. La única diferencia entre BCNF y 3NF es que la condición (b) de 3NF, la cual permite que A sea primo si X no es una superllave, está ausente de BCNF.

En nuestro ejemplo, fd5 viola BCNF en LOTES1A porque AREA no es una superllave de LOTES1A. Notemos que fd5 satisface la 3NF en LOTES1A porque NOM_CONDADO es un atributo primo (condición (b)), pero esta condición no existe en la definición de BCNF. Podemos descomponer a LOTES1A en dos relaciones BCNF LOTES1AX y LOTES1AY, mostradas en la Figura 9.12(a).

En la práctica, la mayoría de los esquemas de relación que están en 3NF están también en BCNF. Sólo si existe una dependencia X → A en un esquema de relación R con X no como una superllave *y* A un atributo primo R estará en 3NF pero no en BCNF. El esquema de relación R mostrado en la Figura 9.12(b) ilustra el caso general de tal relación. Es mejor tener esquemas de relación en BCNF. Si no es posible, 3NF lo hará. Sin embrago, 2NF y 1NF no se consideran buenos diseños de esquemas de relación. Estas formas de normalización fueron desarrolladas históricamente como caminos de paso a 3NF y BCNF.

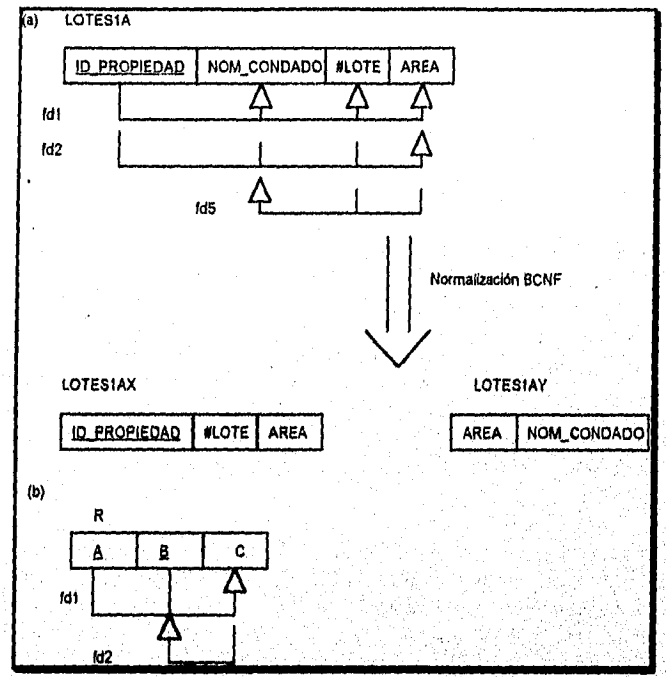
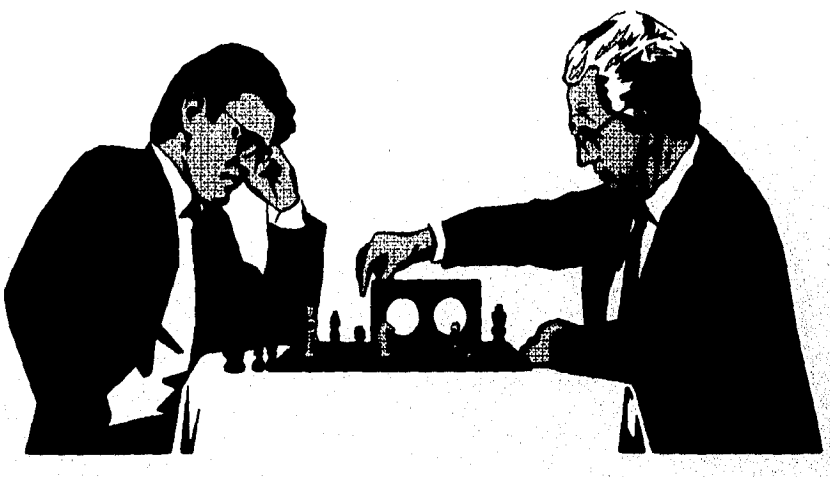


Figura 9.12 BCNF. (a) Normalización BCNF con la dependencia de fd2 "perdida" en la descomposición. (b) Una relación R en 3NF pero no en BCNF.

CAPITULO 10



Panorama del Proceso de Diseño de Bases de Datos

10.1 Rol de los Sistemas de Información en las Organizaciones

10.1.1 Contexto Organizacional para el Uso de Sistemas de Bases de Datos

Ahora discutiremos brevemente cómo los sistemas de bases de datos se han vuelto parte de los sistemas de información de muchas organizaciones. Los sistemas de información en los 60s fueron dominados por sistemas de archivos. Desde principios de los 70s las organizaciones se han estado moviendo gradualmente de estos a bases de datos. Muchas organizaciones han constituido departamentos bajo un administrador de bases de datos (DBA) para preveer y controlar las actividades del ciclo de vida de la base de datos. Similarmente, el manejo de fuentes de información (IRM) se reconoce por grandes organizaciones como una clave del éxito de los negocios. Existen varias razones para ello:

- En las organizaciones un mayor número de funciones están computarizadas, aumentando la necesidad de guardar grandes volúmenes de datos disponibles en un estado actual hasta el momento.

- Conforme crecen la complejidad de los datos y las aplicaciones, necesitan modelarse y mantenerse relaciones complejas entre los datos.

- En muchas organizaciones existe una tendencia hacia la consolidación de las fuentes de información.

Los sistemas de bases de datos satisfacen los tres requerimientos anteriores en gran medida. También son muy útiles otras dos características adicionales en el diseño y manejo de grandes bases de datos:

- La *independencia de datos* protege a los programas de aplicación de cambios en la organización lógica base y en las trayectorias físicas de acceso y estructuras de almacenamiento.

- Los *esquemas externos* (vistas) permiten que los mismos datos se usen en múltiples aplicaciones, cada aplicación tiene su propia vista de los datos.

Una justificación adicional para cambiar a los sistemas de bases de datos es el bajo costo de desarrollo de nuevas aplicaciones en comparasión al costo de los antiguos sistemas de archivos. Esto con frecuencia justifica el alto costo inicial del diseño y conversión del sistema de bases de datos.

Desde principios de los 70s hasta mediados de los 80s, el movimiento fue hacia la creación de depósitos de datos centralizados manejados por un DBMS centralizado. Recientemente, esta tendencia ha sido *invertida*, debido a los siguientes desarrollos:

1. Las computadoras personales y productos de software, tales como Visicalc, Lotus, 123, Simphony, Paradox y dBase IV y V, están siendo muy utilizados por usuarios

quienes previamente pertenecían a la categoría de casuales y ocasionales. Muchos de ellos administradores, ingenieros, científicos, arquitectos, y otros más. Como resultado, está ganando popularidad la práctica de crear **bases de datos personales**. Ahora es posible checar una copia de una parte de grandes bases de datos de un mainframe o un servidor, trabajar en él desde una estación de trabajo, y luego regresarla al mainframe. De manera similar, los usuarios pueden diseñar y crear sus propias bases de datos y cargarlas a una más grande.

2. El advenimiento de manejadores de sistemas de bases de datos está abriendo la opción de distribuir a las bases de datos sobre múltiples sistemas de cómputo para un mejor control y procesamiento local más rápido. Al mismo tiempo, los usuarios locales pueden accesar datos remotos usando los medios proporcionados por el DDBMS.

3. Muchas organizaciones usan ahora **sistemas diccionarios de datos**, los cuales son mini DBMSs que manejan **meta-data** para un sistema -esto es, los datos que describen la estructura, argumentos, aplicaciones, autorizaciones, etc. Estas se usan con frecuencia como una *herramienta integral* para el manejo de fuentes de información. Un diccionario de datos útil debe almacenar y manejar los siguientes tipos de información:

   a. Descipción de los esquemas del sistema de bases de datos.

   b. Información detallada sobre el diseño físico de la base de datos, tales como estructuras de almacenamiento, trayectorias de acceso y tamaños de registros y archivos.

   c. Desripciones de los usuarios de la base de datos, sus responsabilidades, y derechos de acceso.

   d. Descripciones a alto nivel de las transacciones de bases de datos y aplicaciones y de las relaciones de usuarios con las transacciones.

   e. La relación entre transacciones y los elementos de datos referenciados por ellas. Esto es útil para determinar qué transacciones se afectan cuando se cambian las definiciones de datos.

   f. Uso de estadísticas tales como frecuencias de queries y transacciones y conteo de acceso a diferentes porciones de la base de datos.

Esta información está disponible para los administradores, diseñadores, y usuarios autorizados como documentación en línea. Esto mejora el control de los administradores sobre el sistema de información, y los usuarios entienden y usan al sistema. En muchas grandes organizaciones, un sistema de diccionario de datos se considera tan importante como un DBMS.

Recientemente, se ha colocado gran énfasis en los **sistemas de procesamiento de transacciones**, lo cual requiere operaciones sin parar al servicio de la industria. Estas bases de datos se accesan por cientos de transacciones por minuto desde terminales locales y remotas. En tales tipos de sistemas es imperativo un cuidadosos diseño físico de la base de datos que cumpla con las necesidades de procesamiento de transacciones de la organización .

Algunas organizaciones han comprometido su manejo de fuentes de información a ciertos DBMSs y diccionarios de datos. Su inversión en el diseño e implementación de grandes y complejos sistemas les dificulta cambiar a nuevos productos, así las organizaciones quedan bloqueadas en el sistema actual. Con relación a tales sistemas, no se puede echar de menos la importancia de un cuidadoso diseño que tome en cuenta las necesidades de posibles modificaciones en el futuro debido a cambios en los requerimientos. El costo puede ser muy alto si el sistema no evoluciona y se hace necesario el cambiar a otros DBMSs.

10.1.2 Ciclo de Vida del Sistema de Información

En una gran organización, el sistema de bases de datos es típicamente parte de un sistema de información mucho mayor que se usa para manejar los recursos de la organización. Un **sistema de información** incluye todos los recursos dentro de la organización que se ven envueltos en la colección, manejo, uso y diseminación de información. En un medio ambiente computarizado, estos recursos incluyen los datos, el software del DBMS, el hardware y medio de almacenamiento, el pesonal que usa y maneja los datos (DBA, usuarios finales, usuarios paramétricos, y otros), el software de aplicación que accesa y actualiza los datos, y los programadores que desarrollaron las aplicaciones.

Al ciclo de vida de un sistema de información se le llama **macro ciclo de vida**, mientras que al ciclo de vida del sistema de bases de datos se le llama **micro ciclo de vida**. El macro ciclo de vida incluye las siguientes fases:

1. Análisis de factibilidad: Se preocupa de analizar áreas potenciales de aplicaciones, realizando estudios costo-beneficio, y poniendo prioridades entre las aplicaciones.

2. Requerimientos de colección y análisis: Los requerimientos detallados se colectan interactuando con usuarios potenciales para identificar sus problemas y necesidades en particular.

3. Diseño: Esta fase tiene dos aspectos: el diseño del sistema de base de datos, y el diseño de los sistemas de aplicación (programas) que usan y procesan a la base de datos.

4. Implementación: Se implementa al sistema de información, se carga la base de datos, y se implementan y prueban las transacciones.

5. Validación y aceptación de la prueba: Se valida la aceptabilidad del sistema en cumplir los requerimientos de los usuarios y el criterio de desempeño. El sistema se prueba contra criterios de desempeño y especificaciones de comportamiento.

6. Operación: Puede ser precedido por la conversión de usuarios de un sistema anterior, así como de entrenamiento. La fase operacional inicia cuando todas las funciones del sistema operan y han sido validadas. Conforme surgen nuevos requerimientos o aplicaciones, pasan a través de las fases anteriores hasta que son validadas y se incorporan al sistema. El monitoreo de desempeño y mantenimiento del sistema son actividades importantes durante la fase operacional.

10.1.3 Ciclo de Vida de la Aplicación del Sistema de Base de Datos

Las actividades relacionadas con el ciclo de vida de la aplicación del DBS (micro) incluyen las siguientes fases:

1. Definición del sistema: Se define el alcance del sistema de base de datos, sus usuarios, y sus aplicaciones.

2. Diseño: Al final de esta fase, está listo un diseño lógico y físico completo del sistema de base de datos en el DBMS elegido.

3. Implementación: Comprende el proceso de escribir los definiciones conceptuales, externas e internas de la base de datos, creación de archivos vacíos, e implementación de software de aplicación.

4. Carga o conversión de datos: La base de datos se popula ya sea cragando los datos directamente o conviertiendo los archivos existentes al nuevo formato.

5. Aplicación de conversión: cualquier aplicación de software de un sistema anterior se convierte al nuevo sistema.

6. Prueba y validación: Se prueba y se valida al nuevo sistema.

7. Operación: Se ponen en operación al sistema de base de datos y sus aplicaciones.

8. Monitoreo y mantenimiento: Durante la fase operacional, se monitorea y se mantiene constantemente al sistema. El crecimiento y la expansión pueden ocurrir en la información contenida y el software de aplicaciones. De vez en cuando puede ser necesario la modificación y reorganización.

Las actividades 2, 3 y 4 son parte de las fases de diseño e imlementación del ciclo de vida de un gran sistema de información. Los pasos de conversión (4 y 5) no son aplicables cuando la base de datos y las aplicaciones son nuevas. Cuando una organización cambia de un sistema anteriormente establecido a uno nuevo, las actividades 4 y 5 tienden a ser las

más consumidoras de tiempo y esfuerzo para completarlas y también las más menospreciadas. En general, existe retroalimentación entre los distintos pasos debido a que frecuentemente surgen nuevos requerimientos en cada estado.

10.2 El Proceso de Diseño de la Base de Datos

Ahora nos enfocaremos en el paso 2 del ciclo de vida de la aplicación de base de datos, al cual llamamos diseño de la base de datos. El problema del diseño puede establecerse como sigue: *Diseñar la estructura lógica y física de una o más bases de datos para acomodar la información a las necesidades de los usuarios en una organización para un conjunto de aplicaciones definido.*

Las *metas* del diseño son diversas: satisfacer los requerimientos de información de los usuarios y aplicaciones especificados; para proporcionar un estructuramiento natural y fácil de entender de la información; y soportar los *requerimientos de procesamiento* y cualquier objetivo de desempeño tales como el tiempo de respuesta, de procesamiento, y espacio de almacenamiento. Estas metas son muy fáciles de completar y medir. El problema se agraba porque el proceso de diseño casi siempre inicia con requerimientos informales y pobremente definidos. Por el contrario, el resultado de la actividad de diseño es un esquema rígidamente definido que no puede modificarse fácilmente una vez que se implementa la base de datos. En el proceso de diseño podemos identificar seis fases principales:

1. Colección y análisis de requerimientos.
2. Diseño conceptual de la base de datos.
3. Elección de un DBMS.
4. Mapeo del modelo de datos (también llamado diseño lógico de la base de datos).
5. Diseño físico de la base de datos.
6. Implementación del sistema de base de datos.

Como se ilustra en la Figura 10.1 el proceso de diseño consta de dos actividades paralelas. La primera involucra el diseño del **contenido y estructura** de la base de datos; la segunda se relaciona con el diseño de las **aplicaciones y software de procesamiento**. Estas dos aplicaciones están íntimamente entrelazadas. Por ejemplo, podemos identificar elementos de datos que se almacenarán en la base y serán analizados por las aplicaciones. Además, de la fase del diseño físico, durante la cual elegimos las estructuras de almacenamiento y trayectorias de acceso de los archivos, dependen de las aplicaciones que se usarán en estos archivos. Por otro lado, usualmente especificamos el diseño de las aplicaciones refiriendo al esquema, los cuales se especifican durante la primer actividad. Claramente, estas dos actividades influencian fuertemente a la otra. Tradicionalmente, las metodologías del

diseño de bases de datos se han enfocado principalmente en una u otra de estas actividades; esto puede llamarse **manejo de datos** o **manejo de proceso**.

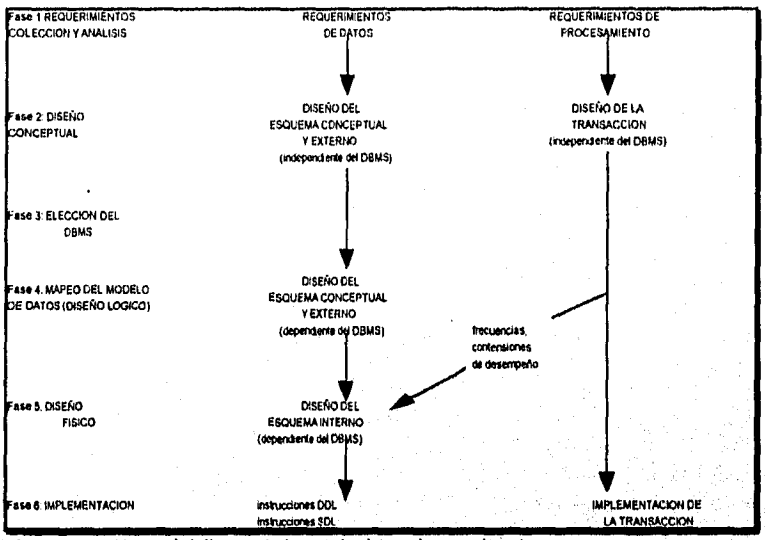


Figura 10.1 Fases del diseño de bases de datos de grandes sistemas.

Las seis fases mencionadas anteriormente no deben proceder exactamente en esa secuencia. En muchos casos se tiene que modificar el diseño de una fase temprana durante una fase posterior. Estos **ciclos de retroalimentación** entre fases -y también dentro de fases- son comunes durante el diseño de la base de datos. La fase 1 de la Figura 10.1 involucra colectar información acerca del uso de la base de datos, mientras que la fase 6 se ocupa de la implementación de la base de datos. Algunas veces las fases 1 y 6 no se consideran para el diseño per se, pero sí parte del ciclo de vida del sistema de información general. El corazón del proceso de diseño de la base de datos son las fases 2, 4 y 5, las cuales resumiremos brevemente:

- *Diseño conceptual de la base de datos (fase 2)*: La meta de esta fase es producir un esquema conceptual independiente a un DBMS específico. Durante esta fase a menudo se usa un modelo de datos de alto nivel tal como el modelo ER o EER. Además, se especifican el mayor número de aplicaciones o transacciones posibles, usando una notación independiente a cualquier DBMS específico.

- *Mapeo del modelo de datos (fase 4)*: A esto también se le llama **diseño lógico de la base de datos**. Durante esta fase **mapeamos** (o **transformamos**) el esquema conceptual del modelo de datos de alto nivel usado en la fase 2 al modelo de datos del DBMS elegido en la fase 3. Podemos iniciar esta fase después de elegir un modelo de implementación de datos, en lugar de esperar que sea elegido un DBMS en específico-

por ejemplo, si decidimos usar algun DBMS relacional pero no se ha decidido por cuál. Al último lo llamamos diseño lógico *independiente del sistema* (pero *dependiente del modelo de datos*). El resultado de esta fase es un *esquema conceptual* en el modelo de datos elegido. Además, durante esta fase se hace el diseño de *esquemas externos* (vistas) para aplicaciones específicas.

- *Diseño físico de la base de datos (fase 5)*: Durante esta fase diseñamos las especificaciones del sistema almacenado en términos de estructuras físicas de almacenamiento, colocación de registros, y trayectorias de acceso.

En las siguientes secciones discutiremsos cada una de las seis fases del diseño de bases de datos.

10.2.1 Fase 1: Colección y Análisis de Requerimientos

Antes de poder diseñar efectivamente una base de datos, debemos conocer las expectativas de los usuarios y los usos de la base de datos tanto como sea posible. Al proceso de identificar y analizar los usos se le llama **colección y análisis de requerimientos**.

Para especificar los requerimientos, primero debemos identificar las otras partes del sistema de información que interactuarán con el sistema de base de datos. Estas incluyen a los usuarios y aplicaciones existentes. Entonces se colectan y analizan los requerimientos de estos usuarios y aplicaciones. Típicamente, las siguientes actividades son parte de esta fase:

- Identificación de las áreas de aplicación y grupos de usuarios. Se eligen claves individuales dentro de cada grupo como los participantes principales en los pasos subsecuentes de colección y especificación de requerimientos.

- Se estudia y analiza la documentación concerniente a las aplicaciones. Se revisa otra documentación -manuales de políticas, formas, reportes, y organigramas- para determinar si tiene alguna influencia en el proceso anterior.

- Se estudia el medio ambinte actual y el uso planeado de la información. Incluyendo el análisis de los tipos de transacciones y sus frecuencias, así como el flujo de información dentro del sistema. Se especifican la entrada y salida de datos para las transacciones.

- Se colectan respuestas escritas a cuestionarios formulados a usuarios potenciales. Estas preguntas involucran las prioridades y la importancia que le dan a cada una de las aplicaciones. Puede entrevistarse a individuos clave para ayudar en el asesoramiento de la información valiosa y la colocación de prioridades.

Los modos anteriores de colectar requerimientos dan lineas pobremente estructuradas y casi informales, las cuales entonces se transforman en una forma mejor estructurada usando alguna de las **técnicas de especificación de requerimientos**. Estas incluyen diagramas HIPO (entrada proceso salida), SADT (análisis estructurado y técnica de diseño), DFDs (diagramas de flujo de datos). Diagramas Orr-Warnier y Nassi-Schneiderman. Todos estos son métodos diagramales de organizar y presentar los requerimientos de información-procesamiento. Los diagramas pueden tomar la forma de jerarquías, diagramas de flujo, estructuras secuenciales y de ciclos, o alguna otra, dependiendo del método. Usualmente acompañan a los diagramas información adicional como texto, tablas, gráficas y requerimientos de decisión. Esta fase puede ser un poco consumidora de tiempo, pero es crucial en el futuro éxito del sistema de información.

Para tratar con la colección y análisis de requerimientos se han propuesto algunas técnicas asistidas por computadora. Estas técnicas incluyen herraminetas automatizadas, que checan la consistencia y completitud de especificaciones. Los requerimientos se almacenan en un solo lugar, usualmente llamada la base de datos de diseño, y puede desplegarse y actualizarse conforme avanza el diseño.

10.2.2 Fase 2: Diseño Conceptual de la Base de Datos

La segunda fase del diseño de bases de datos involucra dos actividades paralelas. La primera, **diseño del esquema conceptual**, examina los requerimientos resultantes de la fase 1 y produce un esquema conceptual de la base de datos. La segunda actividad, **diseño de transacciones**, examina las aplicaciones analizadas en la fase 1 y produce especificaciones de alto nivel para estas transacciones.

Fase 2a: Diseño del Esquema Conceptual. El esquema conceptual producido en esta fase está usualmente contenido en un modelo de datos de alto nivel independiente de un DBMS y por lo tanto no puede usarse para implementar directamente la base de datos. No debe menospreciarse la importancia de un esquema conceptual independiente de un DBMS, por las siguientes razones:

1. La meta del esquema conceptual es un completo entendiemiento de la estructura de la base de datos, significado (semántica), interrelaciones, y argumentos. Esto se logra independientemente de un DBMS especifico porque cada DBMS tiene sus propias idiosincracias y restricciones que no deben influir al diseño del esquema conceptual.

2. El esquema conceptual es invaluable como una *descripción estable* de los contenidos de la base de datos. La elección del DBMS y posteriores decisiones de diseño pueden cambiar sin modificar al esquema independiente del DBMS.

3. Es crucial el buen entendimiento del esquema conceptual para los usuarios y diseñadores de aplicaciones. Es poco importante el uso de un modelo de datos de alto nivel que sea más expresivo y general que los modelos de datos de DBMSs individuales.

4. La descripción diagramal del esquema conceptual puede servir como un excelente vehículo de comunicación entre usuarios, diseñadores, y analistas. Debido a que los modelos de datos de alto nivel se basan en conceptos que son más sencillos de entender que los modelos de bajo nivel, cualquier comunicación concerniente al diseño del esquema se vuelva más exacta y directa.

En esta fase del diseño, es importante usar un modelo de datos de alto nivel -también conocido como modelo semántico o conceptual- que tenga las siguientes características:

1. Expresividad: Debe ser lo suficientemente expresivo para distinguir diferentes tipos de datos, relaciones y argumentos.

2. Simplicidad: Debe ser lo suficientemente sencillo para que los usuarios no especialistas entiendan y usen sus conceptos.

3. Mínimo: Debe tener un número pequeño de conceptos básicos que sean distintos y no confusos en significado.

4. Representación diagramal: El modelo debe tener una notación diagramal para desplegar un esquema conceptual que sea fácil de interpretar.

5. Formalidad: Un esquema conceptual expresado en el modelo de datos debe representar una especificación formal no ambigua de los datos. Por lo tanto, los conceptos del modelo deben definirse exactamente y sin ambiguedades.

Estos requerimientos son algunas veces conflictivos. En particular, el requerimiento 1 entra en conflicto con los demás.

Fase 2b: Diseño de Trnasacciones. El propósito de la fase 2b, la cual procede en paralelo con la fase 2a, es diseñar de manera independiente las características de las transacciones conocidas en un DBMS. Cuando está siendo diseñado un sistema de base de datos, los diseñadores se ocupan de muchas aplicaciones conocidas (o **transacciones**) que correrán en la base de datos una vez que se implemente. Una parte importante del diseño de bases de datos es especificar las características funcionales de estas transacciones durante el proceso de diseño. Esto asegura que el esquema incluirá toda la información requerida por estas transacciones. Además, el conocimiento de la importancia relativa de las distintas transacciones y frecuencias esperadas de su invocación juegan una parte crucial en el diseño físico de la base de datos (fase 5). Usualmente, sólo se conocen algunas de las transacciones al momento del diseño; después de implementarse el sistema, se identifican e implementan continuamente transacciones. Sin embargo, las transacciones más importantes se conocen al momento de la implementación del sistema y se especifican en un estado temprano.

Una técnica común para especificar transacciones a nivel conceptual es identificar sus **entradas/salidas** y comportamiento **funcional**. Especificando los datos de entrada, salida

y flujo interno de control funcional, los diseñadores pueden especificar una transacción de manera conceptual e independiente del sistema. Las transacciones usualmente se agrupan en tres categorías: recuperación, actualización y mixtas. Las **transacciones de recuperación** se usan para desplegar datos en pantalla o producir un reporte. Las **transacciones de actualización** se usan para ingresar nuevos datos o modificar datos existentes en la base. Las **transacciones mixtas** se usan para aplicaciones más complejas que hacen alguna recuperación y algo de actualización. Por ejemplo, supongamos que estamos diseñando una base de datos de reservaciones aéreas. Un jemplo de transacción de recuperación es listar todos los vuelos matutinos entre dos ciudades para determinada fecha. Un ejemplo de transacción de actualización es reservar un lugar en un vuelo en particular. Una transacción mixta puede desplegar primero algunos datos, tales como mostrar a un cliente la reservación de algun vuelo, y luego actualizar a la base de datos, tal como cancelar la resevación borrándola.

10.2.3 Fase 3: Elección de un DBMS

La elección de un DBMS está gobernada por ciertos factores. Algunos son técnicos, otros económicos, y otros tienen que ver con políticas de la organización. Los factores técnicos tienen que ver con la aconsejabilidad del DBMS (relacional, red, jerárquico, orientado a objetos, u otro), estructuras de almacenamiento y trayectorias de acceso que soporta, las interfaces de usuario y programador disponibles, los tipos de lenguajes query de alto nivel, etc. En esta sección nos concentraremos en discutir los factores económicos y organizacionales que afectan la elección de un DBMS. Cuando se selecciona un DBMS deben considerarse los siguientes costos:

1. Costo de adquisición del software: Este es el costo de comprar el software, incluyendo opciones de lenguajes, diferentes interfaces tales como formas y pantallas, opciones de recuperación/respaldo, métodos de acceso especiales, y documentación. Debe selecionarse la versión correcta de un DBMS para un sistema operativo en específico.

2. Costo de mantenimiento: Es el costo recurrente por recibir servicio de mantenimiento del vendedor y mantener la versión actualizada del DBMS.

3. Costo de adquisición del hardware: Puede ser necesario hardware nuevo, tal como memoria adicional, terminales, unidades de disco, o almacenamiento especializado para el DBMS.

4. Costo de la creación y conversión de la base de datos: Es el costo de crear al sistema de bases de datos desde cero o convertir un sistema existente al nuevo DBMS. En el último caso es habitual operar el sistema existente en paralelo con el nuevo sistema hasta que estén completamente implementadas y probadas todas las aplicaciones nuevas. Este costo es difícil de proyectar y con frecuencia se menosprecia.

5. Costo personal: La adquisición del DBMS por primera vez en una organización se acompaña de una reorganización del departamento de procesamiento de datos. En la

mayoría de las compañías que adoptan DBMSs se crean nuevas posiciones del administrador (DBA) y staff.

6. Costo de entrenamiento: Debido a que los DBMSs son productos complejos, debe de entrenarse al personal para usar y programar al DBMS.

7. Costo de operación: El costo de operación continua del sistema no se tomó en cuenta en la evaluación de alternativas debido a que se incurre sin importar el DBMS seleccionado.

Los beneficios de adquirir un DBMS no son fáciles de medir y cuantificar. Un DBMS tiene distintas ventajas intangibles sobre los sistemas tradicionales de archivos, tales como facilidad de uso, mayor disponibilidad de datos, y acceso más rápido a la información. Dentro de los beneficios más tangibles están la reducción del costo de desarrollos, reducción de la redundancia, y mejor control y seguridad. Una organización debe decidir cuando cambiar a un DBMS basado en un análisis costo-beneficio. Este movimiento se lleva a cabo generalmente por los siguientes factores:

- Complejidad de los datos: Conforme las relaciones se vuelven más complejas, se siente con mayor fuerza la necesidad de un DBMS.

- Compartimiento entre aplicaciones: Mientras es mayor el compartimiento entre aplicaciones, mayor redundancia entre los archivos, y por tanto mayor necesidad de un DBMS.

- Evolución dinámica o crecimiento de datos: Si los datos cambian constantemente, es más sencillo cumplir con estos cambios usando un DBMS que un sistema de archivos.

- Frecuencia de requisiciones ad hoc: Los sistemas de archivos no son del todo aconsejables para la recuperación de datos.

- Volúmen de datos y necesidad de control: El tamaño de los datos y la necesidad de control demanda algunas veces de un DBMS.

Finalmente, varios factores económicos y organizacionales afectan la elección de un DBMS sobre otro:

1. Estructura de los datos: Si los datos a almacenar siguen una estructura jerárquica, debe considerarse un DBMS jerárquico. Para datos con muchas interrelaciones, un sistema de red o relacional puede ser más apropiado. La tecnología relacional está ganando popularidad. Para estructuras complejas, se aconsejan los sistemas orientados a objetos.

2. Familiaridad del personal con el sistema: Si el staff de programación dentro de la organización está familizarizado con un DBMS en particular, puede favorecer la reducción del costo de entrenamiento y tiempo de aprendizaje.

3. Disponibilidad de servicios del vendedor: Es deseable la asistencia del vendedor en la solución de problemas del sistema.

Antes de comprar un DBMS, una organización debe tomar en cuenta la configuración hardware/software requerida para correr el DBMS y la portabilidad entre diferentes tipos de hardware. Muchos DBMSs comerciales tienen ahora versiones que corren en muchas configuraciones hardware/software (o **plataformas**). También debe considerarse la necesidad de aplicaciones de respaldo, recuperación, desempeño, integridad, y seguridad. Dentro de las organizaciones muchos DBMSs están siendo diseñados como *soluciones totales* para el procesamiento y manejo de recursos de información. La mayoría de los DBMSs están combinando sus productos con las siguientes opciones o características internas, las cuales a menudo se categorizan como 4GL por los vendedores comerciales:

- Editores de texto y browsers
- Generadores de reportes y utilerías de listados
- Software de comunicaciones (a menudo llamado monitores de teleproceso)
- Entrada de datos y opciones de despliegue tales como formas, pantallas y menús con opciones de edición automáticas.
- Herramientas gráficas de diseño.

En algunos casos puede no ser apropiado usar un DBMS; en su lugar, puede ser preferible desarrollar software casero. Este puede ser el caso de aplicaciones que están muy bien definidas y se conocen todas de antemano. En tal caso, puede ser apropiado un sistema diseñado en casa para implementar las aplicaciones conocidas en la forma más eficiente. En la mayoría de los casos, surgen después de la implementación del sistema nuevas aplicaciones que no fueron previstas al momento del diseño. Esto es precisamente por lo que los DBMSs se han vuelto tan populares: facilitan la incorporación de nuevas aplicaciones sin mayores cambios al sistema existente.

10.2.4 Fase 4: Mapeo del Modelo de Datos (Diseño Lógico de la Base de Datos)

La siguiente fase del diseño es crear un esquema conceptual y esquemas externos en el modelo de datos del DBMS seleccionado. Esto se logra mapeando los esquemas conceptual y externo producidos en la fase 2a desde el modelo de datos de alto nivel hasta el modelo del DBMS. El mapeo puede llevarse a cabo en dos estados:

1. Mapeo independiente del sistema: En este paso el mapeo al modelo de datos del DBMS no considera ninguna característica específica o casos especiales que apliquen a la implementación del modelo de datos en el DBMS.

2. Uniendo los esquemas a un DBMS específico: Diferentes DBMSs implementan un modelo de datos usando opciones y argumentos de modelaje en específico. Podemos tener que ajustar los esquemas obtenidos en el paso 1 para conformar las opciones de implementación específicas del modelo de datos como se usó en el DBMS seleccionado.

El resultado de esta fase deben ser instrucciones DDL en el lenguaje del DBMS seleccionado que especifiquen los esquemas conceptual y externos del sistema. Pero si las instrucciones DDL incluyen algunos parámetros físicos de diseño, una especificación completa DDL debe esperar hasta después de haber completado el diseño físico de la base de datos. Muchas herramientas de diseño CASE pueden generar DDL para sistemas comerciales desde un diseño de esquema conceptual.

10.2.5 Fase 5: Diseño Físico de la Base de Datos

El diseño físico es el proceso de elegir estructuras de almacenamiento específicas y trayectorias de acceso para que los archivos adquieran un buen desempeño en las distintas aplicaciones. Cada DBMS ofrece una variedad de opciones para la organización de archivos y trayectorias de acceso. Estas usualmente incluyen varios tipos de indexado, agrupamento de registros relacionados en bloques de disco, enlace de registros relacionados vía apuntadores, y varios tipos de seccionamiento. Una vez que se elige un DBMS específico, el proceso del diseño físico se restringe a elegir las estructuras más apropiadas para lós archivos de entre las opciones ofrecidas por el DBMS. Para guiar la elección de las opciones del diseño físico de la base de datos se usa el siguiente criterio:

1. Tiempo de respuesta: Es el tiempo transcurrido entre submitir una transacción para ejecución y recibir la respuesta. Una gran influencia en el tiempo de repuesta que está bajo control del DBMS es el tiempo de acceso para elementos de datos referenciados por la transacción. El tiempo de respuesta también se ve influenciada por factores que no están bajo control del DBMS, tales como el sistema de carga, el calendario del sistema operativo, o retrasos de comunicaciones.

2. Utilización de espacio: Es la cantidad de espacio de almacenamiento usado por los archivos de la base de datos y sus trayectorias de acceso.

3. Seguimiento de transacción: Es el número de transacciones promedio por minuto que pueden procesarse por el sistema; es un parámetro crítico de sistemas de transacción tales como aquéllos usados en reservaciones aéreas o bancos. El seguimiento de transacción debe medirse bajo condiciones pico en el sistema.

El desempeño depende del tamaño del registro y número de registros en el archivo. Por lo tanto, debemos estimar estos parámetros por archivo. Además, debemos estimar los patrones de actualización y recuperación para el archivo acumulativamente desde todas las transacciones. Los atributos usados para seleccionar registros deben tener trayectorias de acceso primario e índices secundarios construidos para ellos. Estimativos del crecimiento de archivos, ya sea en el tamaño del registro debido a nuevos atributos o en el número de registros, debe tomarse en cuenta durante el diseño físico de la base de datos.

El resultado de la fase de diseño físico es una determinación *inicial* de estructuras de almacenamiento y trayectorias de acceso para los archivos. Casi siempre es necesario **ajustar** el diseño en base a su desempeño observado después de la implementación del sistema. La mayoría de los sistemas incluyen una utilería de monitoreo para colectar estadísticas de desempeño, las cuales se guardan en el catálogo del sistema o diccionario de datos para análisis posterior. Estas incluyen estadísticas sobre el número de invocaciones de transacciones o queries predefinidos, actividad de entrada/salida contra archivos, conteo de páginas de archivos o registros índices, y frecuencia de uso del índice. Conforme cambian los requerimientos del sistema, se vuelve necesario reorganizar algunos archivos construyendo nuevos índices o métodos de acceso primario.

10.2.6 Fase 6: Implementación del Sistema de Base de Datos

Después de haber complementado los diseños físicos y lógicos, podemos implementar el sistema de bases de datos. Para crear los esquemas y archivos (vacíos) de bases de datos se compilan y se usan las instrucciones en DDL y SDL del DBMS seleccionado. Entonces puede **cargarse** (popularse) con los datos. Si los datos van a convertirse de un sistema anterior, pueden ser necesarias **rutinas de conversión** para reformatear los datos cargándolos a la nueva base de datos.

En este estado deben implementarse las transacciones por los programadores de aplicaciones. Se llevan a cabo las especificaciones conceptuales de transacciones, y el correspondiente código con comandos DML internos escritos y probados. Una vez que las transacciones están listas y los datos se cargan a la base, la fase de diseño e implementación ha finalizado e inicia la fase operacional del sistema.

10.3 Diseño Físico de la Base de Datos

En esta sección discutiremos los factores del diseño físico que afectan el desempeño de aplicaciones y transacciones; luego comentaremos las guías para los DBMSs relacionales, de red y jerárquico.

10.3.1 Factores que Influencían el Diseño Físico de la Base de Datos

El diseño físico es una actividad donde la meta no solo es tener la estructura adecuada en almacenamiento garantizando un buen desempeño. Para un esquema conceptual dado, existen muchas alternativas de diseño en un DBMS dado. No es posible tomar decisiones

significativas y análisis de desempeño hasta conocer los queries, transacciones, y aplicaciones que se espera corran en la base de datos. Se deben analizar estas aplicaciones, sus frecuencias de invocación esperadas, cualquier argumento en su ejecución, y la frecuencia de operaciones de actualización esperadas.

Análisis de los Queries y Transacciones. Antes de emprender el diseño físico, debemos tener claro el uso de la base de datos definiendo los queries y transacciones que esperamos corran de forma aceptable. Para cada *query*, debemos especificar lo siguiente:

1. Los archivos que serán accesados por el query.
2. Los campos sobre los cuales se especifica cualquier condición de selección.
3. Los campos sobre los cuales se especifican condiciones de unión o enlace de múltiples tipos de registros.
4. Los campos cuyos valores serán recuperados por el query.

Los campos listados en los puntos 2 y 3 son candidatos para la definición de estructuras de acceso. Para cada actualización u operación, debemos de especificar lo siguiente:

1. Los archivos que serán actualizados.
2. El tipo de operación de actualización sobre cada archivo(insertar, modificar o borrar).
3. Los campos sobre los cuales cualquier condición de selección debe especificar una operación de borrar o modificar.
4. Los campos cuyos valores cambiarán por una operación de modificación.

Una vez más, los campos del punto 3 son candidatos para estructuras de acceso. Por otro lado, los campos listados en el punto 4 son candidatos de evitar una estructura de acceso, debido a que la modificación de ellos requerirá el actualizar a las estructuras de acceso.

Análisis de la Frecuencia Esperada de Queries y Transacciones. Además de identificar las características de los queries y transacciones esperadas, debemos considerar sus frecuencias de invocación esperadas. Esta información, junto con la de los campos colectados en cada query y transacción, se usa para compilar una lista acumulativa del uso de todos los queries y transacciones. Esto se expresa como la frecuencia esperada de uso de cada campo en cada archivo como campo de selección o de unión, sobre todos los queries y transacciones. Generalmente, para grandes volúmenes de procesamiento, se aplica la regla general "80-20". La regla establece que aproximadamente el 80% es procesamiento y sólo un 20% es de queries y transacciones. Por lo tanto, en situaciones prácticas raramente es necesario colectar estadísticas exhaustivas y frecuencias de

invocación en todos los queries y transacciones; es suficiente determinar el 20% o a los más importantes.

Análisis de los Argumentos de Tiempo de Queries y Transacciones. Algunos queries y transacciones pueden tener argumentos rígidos de desempeño. Por ejemplo, cierta transacción puede tener el argumento de que debe terminar en 5 segundos en el 95% de las ocasiones en que se invoque y nunca tardar mas de 20 segundos. Tales consideraciones pueden usarse para colocar prioridades sobre los campos que son candidatos a trayectorias de acceso. Los campos de selección usados por queries y transacciones con restricciones de tiempo se vuelven candidatos a estructuras de acceso de alta prioridad.

Análisis de las Frecuencias de Actualización. Para un archivo que se actualiza frecuentemente debe especificarse una cantidad mínima de trayectorias de acceso, debido a que la actualización de las trayectorias de acceso hace lentas las operaciones de actualización.

Una vez compilada la información anterior, podemos tomar las decisiones del diseño físico, lo cual consiste principalmente en decidir las estructuras de almacenamiento y trayectorias de acceso de los archivos. Raramente se conocen todos los queries y transacciones al momento del diseño fisco de la base de datos. Con frecuencia surgen nuevas aplicaciones después de haber implementado el sistema, requiriendo la especificación de nuevos queries y transacciones. En tales casos es con frecuencia necesario modificar algunas de las decisiones del diseño físico para poder incorporar las nuevas aplicaciones al sistema. A esto se le llama **sintonizar** el diseño físico. Si algunas de las transacciones o queries con restricciones de tiempo fallan en cumplir los tiempos de respuesta especificados, son necesarias modificaciones al diseño fisco para mejorar la eficiencia de las transacciones.

10.3.2 Diseño Físico de Bases de Datos Relacionales

La mayoría de los sistemas relacionales representan cada relación como un archivo de la base de datos. Las opciones de trayectorias de acceso incluyen especificar el tipo de archivo para cada relación y los atributos sobre los cuales se definirán los índices. Uno de los índices de cada archivo puede ser un índice primario o de agrupamiento. Además, existen varias técnicas de acelerar las operaciones EQUIJOIN o JOIN NATURAL. Primero discutiremos estas técnicas y luego elegiremos organizaciones de archivos y selección de índices.

Técnicas para Acelerar las Operaciones Equijoin o Join Natural. Algunos sistemas relacionales ofrecen la opción de almacenar dos relaciones con relación 1:N entre ellas, representada por una *llave externa*, como un archivo jerárquico de dos niveles, donde se almacena a cada registro del lado 1 (llave primaria), seguido por los registros del lado N (con las llaves externas correspondientes). Este tipo de estructura de almacenamiento hace muy eficiente el join entre los dos archivos en esta relación 1:N. Por ejemplo, consideremos el esquema relacional en la Figura 6.5, y supongamos que cada relación está

implementada como un archivo. El Equijoin con condición EMPLEADO.IMSS = TRABAJA_EN.IMSS debe hacerse tan eficiente como sea posible, podemos usar un archivo mixto de los registros EMPLEADO y TRABAJA_EN. Cada registro EMPLEADO se almacenará seguido por los registros TRABAJA_EN que tengan el mismo valor de IMSS. También es posible *agrupar* los registros TRABAJA_EN en IMSS mientras se mantienen archivos separados.

Otra opción es **desnormalizar** el esquema lógico de la base de datos cuando se diseñen los archivos físicos. Esto se hace para situar los atributos que se requieren con mayor frecuencia en un query en los mismos registros con otros atributos involucrados en el query. Esto lo hacemos **replicando (o duplicando)** los atributos en el archivo donde son necesarios. Debemos compensar esta desnormalización requiriendo que las transacciones de actualización mantengan los valores de los atributos duplicados consistentes con el otro. Por ejemplo, supongamos que el mismo query que requiere la operación JOIN anterior también requiere la recuperación del atributo PNOMBRE del registro PROYECTO con PROYECTO.PNUMERO = TRABAJA_EN.PNO. Podemos replicar el atributo PNOMBRE en los registros TRABAJA_EN de tal forma que el último join no se ejecute. Esta técnica puede llevarse un paso más allá almacenando físicamente un archivo que sea el resultado del JOIN de dos archivos, a pesar de que esta desnormalización extrema debe usarse sólo con sumo cuidado. Todo tipo de anomalías pueden ocurrir y deben tratarse explícitamente cada que se aplican actualizaciones al archivo.

Selección de Organización de Archivos y Selección de Índices. Una opción popular para organizar un archivo separado en un sistema relacional es guardar los registros desordenados y crear tantos índices secundarios como sean necesarios. Los atributos que se usan para condiciones de selección y join son candidatos a índices secundarios.

Otra opción es especificar un atributo de ordenamiento para el archivo especificando un índice primario o de agrupamiento. El atributo más usado para operaciones join debe elegirse para ordenar o agrupar registros, debido a que hace más eficiente la operación join. Si los atributos son accesados con frecuencia por el orden de un atributo, es otro indicador de que debe elegirse para ordenar a los registros. Sólo uno de los atributos de cada archivo debe elegirse para ordenar el archivo físicamente, con un índice primario correspondiente (si el atributo es una llave) o un índice de agrupamiento (si el atributo no es una llave). Muchos sistemas relacionales usan la instrucción UNIQUE para especificar una llave y CLUSTER para especificar un índice de agrupamiento.

10.3.3 Diseño Físico de Bases de Datos para Sistemas de Red

Existen varias opciones importantes para una base de datos de red. La primera involucra decidir sobre la implementación de cada tipo de conjunto. Otra opción, similar a la desnormalización, involucra decidir si alguno de los campos de un registro propietario debe replicarse en un miembro para propósitos de eficiencia. Además, algunos DBMSs de red permiten la definición de llaves seccionadas o índices sobre un tipo de registro. Estos, junto con el tipo de conjunto propiedad del SISTEMA, proporcionan puntos de entrada a

la base de datos, los cuales con frecuencia son seguidos por *navegación desde el punto de entrada* trazando los apuntadores de conjuntos (**procesamiento de conjunto**). Finalmente, deben tomarse decisiones sobre cualquier campo de ordenamiento o tipos de registro o miembros dentro de un tipo de conjunto.

Guías para Elegir entre Opciones de Implementación de Conjuntos. Existen muchas opciones para implementar un tipo de conjunto. Ahora las revisaremos, y presentaremos algunas guía para decidir qué opción usar:

1. Implementar un conjunto por colindancia física: En algunos DBMSs de red, los miembros pueden almacenarse físicamente siguiendo al propietario; a esta opción se le llama implementación de conjunto por **colindancia física**. Si un tipo de registro es un miembro de varios tipos de conjuntos, *sólo uno de ellos* puede elegirse para la implementación por colindancia física. El acceso a miembros de una instancia es más eficiente en esta opción, así los conjuntos que se usan con mayor frecuencia para accesar a *todos los miembros* de un propietario son candidatos para este tipo de implementación.

2. Implementación de un conjunto por arreglos de apuntadores: Otra opción es almacenar un arreglo de apuntadores a los miembros con el registro propietario. Si se selecciona a un *sólo miembro* por su orden dentro del conjunto (tal como PRIMERO, ULTIMO, o iésimo) y con frecuencia a partir del propietario, esta es una buena opción.

3. Uso de diferentes opciones para implementación de apuntadores: La mayoría de los conjuntos se implementan por apuntadores y listas enlazadas. Podemos tener un sencillo apuntador **next** o **next** y **prior** en los miembros de un conjunto. Con cualquiera de estas dos opciones, también podemos tener un apuntador **propietario** en cada miembro. Si al conjunto de miembros se accesa principalmente usando FIND NEXT, es suficiente el apuntador next. Si es FIND PRIOR, debe de incluirse un apuntador prior. Finalmente, si el propietario se accesa desde un miembro usando FIND OWNER, debe incluirse un apuntador propietario. La última opción es útil para registros que participan como miembros en varios conjuntos -por ejemplo, tipos de registros enlazados para relaciones M:N. con esta opción, un programa puede recuperar a los miembros usando un conjunto y luego encontrar directamente a su propietario usando al apuntador propietario del otro conjunto.

En una base de datos de red, el equivalente de la mayoría de las operaciones relacionales EQUIJOIN se *preespecifican* como conjuntos, como puede verse comparando el esquema de COMPAÑIA en el modelo relacional y el de red. Por ejemplo, la condición JOIN EMPLEADO.IMSS = TRABAJA_EN.IMSS en el esquema relacional se representa por el conjunto E_TRABAJEN. De la misma forma, la condición join PROYECTO.NUMERO = TRABAJA_EN.PNUMERO se representa por el conjunto P_TRABAJAEN. Por lo tanto, los joins que se ejecutan frecuentemente en el modelo relacional corresponden a conjuntos

que se trasversan en el modelo de red. Tales conjuntos son candidatos de implementación eficiente por colindancia física.

Desnormalización para Eficiencia o Argumentos Estructurales. Por las siguientes razones se puede desear replicar algunos de los campos de un *propietario* en el *miembro* de un conjunto:

- Si un campo del propietario se replica, puede usarse para especificar un argumento estructural de un conjunto MANUAL, o CONJUNTO DE SELECCION AUTOMATICA para un conjunto AUTOMATIC. Estos campos replicados pueden tener sus valores accesados desde el miembro directamente, *sin que el DBMS tenga que localizar y accesar al propietario*, reduciendo el tiempo de acceso, especialmente si no existe un apuntador propietario en el miembro.

- Otros, campos no llave también se replican en el miembro para eficiencia. Sin embargo, esta réplica significa que las actualizaciones a un campo en el propietario que se replica en los miembros debe propagarse a todos los miembros por el programa de actualización para preservar la consistencia, haciendo lentas las operaciones de actualización.

Opciones de acceso de registros. El modelo de red requiere de **puntos de entrada** a la base de datos para iniciar la búsqueda navegacional de registros. Estos se proporcionan por medio de los conjuntos propiedad del SISTEMA o especificando una estructura de acceso para un tipo de registro. El default es hacer una búsqueda lineal en un AREA -un concepto que establece una partición lógica de la base de datos que se asigna a un área del disco físicamente colindante. Otros **MODOS DE LOCALIZACION** de tipos de registros en el DBTG original, los cuales son seguidos por muchos DBMSs de red, incluyen a los siguientes:

- *CALC* - Los registros se seccionan es un campo específico de él llamado LLAVE CALC. Un registro se recupera directamente por el valor de su CALC KEY.

- *VIA CONJUNTO* - Este ocasiona que los miembros se almacenen cerca del propietario; no se dispone de acceso directo a los registros.

- *DIRECTO* - El programa de aplicación sugiere un página física actual o cerca de la cual debe almacenarse el registro. La dirección actual de almacenamiento (en forma de un registro apuntador) es retornada en un **DBKEY** (llave de la base de datos). El concepto de DBKEY fue sugerido en el reporte original DBTG como un mecanismo de eficiencia, pero fue olvidado en los últimos reportes.

La opción directa se usa cuando un programa guarda un apuntador al registro y posteriormente lo usa para cargar al registro directamente. El concepto de **AREA** permite que el diseñador especifique que los registros de cierto tipo se coloquen *físicamente cerca*

el uno del otro en disco, quizá en el mismo cilindro. La especificación de AREAS fue una decisión importante del diseño físico en sistemas que soportan este concepto.

Selección de Conjuntos Propiedad del SISTEMA y Ordenamiento de Registros. Los conjuntos propiedad del SISTEMA se definen para procesar todos los registros de un tipo en algun orden deseado, típicamente para usarse en aplicaciones de generación de reportes. Estos registros pueden ordenarse por valores de un campo especificado en la cláusula ORDER. Estos conjuntos deben especificarse si intentamos usar registros de un tipo como "puntos de entrada" y después localizarlos por otros conjuntos. Por ejemplo, si frecuentemente hacemos reportes que impriman información de departamentos y desplieguen información de los proyectos de esos departamentos, podemos usar un conjunto propiedad del SISTEMA para accesar a los DEPARTAMENTOS. Después podemos recuperar a los registros relacionados EMPLEADO y PROYECTO vía los conjuntos TRABAJA_PARA, MANEJA, y CONTROLA. Si con frecuencia queremos que los departamentos se accesen en orden del número de departamento, podemos ordenar al conjunto propiedad del SISTEMA por el campo DNO.

10.3.4 Diseño Físico de Bases de Datos para Sistemas Jerárquicos

Las principales decisiones para el diseño físico de sistemas de bases de datos jerárquicos están muy interrelacionados con el diseño lógico debido a muchas de las opciones que hemos definido para especificar jerarquías del mismo esquema conceptual. Sin embargo, existen muchas opciones adicionales, tales como la elección de campos de seccionado o indexado para el registro raíz o la elección de indexado "secundario" para registros no raíz e implementación de relaciones virtuales padre-hijo. El acceso a los datos en una base de datos jerárquica se argumenta por la estructura jerárquica y típicamente procede localizando primero al registro raíz. Con un árbol de ocurrencia, la búsqueda se conduce ya sea secuencialmente sobre los registros en el árbol o siguiendo ciertos esquemas de apuntadores. Los registros raíz pueden indexar o seccionar el acceso sobre ciertos campos para localizar eficientemente al árbol de ocurrencia requerido. Se habrá de enfrentar las siguientes decisiones que afectan el desmpeño de la base de datos:

1. Elección de registros raíz y jerarquías: Los registros raíz son puntos de entrada a la base de datos, debido a que todos los descendientes pueden accesarse de la raíz. Además, pueden especificarse fácilmente sobre la raíz trayectorias de acceso tales como seccionado e índices. Los registros que se usan con frecuencia para iniciar una recuperación son buenos candidatos para elegirse como raíces de las jerarquías.

2. Opciones de implementación para relaciones padre-hijo(PCR): La implementación más común de un PCR es un archivo jerárquico, el cual usa colindancia física y almacena los registros como una secuencia jerárquica (preorden transaversal). Sin embrago, pueden agregarse apuntadores que faciliten la localización de registros descendientes de un cierto tipo de registros (llamados *índices secundarios* en IMS). De manera similar, también pueden agregarse apuntadores que faciliten la localización de un registro ancestro (llamados *apuntadores físicos padres* en IMS).

3. Elección de apuntadores a registros: La opción del apuntador minimiza la redundancia al costo de tener que definir una relación virtual padre-hijo (VPCR). Elegir entre esta opción y la opción de replicar registros en una jerarquía es una decisión importante del diseño. La última, minimiza la redundancia, mientras que la anterior proporciona una recuperación más rápida al costo de complicar tremendamente el proceso de actualización.

4. Diferentes opciones para implementar relaciones virtuales padre-hijo: La mayoría de los VPCRs se implementan por apuntadores en los registros hijo, lo cual facilita localizar al padre desde el hijo. Esto es similar al apuntador propietario del modelo de red. Para proporcionar acceso de un padre virtual a su primer hijo virtual, puede usarse un apuntador (llamado *apuntador lógico hijo* en IMS). El hijo entonces apunta al siguiente hijo virtual teniendo al mismo padre virtual (usando un *apuntador lógico gemelo*). Esto es similar a los apuntadores next de un conjunto en bases de datos de red.

5. Registros virtuales padre falsos: Debido a que los VPCRs no tienen nombres, es deseable tener un registro padre virtual en *al menos un VPCR*. Si el diseño lógico elegido tiene un registro que es padre virtual en varios VPCRs, podemos crear un registro falso que sea hijo real de ese registro para cada VPCR adicional. Cada registro falso es ahora un padre virtual en un solo VPCR.

Por razones de eficiencia las bases jerárquicas también permiten el particionamiento de una jerarquía en grupos de registros. Esto es similar al concepto de AREA de los DBMSs de red. Un grupo, llamado **grupo de datos** en IMS, contiene un sub-árbol del esquema jerárquico y se mapea a un área física de almacenamiento. Esto mejora el acceso a registros dentro del sub-árbol almacenándolos en la proximidad. Una raíz de tal grupo puede usar acceso directo a través de un índice secundario.

10.4 Herramientas Automatizadas de Diseño

La mayoría de los diseños de bases de datos se siguen llevando a cabo manualmente por diseñadores expertos, quienes usan su experiencia y conocimiento en el proceso de diseño. Sin embargo, muchos aspectos del diseño son difíciles de llevar a cabo a mano y ameritan la automatización. Por ejemplo, es relativamente directo automatizar mucho de la fase de mapeo. Detectar conflictos entre los esquemas antes de integrarlos puede ser algo difícil de hacer a mano. De manera similar, evaluar diferentes alternativas cuantitativamente para el diseño físico de la base de datos puede ser muy consumidor de tiempo. Existen numerosas herramientas de diseño que ayudan con los aspectos iniciales del diseño de la base de datos, tal como el conceptual, el mapeo, y los aspectos físicos, así como con los requerimientos de colección y análisis. Sólo mencionaremos las características que una buena herramienta de diseño debe poseer:

- Una interface fácil de usar: Esto es crítico porque permite a los diseñadores enfocarse en la tarea y no en el entendimiento de la herramienta. Las interfaces gráficas y de

lenguaje natural son comúnmente usadas. Pueden personalizarse diferentes interfaces para usuarios finales o diseñadores expertos.

- Componentes analíticos: La mayoría de las herramientas proporcionan componentes analíticos para tareas que son difíciles de realizar a mano, tales como la evaluación física de alternativas de diseño o detección de argumentos en conflicto entre las vistas.

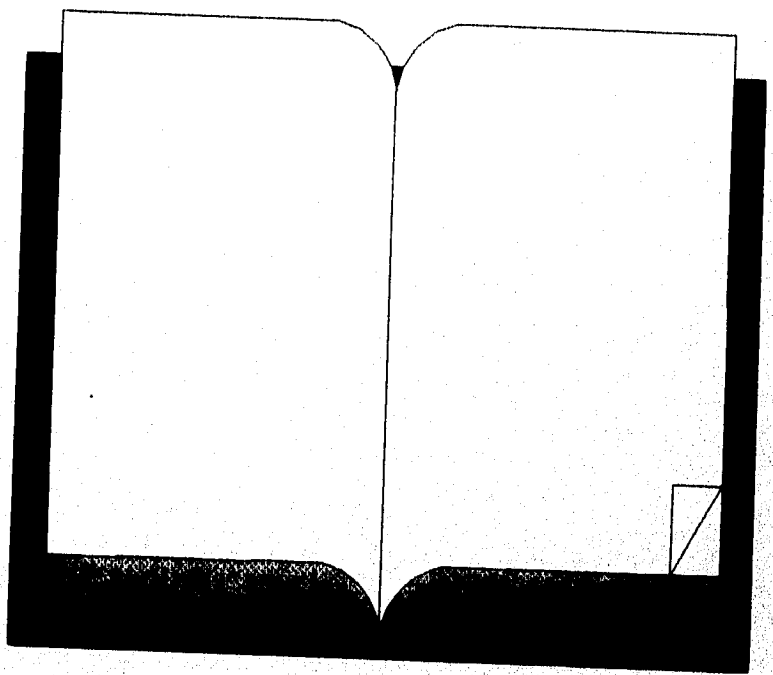
- Componentes heurísticos: Los aspectos del diseño que no pueden cuantificarse precisamente pueden automatizarse ingresando reglas heurísticas en la herramienta de diseño. Estas reglas se usan para evaluar alternativas de diseño heurísticamente.

- Análisis de cambio: Una herramienta debe presentar al diseñador el análisis comparativo adecuado cada que presente múltiples alternativas de donde elegir.

- Despliegue de los resultados del diseño: Los resultados del diseño, como los esquemas, se despliegan en forma diagramal. Pueden mostrarse otro tipo de resultados que se interpreten fácilmente como tablas, listas, o reportes.

Actualmente existe un crecimiento del valor de las herramientas de diseño, y se están volviendo una obligación para trabajar con los problemas de grandes bases de datos. El proceso de diseño se está volviendo inimaginable sin el soporte de herramientas adecuadas. También existe una creciente conciencia de que el diseño del esquema y de la aplicación van de la mano. Las herramientas CASE cubren ambas áreas. Algunas herramientas usan tecnología de sistemas expertos para guiar el proceso de diseño incluyendo la experiencia de diseño en forma de reglas. La tecnología de los sistemas expertos es también útil en la fase de colección y análisis de requerimientos, lo cual es típicamente un proceso laborioso y frustrante. La tendencia es usar diccionarios y herramientas de diseño para lograr mejores diseños para complejas bases de datos.

CAPITULO 11



Catálogo del Sistema

11.1 Catálogos para DBMSs Relacionales

La información almacenada en un catálogo de un DBMS relacional incluye descripciones de los nombres de la relación, de los atributos, dominios de atributos (tipos de datos), llaves primarias, atributos llave secundaria, llaves externas, y otro tipo de argumentos, así como descripciones de vistas y de nivel interno de las estructuras de almacenamiento e índices. También se incluye la información de seguridad y autorización, la cual especifica la autoridad de los usuarios para accesar a las relaciones de la base de datos y vistas y los creadores o propietarios de cada relación.

En DBMSs relacionales es común almacenar el catálogo como relaciones y usar el software del DBMS para hacer queries, actualizar, y mantener el catálogo. Esto permite que rutinas del DBMS (así como los usuarios) accesen la información almacenada en el catálogo siempre que estén autorizados para ello, usando el lenguaje query del DBMS.

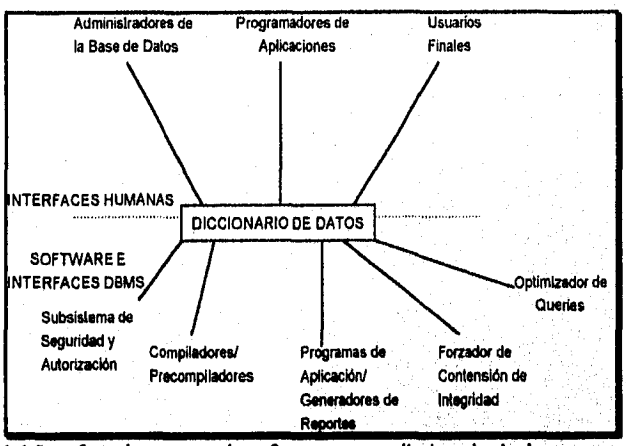


Figura 11.1 Interfaces humanas y de software para un diccionario de datos.

En la Figura 11.2 se muestra una posible estructura de catálogo para la información de la relación, la cual almacena nombres de las relaciones, de atributos, tipos de atributos, e información de llaves primarias. La Figura 11.2 también muestra como pueden incluirse los argumentos de llaves externas. La llave primaria de CATALOGO_ATR_REL es la combinación de atributos {NOMBRE_REL, NOMBRE_ATR}, porque todos los nombres de la relación deben ser únicos y todos los nombres dentro de una relación particular deben también ser únicos. Otro archivo de catálogo puede almacenar información de cada relación, tal como el tamaño de la tupla, número de tuplas, número de índices, y nombre del creador.

Para incluir información sobre atributos llave secundarios de una relación, podemos extender simplemente el catálogo anterior si asumimos que un atributo puede ser un *miembro de una llave única*. En esta caso podemos reemplazar el atributo

MIEMBRO_DE_PK del CATALOGO_ATR_REL con un atributo NUMERO_LLAVE; el valor de NUMERO_LLAVE es 0 si el atributo no es miembro de ninguna llave, 1 si es miembro de la llave primaria, e i > 1 para la iésima llave secundaria, donde las llaves secundarias de una relación se numeran 2, 3, ..., n. Sin embargo, si un atributo puede ser miembro de *más de una llave*, lo cual es un caso general, la representación anterior no es suficiente. Una posibilidad es almacenar la información de los atributos llave en un segundo catálogo RELACION_LLAVES, con atributos {NOMBRE_REL, NUMERO_LLAVE, ATR_MIEMBRO} que en conjunto son llave de RELACION_LLAVES. Esto se muestra en la Figura 11.3(a). El compilador DDL asigna el valor de 1 a NUMERO_LLAVE para la llave primaria y valores de 2, 3, ..., n para las llaves secundarias. Cada llave tendrá una tupla en RELACION_LLAVES para cada atributo que sea parte de esa llave, y el valor de ATRIBUTO_MIEMBRO da el nombre del atributo. Para almacenar información que involucre llaves externas puede usarse una estructura similar.

CATALOGO_ATR_REL

NOMBRE_REL	NOMBRE_ATR	TIPO_ATR	MIEMBRO_PK	MIEMBRO_FK	RELACION_FK
EMPLEADO	NOMBRE	VSTR15	no	no	
EMPLEADO	INIC	CHAR	no	no	
EMPLEADO	APELLIDO	VSTR15	no	no	
EMPLEADO	IMSS	STR9	si	no	
EMPLEADO	FECHANAC	STR9	no	no	
EMPLEADO	DIRECCION	VSTR30	no	no	
EMPLEADO	SEXO	CHAR	no	no	
EMPLEADO	SALARIO	INTGER	no	no	
EMPLEADO	SUPERIMSS	STR9	no	si	
EMPLEADO	DNO	INTEGER	no	si	EMPLEADO
DEPTO	DNOMBRE	VSTR	no	no	DEPTO
DEPTO	DNUMERO	INTEGER	si	no	
DEPTO	GTEIMSS	STR9	no	si	
DEPTO	FECINIGTE	STR10	no	no	EMPLEADO
DEP_LOC	DNUMERO	INTEGER	si	si	
DEP_LOC	DLOC	VSTR15	si	no	DEPTO
PROYECTO	PNOMBRE	VSTR10	no	no	
PROYECTO	PNUMERO	INTEGER	si	no	
PROYECTO	PLOC	VSTR15	no	no	
PROYECTO	ONO	INTEGER	no	si	
TRABAJA_EN	EIMSS	STR9	si	si	DEPTO
TRABAJA_EN	PNO	INTEGER	si	si	
TRABAJA_EN	HORAS	REAL	no	no	PROYECTO
DEPENDIENTE	EIMSS	STR9	si	si	
DEPENDIENTE	NOM_DEP	VSTR15	si	no	EMPLEADO
DEPENDIENTE	SEXO	CHAR	no	no	
DEPENDIENTE	FECHANAC	STR9	no	no	

Figura 11.2 Catálogo básico para el esquema relacional.

Después, consideraremos la información que tiene que ver con índices. En el caso general donde un atributo puede ser miembro de más de un índice, puede usarse el catálogo RELACION_INDICES de la Figura 11.3(b). La llave de RELACION_INDICES es la combinación de {NOMBRE_INDICE, MIEMBRO_ATR} (asumiendo que los nombres de los índices son únicos). MIEMBRO_ATR es el nombre de un atributo incluido en el índice. Por ejemplo, si especificamos tres índices en la relación TRABAJA_EN -un índice de agrupamiento sobre EIMSS, uno secundario sobre PNO, y otro secundario en la combinación de {EIMSS, PNO}- los atributos PNO y EIMSS son miebros de dos índices.

Los campos ATR_NO y ASC_DESC en INDICES especifican el orden de cada atributo en el índice y si está ordenado en forma ascendente o descendente.

Las definiciones de vistas también deben almacenarse en el catálogo. Una vista se espacifica mediante un query, con un posible renombramiento de los valores que aparecen en el resultado del query. Podemos usar las dos relaciones del catálogo de la Figura 11.3(c) para almacenar definiciones de vistas. La primera, VISTAS_QUERIES, tiene dos atributos {NOMBRE_VISTA, QUERY} y almacena el query (la cadena de texto completa) correspondiente a la vista. La segunda, VISTA_ATRIBUTOS, tiene atributos {NOMBRE_VISTA, ATR_NUM} para almacenar los nombres de los atributos de la vista; donde ATR_NUM es un número entero mayor que cero especificando la correspondencia de cada atributo de la vista con los atributos del resultado del query. La llave de VISTAS_QUERIES es NOMBRE_VISTA y la de VISTA_ATR es la combinación de {NOMBRE_VISTA, NOMBRE_ATR}.

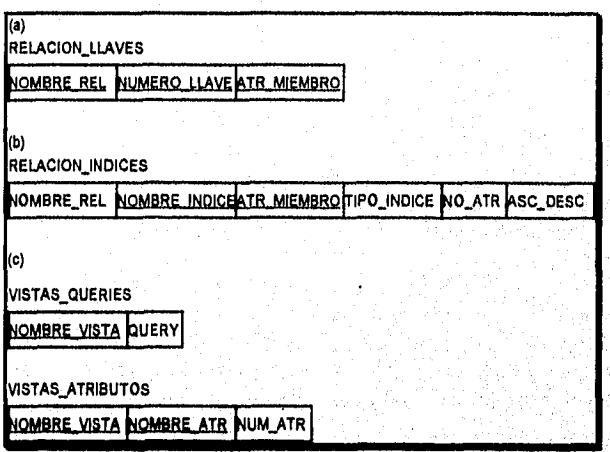


Figura 11.3 Otras posibles relaciones de catálogo para un sistema relacional. (a) Relación de catálogo para almacenar información general de llaves. (b) Relación de catálogo para almacenar información de índices. (c) Relaciones de catálogo para almacenar información de vistas.

El ejemplo anterior ilustra el tipo de información almacenada en un catálogo, la cual normalmente incluye mucho más archivos e información. La mayoría de los sistemas relacionales almacenan sus archivos de catálogos como relaciones DBMS. Sin embrago, debido a que el catálogo se accesa con mucha frecuencia por los módulos del DBMS, es importante implementar un acceso lo más eficiente posible. Puede ser más eficiente implementar el catálogo usando un conjunto especializado de estructuras de datos y rutinas de acceso.

11.2 Otra Información del Catálogo Accesada por los Módulos del DBMS

Los módulos del DBMS usan y accesan al catálogo con mucha frecuencia; por lo que es importante implementar el acceso al catálogo tan eficientemente como sea posible. En esta sección discutiremos las diferentes formas en las cuales los módulos del DBMS usan y accesan al catálogo. Estas incluyen a las siguientes:

1. Compiladores DDL (y SDL): Estos módulos procesan y checan la especificación de un esquema en el lenguaje de definición de datos y almacenan esa descripción en el catálogo. Se extraen del DDL y SDL todos los constructores y argumentos a todos los niveles -conceptual, interno y externo- si es necesario, se ingresan especificaciones al catálogo, como en cualquier información de mapeo entre niveles. Por lo tanto, estos módulos de software *populan* la minibase de datos del catálogo (o **meta-base de datos**) con datos, siendo los datos descripciones de los esquemas.

2. Parseador y verificador de query y DML: Estos módulos parsean queries, instrucciones de recuperación y actualización DML; y checan al catálogo para verificar si todos los nombres de los esquemas referenciados en estas instrucciones son válidos. Por ejemplo, en un sistema relacional, un parseador de query checaría que todos los nombres de las relaciones especificados en el query existen en el catálogo y que los atributos especificados pertenecen a las relaciones apropiadas. De manera similar, en un sistema de red, cualquier tipo de registro o conjunto referenciado en los comandos DML se extraen del catálogo, y se verifican los comandos DML.

3. Compilador de Queries y DML: Estos compiladores convierten queries de alto nivel y comandos DML a comandos de acceso a archivos de bajo nivel. Durante este proceso el mapeo entre el esquema conceptual y las estructuras de archivo se accesan desde el catálogo. Por ejemplo, el catálogo debe incluir una descripción de cada archivo y sus campos y correspondencias entre campos y atributos a nivel conceptual.

4. Optimizador de query y DML: El optimizador de queries accesa al catálogo para trayectorias de acceso e información de implementación, para determinar la mejor manera de ejecutar un query o comando DML. Por ejemplo, el optimizador accesa al catálogo para checar cuáles campos de una relación tienen accesos o índices, antes de decidir cómo ejecutar una selección o join en la relación. De manera similar, un comando DML de procesamiento en conjunto en una base de datos de red tales como FIND OWNER checan al catálogo para determinar si existe un apuntador OWNER para el miembro o si debe trazar a través de los apuntadores NEXT hasta alcanzar al propietario.

5. Chequeo de autoridad y seguridad: El DBA tiene comandos privilegiados para actualizar la autorización y seguridad de una parte del catálogo. Se checan todos los accesos por un usuario a una relación o registro para la autorización propia de acceso al catálogo.

6. Mapeo externo a conceptual de queries y DML: Los comandos queries y DML especificados con referencia a una vista externa o esquema debe transformarse para referir al esquema conceptual antes de que puedan procesarse por el DBMS. Esto se logra accesando la descripción de la vista del catálogo para llevar a cabo la transformación.

La información almacenada en el catálogo se accesa muy frecuentemente por prácticamente todos los módulos del DBMS. DBMSs más sofisticados necesitan almacenar información adicional en el catálogo, tales como el número de registros en cada relación o el tipo, el promedio de selectividad de diferentes campos en base a una relación, el número de niveles en un índice, y el promedio de miembros en una instancia de un conjunto. Esta información debe actualizarse automáticamente por el DBMS. Además, un diccionario catálogo/datos expandido debe incluir información que sea útil para los usuarios de la base de datos, tales como decisiones de diseño y justificaciones. Claramente, el catálogo es un componente muy importante de cualquier DBMS generalizado.

CAPITULO 12

Conceptos de Procesamiento de Transacciones

12.1 Introducción al Procesamiento de Transacciones

En esta sección introduciremos informalmente los conceptos de ejecución concurrente y recuperación de fallas de transacciones.

12.1.1 Sistemas Monousuario Versus Sistemas Multiusuario

Un criterio de clasificación de un sistema de bases de datos es por el número de usuarios que pueden usar al sistema *concurrentemente* -esto es, al mismo tiempo. Un DBMS es **monousuario** si al menos un usuario puede usar el sistema a la vez, y es **multiusuario** si muchos usuarios pueden usar al sistema concurrentemente. Los DBMSs monousuario están restringidos a algunos sistemas de microcomputadoras; la mayoría de los demás DBMSs son multiusuario. Por ejemplo, un sistema de reservaciones aéreas es usado concurrentemente por cientos de agentes de viajes y personal. Sistemas bancarios, agencias de seguros, y compañías similares son también operadas por muchos usuarios quienes submiten transacciones concurrentemente al sistema.

Debido al concepto de **multiprogramación**, el cual permite que la computadora procese diferentes programas (o transacciones) al mismo tiempo, varios usuarios pueden usar sistemas de cómputo simultáneamente. Si existe un solo CPU, puede procesar al menos un programa a la vez. Sin embargo, los **sistemas operativos multiprogramación** ejecutan algunos comandos de un programa, luego los suspenden y ejecutan comandos del siguiente programa, y así sucesivamente. Un programa es continuado en el punto donde fue suspendido cuando le toca su turno de usar el CPU otra vez. Por lo tanto, la ejecución concurrente de programas es **intercalada**, lo cual muestra dos programas A y B ejecutando concurrentemente. El intercalamiento mantiene al CPU ocupado cuando un programa en ejecución requiere de una entrada o salida (I/O), tal como leer un bloque del disco. El CPU se switchea para ejecutar otro programa en lugar de permanecer inactivo durante el tiempo de I/O.

Si el sistema de cómputo cuenta con varios procesadores, es posible el procesamiento **simultáneo** de varios programas, llevando a una concurrencia simultánea en lugar de intercalada, como se ilustra en la Figura 10.1 por los programas C y D. La mayor parte de la teoría que tiene que ver con el control de concurrencia en bases de datos se desarrolla en términos de concurrencia intercalada, a pesar de que puede adaptarse a concurrencia simultánea. En lo que resta del capítulo, asumiremos el *modelo intercalado de concurrencia*.

En un DBMS multiusuario, los datos almacenados en las fuentes primarias pueden accesarse concurrentemente por los usuarios, los cuales recuperan y modifican constantemente información de la base de datos. Se le llama **transacción** a la *ejecución de un programa* que accesa o cambia los contenidos de la base de datos.

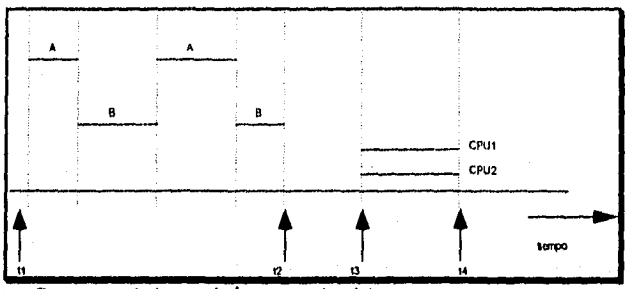


Figura 12.1 Concurrencia intercalada versus simultánea.

12.1.2 Operaciones de Lectura yEscritura de una Transacción

Con el propósito de discutir técnicas de control y recuperación trabajamos con transacciones a nivel de elementos de datos y bloques de disco. A este nivel, las operaciones de acceso a la base de datos que una transacción puede incluir son:

- **leer_elemento(X)**: Leer un elemento de la base de datos llamado X en una variable de programa. Para simplificar nuestra notación, suponemos que a *la variable de programa también se le llama X.*
- **escribir_elemento(X)**: Escribe el valor de la variable X en un elemento de la base de datos llamado X.

La unidad básica de transferencia de datos del disco a la memoria de la computadora es un bloque. En general, un elemento será el campo de algun registro en la base de datos, a pesar de que pueda ser una unidad mayor como el registro o aún un bloque completo. La ejecución del comando leer_elemento(X) incluye los siguientes pasos:

1. Encontrar la dirección en el disco que contenga al elemento X.
2. Copiar el bloque del disco a un buffer en memoria principal (si es que todavía no está en el buffer).
3. Copiar el elemento X del buffer a la variable llamada X.

La ejecución del comando escribir_elemento(X) incluye los siguientes pasos:

1. Encontrar la dirección en el disco que contenga al elemento X.
2. Copiar el bloque del disco a un buffer en memoria principal (si es que todavía no está en el buffer).
3. Copiar al elemento X la variable llamada X a su localidad correcta en el buffer.

4. Almacenar el bloque actualizado del buffer de regreso al disco (ya sea inmediata o posteriormente).

El paso 4 es el que se encarga de actualizar el disco. En algunos casos el buffer no se almacena inmediatamente en el disco, en caso de que se hagan cambios adicionales al buffer. Usualmente, la decisión de cuando almacenar un bloque modificado que está en memoria principal se maneja por el administrador de recuperación o el sistema operativo.

```
(a)   leer_elemento(X);
      X:=X-N;
      escribir_elemento(X);
      leer_elemento(Y);
      Y:=Y+N;
      escribir_elemento(Y);

(b)   leer_elemento(X);
      X:=X+M;
      escribir_elemento(X);
```

Figura 12.2 Dos transacciones sencillas. (a) Transacción T1. (b) Transacción T2.

Para accesar a la base de datos una transacción incluirá operaciones lee_elemento y escribe_elemento. La Figura 12.2 muestra ejemplos de dos transacciones sencillas. El control de concurrencia y los mecanismos de recuperación están principalmente ocupados de los comandos de acceso en una transacción.

Pueden ejecutarse concurrentemente transacciones submitidas por varios usuarios y accesar y actualizar los mismos elementos de la base de datos. Si esta ejecución concurrente no está controlada, puede llevar a problemas tales como la *inconsistencia de la base de datos*.

12.1.3 Por Qué es Necesario el Control de Concurrencia

Cuando se ejecutan transacciones de manera descontrolada pueden ocurrir muchos problemas. Ilustraremos algunos de estos problemas refiriendo a una base de datos de reservaciones aéreas en la cual se almacena un registro para cada vuelo. Cada registro incluye entre otra información el número de asientos reservados en ese vuelo como un *elemento de datos nombrado*. La Figura 12.2(a) muestra una transacción T1 que *cancela* N reservaciones de un vuelo cuyo número de asientos reservados en otro vuelo se almacena en el elemento de la base de datos llamado X y *reserva* el mismo número de asientos en otro vuelo cuyo número de asientos reservados se almacena en el elemento de la base de datos llamado Y. La Figura 12.2(b) muestra una transacción más sencilla T2 que sólo *reserva* M asientos en el primer vuelo referenciado en la transacción T1.

Cuando se escribe un programa de base de datos, se tiene el número de vuelo, sus fechas, y el número de asientos como parámetros; por lo tanto, puede usarse el mismo programa para ejecutar muchas transacciones, cada una con diferentes vuelos y número de asientos. Para propósitos de control de concurrencia, una transacción es una *ejecución particular* de un programa sobre una fecha, vuelo y número de asientos en específico. En la Figura 12.2(a) y (b), las transacciones T1 y T2 son *ejecuciones específicas* de los programas que

refieren a los vuelos específicos cuyo número de asientos se almacenan en los elementos de datos X y Y. Ahora discutiremos los tipos de problemas que podemos encontrar con estas dos transacciones.

Problema de Pérdida de la Actualización. Ocurre cuando dos transacciones que accesan a los mismos elementos de la base de datos tienen sus operaciones intercaladas de forma que hacen incorrecto el valor de algún elemento. Supongamos que las transacciones T1 y T2 se submiten aproximadamente al mismo tiempo, y que sus operaciones se intercalan por el sistema operativo, como se muestra en la Figura 12.3(a); entonces el valor final de X es incorrecto, porque T2 lee el valor de X *antes* de que T1 lo cambie en la base de datos, por lo tanto, *se pierde el valor actualizado resultante de* T1. Por ejemplo, si X=80 al inicio (originalmente había 80 reservaciones en el vuelo), N = 5 (T1 cancela 5 asientos en el vuelo correspondiente a X y los reserva en el vuelo correspondiente a Y), y M = 4 (T2 reserva 4 asientos en X), el resultado final debería ser 79, pero en el calendario de la figura 12.3(a), X=84 porque *se perdió* la actualización que canceló 5 asientos.

Problema de Actualización Temporal (o Lectura Sucia). Ocurre cuando una transacción actualiza a un elemento de la base de datos y luego la transacción falla por alguna razón. El elemento actualizado es accesado por otra transacción antes de que cambie a su valor original. La Figura 12.3(b) muestra un ejemplo donde T1 actualiza al elemento X y luego falla antes de terminar, así el sistema debe cambiar a X a su valor original. Antes de ser así, la transacción T2 lee el valor "temporal" de X, el cual no se registrará permanentemente en la base de datos debido a la falla de T1. Al elemento del valor X que se lee por T2 se le llama *dato sucio*, debido a que ha sido creado por una transacción que no se ha completado aún; por lo tanto, a este problema también se le conoce como problema de *lectura sucia*.

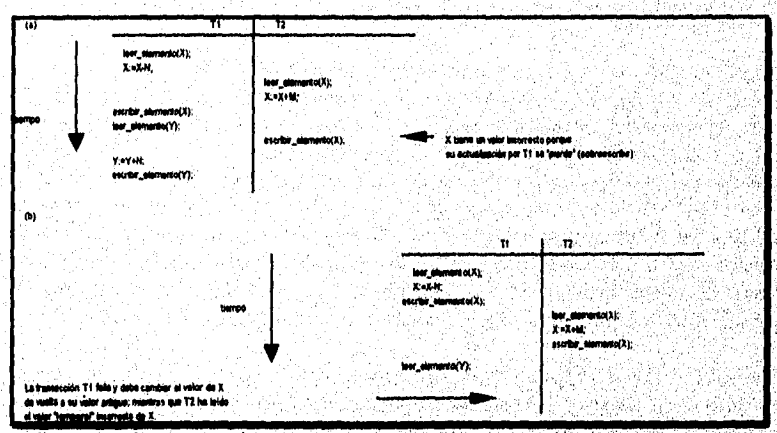


Figura 12.3 Algunos problemas que ocurren cundo la ejecución concurrente no tiene control. (a) El problema de pérdida de actualización. (b) El problema de actualización temporal.

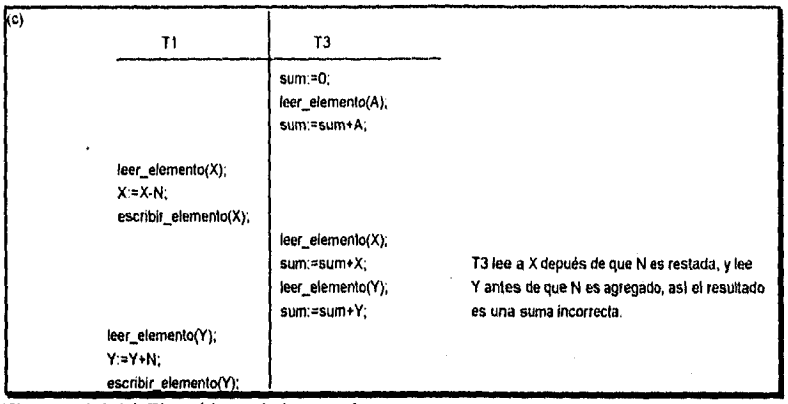


Figura 12.3 (c) El problema de la suma incorrecta.

El Problema de la Suma Incorrecta. Si una transacción calcula una suma sobre un número de registros mientras que otras transacciones actualizan a algunos de estos, la función de agregar puede calcular algunos valores antes de que se actualicen. Por ejemplo, supongamos que una transacción T3 calcula el total de reservaciones en todos los vuelos; mientras que se ejecuta la transacción T1. Si ocurre la operación intercalada en la Figura 12.3(c), el resultado de T3 será erróneo en una cantidad N porque T3 lee el valor de X *después* de que han sido sustraídos N asientos de ella pero lee el valor de Y *antes* de que esos N asientos hayan sido agregados a ella.

Otro problema que puede ocurrir es la **lectura irrepetible**, donde una transacción T lee un elemento dos veces, y entre las dos lecturas el elemento es cambiado por otra transacción T'. Por lo tanto, T recibe *diferentes valores* en sus dos lecturas del mismo elemento.

12.1.4 Por Qué es Necesaria la Recuperación

Cada vez que se transmite una transacción para su ejecución al DBMS, el sistema es responsable de asegurarse que (a) todas las operaciones de la transacción se completen exitosamente y su efecto se registre permanentemente en la base de datos, o (b) la transacción no tenga efecto en la base de datos o en cualquier otra transacción. El DBMS no debe permitir que se apliquen algunas operaciones de una transacción T a la base de datos mientras que otras operaciones de T no lo sean. Esto puede suceder si una transacción **falla** después de ejecutar algunas de sus operaciones pero antes de ejecutarlas a todas.

Tipos de Fallas. Existen varias razones para que una transacción falle a media ejecución:

1. Falla de la computadora (caída del sistema): Ocurre un error de hardware o software en el sistema de cómputo durante la ejecución de la transacción. Si el hardware falla, el contenido de la memoria de la computadora puede perderse.

2. Error de la transacción o del sistema: Alguna operación en la transacción puede causar que falle, tal como un overflow de entero o una división entre cero. La falla de transacción puede ocurrir también debido a valores erróneos de parámetros o a un error en la lógica del programa. Además, el usuario puede interrumpir a propósito la transacción durante su ejecución -por ejemplo, usando Ctrl-C en un sistema VAX/VMS o UNIX.

3. Errores locales o condiciones de excepción detectados por la transacción: Durante la ejecución de la transacción, pueden ocurrir ciertas condiciones que necesitan de la cancelación de la misma. Por ejemplo, los datos para la transacción no se encontraron. Una condición, tal como un balance de cuentas insuficiente en una base de datos bancaria, pueden ocasionar que sea cancelada una transacción tal como un retiro de una cuenta sin fondos. Esto puede llevarse a cabo por un ABORT programado en la transacción misma.

4. Reforzamiento del Control de Concurrencia: El método de control de concurrencia puede decidir abortar la transacción, para reestablecerla después, debido a que viola la seriabilidad o porque varias transacciones están en estado de bloqueo.

5. Falla en el disco: Algunos bloques de disco pueden perder sus datos debido a una malfunción de lectura o escritura o por la falla en una cabeza del disco. Esto puede suceder durante una operación de lectura o escritura de la transacción.

6. Problemas físicos y catástrofes: Esto se refiere a un sin fin de problemas que incluyen falla en la energía del aire acondicionado, fuego, robo, sabotaje, sobreescritura en discos o cintas por error, y montaje de una cinta equivocada por el operador.

Las fallas de tipos 1 al 4 son más comúnes que las del tipo 5 y 6. Cada que ocurre una falla del 1 al 4, el sistema debe guardar suficiente información para recuperarse de la falla. Las fallas en disco u otras fallas catastróficas del tipo 5 o 6 no suceden muy frecuentemente; si ocurren, es muy difícil recuperarse de ellas.

El concepto de transacción atómica es fundamental para muchas técnicas de control de concurrencia y recuperación de fallas. En la siguiente sección presentaremos el concepto de transacción y discutiremos por qué es tan importante.

12.2 Conceptos de Transacción y Sistema

A la *ejecución de un programa* que incluye operaciones de acceso a la base de datos se le llama **transacción de la base de datos** o simplemente **transacción**. Si las operaciones en una transacción no actualizan a ningun dato de la base sino que sólo recuperan datos, la transacción se llama **transacción de solo lectura**. En este capítulo nos interesaremos principalmente en las transacciones que hacen actualizaciones a la base, así la palabra *transacción* refiere a la ejecución de un programa que *actualiza a la base* a menos que se establezca lo contrario.

12.2.1 Estados de Transacción y Operaciones Adicionales

Una transacción es una unidad atómica de trabajo que se completa del todo o no se lleva a cabo. Para propósitos de recuperación, el sistema necesita registrar cuando inicia, termina, y finaliza o aborta la transacción. Por lo tanto, el administrador de recuperaciones registra las siguientes operaciones:

- INICIO_TRANSACCION: Marca el inicio de la ejecución de la transacción.
- LECTURA o ESCRITURA: Especifican operaciones de lectura o escritura sobre los elementos de la base de datos que se ejecutan como parte de la transacción.
- FIN_TRANSACCION: Especifica que las operaciones de lectura y escritura han finalizado. Sin embargo, en este punto puede ser necesario checar si los cambios introducidos por la transacción pueden aplicarse de manera permanente a la base de datos o si la transacción tiene que abortarse debido a que viola el control de concurrencia o por alguna otra razón.
- TRANSACCION_COMPLETA: Señala el *final exitoso* de la transacción de tal forma que cualquier cambio ejecutado por la transacción puede **completarse** de manera segura a la base de datos y no ser destruido.
- RETROCESO (o ABORTO): Señala que la transacción ha *finalizado sin éxito*, de tal forma que deben *deshacerse* los cambios o efectos que la transacción pudo haber aplicado a la base de datos.

Además de las operaciones anteriores, algunas técnicas de recuperación requieren operaciones adicionales que incluyen a las siguientes:

- UNDO: Similar a retroceso excepto que se aplica a una operación sencilla en lugar de a toda la transacción.
- REDO: Especifica que ciertas *operaciones de la transacción* deben rehacerse para asegurar que todas las operaciones de una transacción finalizada se han aplicado exitosamente a la base de datos.

La Figura 12.4 muestra un diagrama de transición de estados que describe cómo se mueve una transacción a través de sus estados de ejecución. Una transacción entra en un estado **activo** inmediatamente después de iniciar su ejecución, donde puede efectuar operaciones READ o WRITE. Cuando la transacción finaliza, se mueve al estado **parcialmente completado**. En este momento, algunas técnicas de control de concurrencia requieren que se hagan ciertos *chequeos* para asegurarse que la transacción no interfirió con otras transacciones en ejecución. Además, algunos protocolos de recuperación necesitan asegurarse de que no resulte una falla del sistema en una incapacidad de registrar los cambios de la transacción de manera permanenete. Una vez que ambos chequeos son

exitosos, se dice que la transacción ha alcanzado su punto de finalización y entra al **estado final**. Una vez que la transacción entra al estado final, ha concluido su ejecución de manera exitosa.

Sin embargo, una transacción puede ir al estado de **falla** si alguno de los chequeos falla o si se aborta durante su estado activo. La transacción puede entonces haberse abortado para deshacer el efecto de sus operaciones WRITE en la base de datos. El estado **terminado** corresponde a la transacción de dejar el sistema. Las transacciones fallidas o abortadas pueden *reestablecerse* después, ya sea automáticamente o después de haber sido resubmitida, como transacciones *nuevas*.

12.2.2 Bitácora del Sistema

Para poder recuperarse de fallas en el sistema, se mantiene una **bitácora** (algunas veces llamado **itinerario**). La bitácora registra todas las transacciones que afectan a los valores de elementos de la base de datos. Esta información puede ser necesaria para permitir la recuperación de fallas en la transacción. La bitácora se guarda en disco, de tal manera que no se afecta por ningun tipo de falla más que por una falla catastrófica. Además, la bitácora se respaldada periódicamente en cinta para mantenerse en caso de tales fallas catastróficas. Ahora listaremos los tipos de entradas que se escriben en la bitácora y la acción que cada una realiza. En estas entradas, T refiere a un **id-transacción** único que se genera automáticamente por el sistema y que se usa para identificar cada transacción:

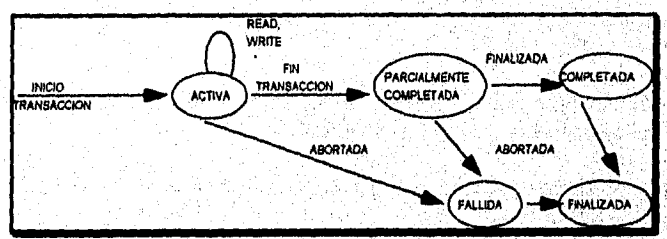


Figura 12.4 Diagrama de transición de estados para ejecución de transacciones.

1. [inicio_transacción, T]: Registra que la transacción T ha iniciado su ejecución.

2. [escribir_elemento, T, X, valor_anterior, valor_nuevo]: Registra que la transacción T ha cambiado el valor de algun elemento de la base de datos X de valor_anterior a valor_nuevo.

3. [leer_elemento, T, X]: Registra que la transacción T ha leído el valor del elemento X.

4. [finalizar, T]: Registra que la transacción T ha finalizado exitosamente, y afirma que su efecto pudo registrarse permanentemente en la base de datos.

5. [abortar, T]: Registra que la transacción T ha sido abortada.

Los protocolos de recuperación que evitan cancelaciones en casacada no requieren que las operaciones READ se escriban en la bitácora, mientras que otros protocolos si. En el último caso, la desventaja de registrar operaciones en la bitácora es reducido, debido a que sólo se registran las operaciones WRITE. Además, los protocolos estrictos requieren entradas WRITE más sencillas que no incluyen valor_nuevo.

Aquí suponemos que las transacciones no pueden anidarse. Que todos los cambios permanentes a la base de datos ocurran dentro de las transacciones, así la noción de recuperación de una falla en la transacción se remonta a deshacer o rehacer las operaciones *recuperables* individualmente de la bitácora. Si el sistema falla, se puede recuperar un estado consistente de la base de datos examinando la bitácora y usando alguna técnica. Debido a que la bitácora tiene registrada cada operación WRITE que cambia el valor de alguno de los elementos de la base, es posible **deshacer** el efecto de estas operaciones trasando la bitácora de regreso y modificando todos los elementos a sus valores anteriores. Puede requerirse el rehacer las operaciones de una transacción si todas las actualizaciones se registran en la bitácora pero ocurre una falla antes de asegurarnos que todos los nuevos_valores han sido escritos de manera permanente en la base de datos.

12.2.3 Punto Final de una Transacción

Una transacción T alcanza su **punto final** cuando todas sus operaciones que accesan a la base de datos han sido ejecutadas exitosamente y el efecto de todas las operaciones en la base de datos han sido registradas en la bitácora. Además del punto final, una transacción se dice estar **finalizada**, y su efecto se asume *registrado permanentemente* en la base de datos. La transacción entonces escribe una entrada [fin,T] en la bitácora. Si ocurre una falla del sistema, se vuelve a buscar en la bitácora todas las transacciones T que se han escrito [inicia_transacción, T] a la bitácora pero que aún no han escrito [finaliza,T]; estas transacciones pueden *abortarse* para deshacer su efecto en la base de datos durante el proceso de recuperación. Las transacciones que han escrito su entrada de finalización en la bitácora también deben registrar todas sus operaciones WRITE; de otro modo no serían completadas, de tal forma que su efecto en la base de datos puede *rehacerse* a partir de las entradas a la bitácora.

Notemos que el archivo bitácora debe guardarse en el disco. Es común guardar un bloque del archivo bitácora en memoria hasta que se llene de entradas y luego regresarlo a disco una sola vez, en lugar de escribir en disco cada vez que se agrega una entrada. Al momento de una falla en el sistema, sólo se consideran las entradas que han sido *escritas al disco* en el proceso de recuperación porque el contenido de la memoria puede perderse. Por lo tanto, *antes* de que una transacción alcance su punto final, cualquier porción de bitácora que no haya sido escrita en disco deberá hacerlo ahora. A este proceso se le llama **escritura-forzada** de la bitácora antes de realizar una transacción.

12.2.4 Chequeos en la Bitácora del Sistema

Periódicamente se escribe un registro [checkpoint] en la bitácora al momento en que el sistema escribe a la base de datos en disco el efecto de todas las operaciones WRITE de transacciones finalizadas. Por lo tanto, en caso de que el sistema falle todas las transacciones que tienen sus entradas [fin,T] en bitácora antes de una entrada [checkpoint] no necesitan rehacer sus operaciones WRITE.

El administrador de recuperación de un DBMS debe decidir en qué intervalos tomar el chequeo. El intervalo puede medirse en tiempo -digamos, cada m minutos- o por el número t de transacciones finalizadas desde el último chequeo, donde los valores m y t son parámetros del sistema. Tomar un chequeo consiste en las siguientes acciones:

1. Suspender la ejecución de transacciones temporalmente.

2. Escritura forzada de todas las operaciones de actualización de transacciones finalizadas de memoria principal a buffers en disco.

3. Escribir un registro [checkpoint] a bitácora, y forzar su escritura al disco.

4. Suspender la ejecución de transacciones.

Un registro de chequeo puede incluir también información adicional, tal como una lista de ids de transacciones, y las localidades (direcciones) del primer y último registros en la bitácora de cada transacción activa. Esto puede facilitar deshacer las operaciones de transacciones en caso de que una transacción deba abortarse.

12.3 Propiedades Deseables de Transacciones

Las transacciones atómicas deben poseer varias propiedades. A estas con frecuencia se les llaman **propiedades ACID**, y deben reforzarse por los métodos de control de concurrencia y de recuperación del DBMS. Las propiedades ACID son las siguientes:

1. **Atomicidad**: Una transacción es una unidad atómica de procesamiento; ya sea que se lleve a cabo completamente o nada.

2. **Preservación de Consistencia**: Una correcta ejecución de la transacción debe llevar a la base de datos de un estado consistente a otro similar.

3. **Aislamiento**: Una transacción no debe hacer visibles sus actualizaciones a las demás transacciones hasta que haya finalizado; esta propiedad, cuando se forza estrictamente, resuelve el problema temporal de la actualización y hace innecesarios los abortos en cascada.

4. **Durabilidad o permanencia**: Una vez que la transacción cambia a la base de datos y los cambios han finalizado, estos cambios no deben perderse nunca debido a una falla subsecuente.

La propiedad de atomicidad requiere que se ejecute una transacción por completo. Es responsabilidad del método de recuperación asegurar la atomicidad. Si alguna transacción falla por alguna razón, el método de recuperación debe deshacer cualquier efecto de la transacción sobre la base de datos.

La propiedad de preservación de la consistencia se considera generalmente responsabilidad de los programadores quienes escriben las aplicaciones de módulos del DBMS que reforza los argumentos de integridad. Recordemos que un **estado de la base de datos** es una colección de todos los elementos de datos (valores) almacenados en la base en algun momento dado. Un **estado consistente** de la base de datos satisface los argumentos especificados en el esquema así como cualquier otro argumento que deba mantenerse en la base de datos. Un programa debe de escribirse de tal forma que garantice que, si la base de datos es consistente antes de ejecutar la transacción, continuará así después de la ejecución *completa* de la transacción, asumiendo que no ocurra *interferencia con otras transacciones.*

El aislamiento se reforza por el método de control de concurrencia. Han habido intentos de definir el *nivel* o *grado de aislamiento* de una transacción. Una transacción se dice tener grado de aislamiento cero si no sobreescribe las lecturas sucias de transacciones de más alto nivel. Una transacción con grado de aislamiento uno no pierde actualizaciones; la de grado 2 no pierde actualizaciones ni lecturas sucias. Finalmente, el aislamiento de grado 3 (también llamado *aislamiento verdadero*) tiene, además de las propiedades del grado 2, lecturas repetibles. La propiedad de durabilidad es la responsable del método de recuperación.

12.4 Calendarios y Recuperabilidad

Cuando las transacciones se ejecutan concurrentemente de manera intercalada, el orden de ejecución de las operaciones de las distintas transacciones forma lo que se conoce como **calendario** de transacción (o **histórico**).

12.4.1 Calendarios de Transacciones

Un **calendario** S de n transacciones T1, T2, ..., Tn es un ordenamiento de las operaciones de las transacciones sujeto al argumento que, para cada transacción Ti que participe en S, las operaciones de Ti en S deben aparecer en el mismo orden en el cual ocurrieron en Ti. Sin embargo, las operaciones de las demás transacciones Tj pueden intercalarse con las operaciones de Ti en S. Por ahora, consideremos el orden de operaciones en S como un *ordenamiento total*, a pesar de que es posible tratar con calendarios cuyas operaciones forman *ordenes parciales.*

La Figura 12.3(a) y (b) muestra dos posibles calendarios de las transacciones T1 y T2. Para el propósito de recuperación y control de concurrencia, estamos principalmente interesados en las operaciones leer_elemento y escribir_elemento de las transacciones, así como las operaciones de finalizar y abortar. Una notación para describir un calendario usa los símbolos r, w, c y a para las operaciones de leer_elemento, escribir_elemento, finalizar y abortar, respectivamente, y se agrega como subíndice a cada operación en el calendario el id de la transacción. En esta notación, el elemento que se lee o escribe sigue a las operaciones r y w entre paréntesis. Por ejemplo, el calendario de la Figura 12.3(a), al cual llamaremos Sa, puede escribirse como sigue cuando aumentó con las operaciones de fin:

Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;

De manera similar, el calendario de la Figura 12.3(b), al cual llamaremos Sb, puede escribirse como sigue, si asumimos que la transacción T1 abortó después de su operación leer_elemento(Y):

Sb: r1(X); w1(X); r2(X); w2(X); c2; r1(Y); a1;

Dos operaciones en un calendario se dicen estar en **conflicto** si pertenecen a diferentes transacciones, si accesan al mismo elemento X, y si una de las dos operaciones es escribir_elemento(X). Por ejemplo, en el calendario Sa, las operaciones en conflicto r1(X) y w2(X), como las operaciones r2(X) y w1(X), y las operaciones w1(X) y w2(X). Sin embargo, las operaciones r1(X) y r2(X) no están en conflicto, ya que ambas son operaciones de lectura; y las operaciones w2(X) y w1(Y) no están en conflicto, debido a que operan sobre diferentes elementos de datos X y Y.

Un calendario S de n transacciones T1, T2, ..., Tn se dice ser un **calendario completo** si se cumplen las siguientes condiciones:

1. Las operaciones en S son exactamente las mismas en T1, T2, ..., Tn, incluyendo una operación de fin o aborto como la última de cada transacción en el calendario.

2. Para cualquier par de operaciones de la misma transacción Ti, su orden de aparición en S es el mismo que el orden de aparición en Ti.

3. Para cualesquiera dos operaciones en conflicto, una de las dos debe ocurrir antes que la otra en el calendario.

Las condiciones anteriores permiten que dos *operaciones sin conflicto* ocurran en el calendario sin definir cuál ocurre primero, llevando así a la definición de un calendario como **orden parcial** de las operaciones en las n transacciones. Sin embargo, para cualquier par de operaciones conflictivas debe especificarse un orden total en el calendario (condición 3) y para cualquier par de operaciones de la misma transacción (condición 2). La condición 1 simplemente establece que todas las operaciones deben aparecer completas

en el calendario. Debido a que cada transacción ya ha sido completada o abortada, un calendario completo no contendrá ninguna transacción activa.

En general, es difícil encontrar calendarios completos en un sistema de procesamiento de transacciones, debido a que nuevas transacciones están siendo submitidas continuamente al sistema. Por lo tanto, es útil definir el concepto de la **proyección de finalización** C(S) de un calendario S, el cual incluye sólo las operaciones en S que pertenecen a transacciones completadas -esto es, transacciones Ti cuya operación de finalización ci está en S.

12.4.2 Caracterización de Calendarios Basado en Recuperabilidad

Para algunos calendarios es fácil recuperarse de fallas en transacciones, mientras que para otros el proceso de recuperación puede ser algo difícil. Por lo tanto, es importante caracterizar los tipos de calendarios para los cuales es posible la recuperación, así como aquéllos para los cuales es relativamente fácil. Primero, debemos asegurarnos de que, una vez que finaliza una transacción T, *nunca debe ser necesario* abortar a T. A los calendarios que cumplen este criterio se les llama *calendarios recuperables*.

Un calendario S se dice ser **recuperable** si ninguna transacción T en S finaliza hasta que hayan finalizado todas las transacciones T' que hayan escrito un elemento que lee a T. Una transacción T se dice **leer de** la transacción T' en un calendario S si algun elemento X es escrito primero por T' y posteriormente leído por T. En el calendario S, T' no debe haber abortado antes de que T lea al elemento X, y no debe haber transacciones que escriban a X después que T' la escribe y antes de que T la lea (a menos que esas transacciones, si existen, hayan abortado antes de que T lea a X).

El calendario Sa dado en la sección precedente es recuperable. Sin embargo, consideremos los dos calendarios parciales Sc y Sd que siguen:

Sc: r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1;
Sd: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;

Sc no es recuperable, porque T2 lee al elemento X de T1, y luego T2 finaliza antes de que lo haga T1. Si T1 aborta después de la operación c2 en Sc, entonces el valor de X que lee T2 ya no es válido y T2 debe abortarse *después de que haya finalizado*, llevando a un calendario que no es recuperable. Para que el calendario sea recuperable, la operación c2 en Sc debe posponerse hasta después de que T1 finalice, como se mostró en Sd.

En un calendario recuperable, ninguna transacción finalizada necesita regresarse. Sin embargo, es posible que ocurra por un fenómeno conocido como *retroceso en cascada* (o *aborto en casacada*), donde las transacciones no finalizadas tienen que abortarse debido a que leen un elemento de una transacción fallida. Esto se ilustra en el calendario Se, donde la transacción T2 tiene que abortarse porque lee a X de T1, y T1 entonces aborta:

Se: r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1;

Debido a que el aborto en cascada puede ser consumidor de tiempo -numerosas transacciones pueden abortarse- es importante caracterizar los calendarios donde se garantice que este fenómeno no ocurra. Un calendario se dice que **evita el aborto en cascada** si cada transacción en el calendario sólo lee elementos que fueron escritos por transacciones *finalizadas*. En este caso, todos los elementos leídos estarán finalizados, así no ocurrirá ningún aborto en cascada. Para satisfacer este criterio, debe posponerse el comando r2(X) hasta después que T1 haya finalizado (o abortado), retrasando así a T2 pero nos asegura que no habrá aborto en cascada si T1 aborta.

Finalmente, existe un tercer tipo de calendarios más restrictivo, llamado **calendario estricto**, en el cual las transacciones no pueden leer ni escribir a un elemento X hasta que haya finalizado (o abortado) la última transacción que escribió a X. Los calendarios estrictos simplifican el proceso de recuperar operaciones de escritura a una cuestión de restaurar la **imágen anterior** de un elemento X, el cual es el valor que X tenía antes del aborto de la operación de escritura. Este simple procedimiento puede no trabajar siempre correctamente a menos que sea estricto el calendario. Por ejemplo, consideremos el calendario Sf:

Sf: w1(X,5); w2(X,8); a1;

Supongamos que el valor de X era originalmente el 9, el cual es la imágen anterior almacenada en la bitácora del sistema junto con la operación w1(X,5). Si T1 aborta, como en Sf, el procedimiento de recuperación que reestablece la imágen anterior de una operación de escritura abortada restaurará el valor de X a 9, aunque ya haya sido cambiada a 8 por la transacción T2, llevando así a resultados incorrectos. A pesar de que Sf evita los abortos en cascada, no es un calendario estricto, ya que permite que T2 escriba el elemento X aunque la transacción T1 que escribió a X al final no haya finalizado (o abortado). Un calendario estricto no tiene este problema; es recuperable y evita el aborto en cascada.

Hasta ahora hemos caracterizado a los calendarios en términos de su recuperabilidad. En la siguiente sección caracterizaremos al tipo de calendarios que se consideran correctos cuando se ejecutan transacciones concurrentes; a estos se les llama *calendarios serializables*.

12.5 Seriabilidad de Calendarios

Supongamos que dos usuarios -empleados de reservaciones aéreas- submiten al DBMS las transacciones T1 y T2 de la Figura 12.2 aproximadamente al mismo tiempo. Si no se permite el intercalado, existen solo dos formas de ordenar las operaciones de las dos transacciones para su ejecución:

1. Ejecutar todas las operaciones de la transacción T1 (en secuencia) seguida por todas las operaciones de la transacción T2 (en secuencia).

2. Ejecutar todas las operaciones de la transacción T2 (en secuencia) seguida por todas las operaciones de la transacción T1 (en secuencia).

Estas alternativas se muestran en la Figura 12.5(a) y (b), respectivamente. Si no se permiten operaciones de intercalado, habrá muchos ordenes posibles en los cuales el sistema pueda ejecutar las operaciones individuales de las transacciones. En la Figura 12.5(c) se muestran dos calendarios posibles.

Un aspecto importante del control de concurrencia, llamada **teoría de seriabilidad**, intenta determinar qué calendarios son "correctos" y cuales no lo son para desarrollar técnicas que permitan sólo calendarios correctos. Esta sección define los calendarios de seriabilidad, presenta algo de esta teoría, y discute cómo será utilizado en la práctica.

12.5.1 Calendarios Serializables, No Serializables y Conflictos de Calendarios Serializables

La Figura 12.5, muestra diferentes calendarios de las transacciones T1 y T2 de la Figura 12.2. A los calendarios A y B de la Figura 12.5(a) y (b) se les llama *calendarios serializables* porque las operaciones de cada transacción se ejecutan consecutivamente, sin ninguna operación de intercalado de la otra transacción. En un calendario serializable, se efectúan transacciones completas en forma serial: T1 y luego T2 en la Figura 12.5(a), y T2 y luego T1 en la Figura 12.5(b). A los calendarios C y D de la Figura 12.5(c) se les llama *no seriales* porque cada secuencia intercala operaciones de las dos transacciones.

Formalmente, un calendario S es **serializable**, si para cada transacción T participante en el calendario, todas las operaciones de T se ejecutan consecutivamente en el calendario; de otro modo, al calendario se le llama **no serializable**. Una suposición razonable que puede hacerse, es que si se considera que las transacciones son *independientes, cada calendario serializable se considera correcto.* Esto es así porque asumimos que cada transacción es correcta si se ejecuta por sí sola y las transacciones no dependen de las otras. Por lo tanto, no importa qué transacción se ejecute primero. Mientras que una transacción se ejecute de principio a fin sin ninguna interferencia de las operaciones de otras transacciones, obtendremos el resultado correcto en la base de datos. El problema con los calendarios serializables es que limitan las operaciones de concurrencia o intercalado. En un calendario serial, si una transacción espera a que una operación de I/O se complete, no podemos switchear al CPU otra transacción, desperdiciando así tiempo valioso de procesamiento y haciendo generalmente inaceptables a los calendarios serializables.

Para ilustrar nuestra discusión, consideremos los calendarios de la Figura 12.5, y asumamos que los valores iniciales son de X=90, Y=90, N=3 y M=2. Después de ejecutar las transacciones T1 y T2, esperaríamos que los valores en la base de datos fueran X=89 y Y=93, de acuerdo al significado de las transacciones. Por supuesto que la ejecución de

cualquiera de los calendarios A o B da los resultados correctos. Ahora consideremos los calendarios C y D no serializables. El calendario C (igual al de la Figura 12.3(a) da el resultado X=92 y Y=93, en el cual el valor de X es erróneo, mientras que el calendario D da los resultados correctos).

El calendaio C da un resultado erróneo por el problema de pérdida de la actualización; la transacción T2 lee al valor de X *antes* de que sea cambiado por la transacción T1, así sólo se refleja el efecto de T2 en X en la base de datos. El efecto de T1 en X *se pierde*, se sobreescribe por T2, llevando a un resultado incorrecto del valor X. Sin embargo, algunos calendarios no serializables dan el resultado correcto esperado, tal como el calendario D. Ahora debemos determinar cuál de los calendarios no serialozables dará *siempre* un resultado correcto y cuál puede dar un resultado erróneo. El concepto utilizado para caracterizar calendarios de esta forma es la seriabilidad de un calendario.

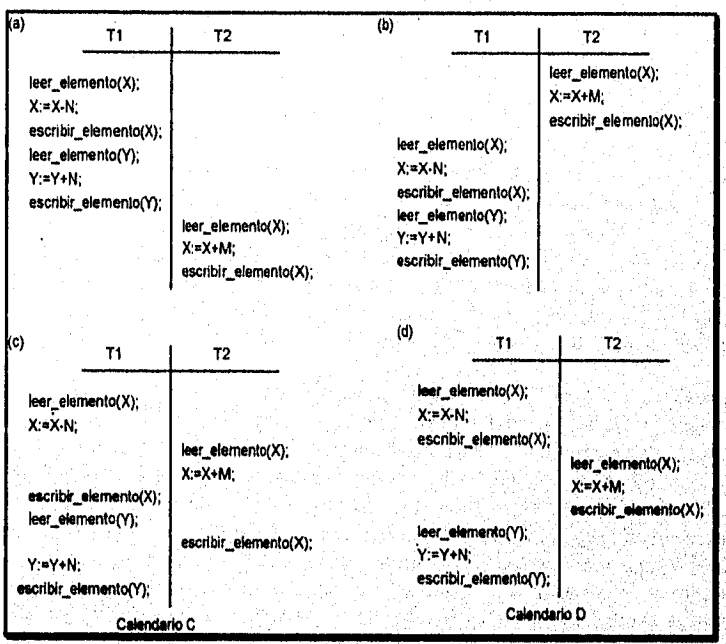


Figura 12.5 Algunos calendarios que involucran a las transacciones T1 y T2. (a) Calendario A: T1 seguida por T2. (b) Calendario B: T2 seguida por T1. (c) Dos calendarios con operaciones intercaladas.

Un calendario S de n transacciones es **serializable** si es *equivalente a algun calendario serializable* de las mismas n transacciones. Ahora definiremos brevemente el concepto de equivalencia de transacciones. Notemos que existen n! posibles calendarios serializables de n transacciones y mucho más posibles calendarios no serializables. Podemos formar dos grupos disjuntos de los calendarios no serializables: aquéllos que son equivalentes a uno (o

más) de los calendarios serializables, y por lo tanto serializables; y aquéllos que no son equivalentes a *ningún* calendario serializable, y por lo tanto no serializables.

Decir que un calendario no serializable S es serializable equivale a decir que es correcto, porque es equivalente a un calendario serializable, lo cual se considera correcto. La pregunta restante es: Cuándo se considera "equivalentes" a dos calendarios? Existen varias formas de definir la equivalencia de calendarios. La más sencilla, pero menos satisfactoria, involucra comparar los efectos de los calendarios en la base de datos. A dos calendarios se les llama **equivalentes de resultados** si producen el mismo estado final de la base de datos. Por ejemplo, en la Figura 12.6, los calendarios S1 y S2 producirán el mismo estado final si se jecutan sobre una base de datos con un valor inicial de X=100; pero para otros valores iniciales de X, los calendarios no son equivalentes. Además, estos dos calendarios ejecutan diferentes transacciones, de tal forma que definitivamente no deben considerarse equivalentes. Por lo tanto, la equivalencia de resultados no se usa para definir la equivalencia de calendarios.

El acercamiento más seguro y general de definir la equivalencia de un calendario es no hacer ninguna suposición acerca de los tipos de operaciones incluídos en las transacciones. Para que dos calendarios sean equivalentes, las operaciones aplicadas a cada elemento afectadas por los calendarios deben de aplicarse a ese elemento en ambos calendarios *en el mismo orden*. Generalmente se aceptan dos definiciones de equivalencia de calendarios: *equivalencia de conflictos* y *equivalencia de vistas*.

Dos calendarios dicen ser **equivalentes de conflictos** si el orden de cualesquiera dos operaciones *conflictivas* es el mismo en ambos calendarios. Dos operaciones en un calendario se dicen estar en *conflicto* si pertenecen a diferentes transacciones, si accesan al mismo elemento, y si una de las dos operaciones es escribir_elemento. Si dos operaciones en conflicto se aplican en *diferentes órdenes* a dos calendarios, el efecto puede ser diferente ya sea en las transacciones o en la base de datos, y por lo tanto no son equivalentes en conflicto. Por ejemplo, si ocurre una operación read y write en el orden r1(X), w2(X) en un calendario Sa, y en el orden inverso w2(X), r1(X) en un calendario Sb, el valor leído por la operación r1(X) puede ser diferente en los dos calendarios. De manera similar, si ocurren dos operaciones write en el orden w1(X), w2(X) en un calendario Sa, y en el orden inverso w2(X), w1(X) en el calendario Sb, la siguiente operación r(X) en los dos calendarios leerá potencialmente valores diferentes; o si estos son las últimas operaciones que accesan al elemento X en los calendarios, el valor final de X será diferente para los dos calendarios.

Usando la noción de equivalencia de conflictos, definimos que un calendario S está en **conflicto serializable** si es equivalente a algun calendario serializable S'. En tal caso, podemos reordenar las operaciones *sin conflicto* en S hasta formar el calendario serializable equivalente S'. De acuerdo a esta definición, el calendario D de la Figura 12.5(c) es equivalente al calendario serializable A de la Figura 12.5(a). En ambos calendarios, el leer_elemento(X) de T2 lee el valor de X escrito por T1, mientras que las otras operaciones leer_elemento leen los valores de la base de datos de su estado inicial.

Además, T1 es la última transacción para escribir Y, y T2 es la última transacción para escribir X. Debido a que A es un calendario serializable y D es equivalente a A, D es *calendario serializable*. Notemos que las operaciones r1(Y) y w1(Y) del calendario D no están en conflicto con las operaciones r2(X) y w2(X), ya que accesan diferentes elementos de datos. Por lo tanto, podemos mover r1(Y), w1(Y) antes de r2(X), w2(X), llevando al calendario serializable equivalente T1, T2.

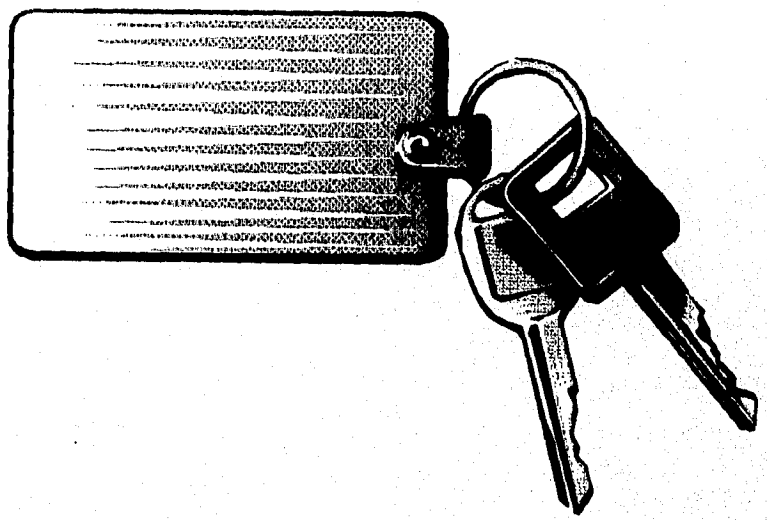
El calendario C de la Figura 12.5(c) es equivalente a cualquiera de los dos posibles calendarios serializables A y B. No es equivalente a A, porque la operación leer_elemento(X) en T2 lee el valor escrito por T1 en el calendario A pero lee el valor original de X en el calendario C, violando así la regla de equivalencia 1. Al mismo tiempo, el calendario C no es equivalente a B. Por lo tanto, el calendario C *no es serializable*. Si intentamos reordenar las operaciones del calendario C para encontrar un calendario serializable equivalente, fallaremos porque r2(X) y w1(X) están en conflicto, así que no podemos mover a r1(X) abajo para obtener el calendario serial equivalente T1, T2. De manera similar, debido a que w1(X) y w2(X) están en conflicto, no podemos mover w1(X) hacia abajo para obtener el calendario serial equivalente T2, T1.

12.5.2 Prueba de Conflicto de Seriabilidad en un Calendario

Existe un algoritmo sencillo para determinar el conflicto de seriabilidad de un calendario. La mayoría de los métodos de control de concurrencia no prueban la seriabilidad. En su lugar, se desarrollaron protocolos, o reglas que garanticen que un calendario será serializable.

El algoritmo 12.1 puede usarse para probar el conflicto de seriabilidad de un calendario. El algoritmo busca las operaciones leer_elemento y escribir_elemento en el calendario para construir una **gráfica de precedencia** (o **gráfica de serialización**). Una gráfica de precedencia es una **gráfica dirigida** G=(N,E) que consta de un conjunto de nodos N={T1, T2, ..., Tn} y un conjunto de puntas dirigidas E={e1, e2, ..., em}. En la gráfica existe un nodo para cada transacción Ti en el calendario. Cada punta ei es de la forma (Tj →Tk), 1≤j≤n, 1≤k≤n, donde a Tj se le llama el **nodo inicial** de ei y a Tk se le llama el **nodo final** de ei de tal forma que una de las operaciones en Tj aparece en el calendario antes de alguna *operación en conflicto* en Tk.

CAPITULO 13



Técnicas de Control de Concurrencia

13.1 Técnicas de Bloqueo para el Control de Concurrencia

Una de las principales técnicas usadas para controlar la ejecución concurrente de transacciones se basa en el concepto del bloqueo de elementos. Un **bloqueo** es una variable asociada con un elemento de datos de la base y describe el status de ese elemento con respecto a las posibles operaciones que pueden aplicarse a él. Generalmente, existe un bloqueo para cada elemento de la base de datos. Los bloqueos se usan como medio de sincronizar el acceso de transacciones concurrentes a los elementos de la base de datos.

13.1.1 Tipos de Bloqueos

En el control de concurrencia pueden usarse varios tipos de bloqueos. Primero presentaremos los bloqueos binarios, los cuales son sencillos pero algo restrictivos en su uso. Después discutiremos los bloqueos compartidos y exclusivos, los cuales proporcionan mayores capacidades de bloqueo.

Bloqueos Binarios. Un **bloqueo binario** puede tener dos **estados** o **valores**: bloqueado y desbloqueado (o 1 y 0). A *cada* elemento X de la base de datos se le asocia un bloqueo distinto. Si el valor del bloqueo en X es 1, el elemento X *no puede accesarse* por una operación que requiera de ese elemento. Si el valor del bloqueo en X es 0, el elemento puede accesarse cuando se requiera. Nos referiremos al *valor* del bloqueo asociado con el elemento X como **LOCK(X)**.

Cuando se usa el bloqueo binario deben incluirse dos operaciones, bloquear_elemento y desbloquear_elemento. Una transacción requiere acceso a un elemento X llamando a una operación **bloquear_elemento (X)**. Si LOCK(X) = 1, la transacción es forzada a esperar; de otro modo, la transacción pone a LOCK(X):=1 (bloquea el elemento) y se permite su acceso. Cuando la transacción está en proceso usando el elemento, libera una operación **desbloquear_elemento(X)**, la cual pone a LOCK(X):=0 (desbloquea el elemento) de tal forma que X puede accesarse por otras transacciones. En la Figura 13.1 se muestra una descripción de las operaciones bloquear_elemento(X) y desbloquear_elemento(X).

```
bloquear_elemento(X);
B: if LOCK(X) = 0
        then LOCK(X) ← 1
        else begin
          wait (until LOCK = 0 and
                  el administrador de bloqueos levante la transacción);
        go to B
        end;

desbloquear_elemento(X);
LOCK(X)← 0
        if cualquier transacción está en espera
        then levantar una de las transacciones en espera;
```

Figura 13.1 Operaciones de bloqueo y desbloqueo para bloqueos binarios.

Notemos que las operaciones bloquear_elemento(X) y desbloquear_elemento(X) deben implementarse como unidades indivisibles, esto es, una vez que una operación de bloqueo

o desbloqueo inició no debe permitirse el intercalamiento hasta que la operación termine o la transacción espera. En la Figura 13.1, el comando de espera dentro de la operación bloquear_elemento(X) se implementa usualmente poniendo a la transacción en una cola de espera para el elemento X hasta que se desbloquea y la transacción tenga acceso a ella. En la misma cola se colocan otras transacciones que también requieren de acceso a X. Por lo tanto, el comando de espera se considera fuera de la operación bloquear_elemento. Para registrar y controlar el acceso a bloqueos el DBMS tiene un subsistema **administrador de bloqueos**.

Cuando se usa un esquema de bloqueo binario, cada transacción debe obedecer las siguientes reglas:

1. Una transacción T debe liberar la operación bloquear_elemento(X) antes de realizar cualquier operación leer_elemento(X) o escribir_elemento(X) sobre T.

2. Una transacción T debe liberar una operación desbloquear_elemento(X) después de que se hayan realizado sobre T todas las operaciones leer_elemento(X) y escribir_elemento(X).

3. Una transacción T no liberará una operación bloquear_elemento(X) si ya tiene bloqueado a ese elemento.

4. Una transacción T no liberará una operación desbloquear_elemento(X) a menos que ya tenga bloqueado a ese elemento.

Entre las operaciones bloquear_elemento(X) y desbloquear_elemento(X) en la transacción T, se dice que T **retiene el bloqueo** sobre el elemento X. Ningun par de transacciones puede accesar al mismo elemento concurrentemente. Notemos que implementar un bloqueo binario es sencillo; todo lo que se necesita es una variable binaria, LOCK, asociada con cada elemento X de la base de datos. En su forma más sencilla, cada bloqueo puede ser un registro con dos campos <nombre del elemento, LOCK> más una cola para transacciones en espera. Estos registros seguardan en una **tabla de bloqueo**.

Bloqueos Compartidos y Exclusivos. En el **bloqueo de modo-múltiple** existen tres operaciones de bloqueo: bloqueo_lectura(X), bloqueo_escritura(X) y desbloquear(X). A un elemento de bloqueo de lectura también se le llama **bloqueo compartido**, debido a que se les pemite leer el elemento a otras transacciones, mientras que a un elemento de bloqueo de escritura se le llama **bloqueo exclusivo**, porque una sola transacción retiene al bloqueo sobre el elemento.

Un método sencillo, pero no muy general, de implementar las tres operaciones anteriores sobre un bloqueo de modo múltiple es registrar el número de transacciones que retienen un bloqueo compartido sobre un elemento. Cada bloqueo puede ser un registro con tres campos:

<nombre del elemento, LOCK, no_de_lecturas>. El valor de LOCK es cualquiera de bloqueo de lectura, bloqueo de escritura o desbloqueo, adecuadamente codificado. Una vez más, para ahorrar espacio, el sistema sólo registra a los elementos bloqueados. Como antes, cada una de las tres operaciones debe considerarse indivisible; no debe permitirse el intercalado una vez que una de las representaciones inicia hasta que cualquiera termina o la transacción se coloca en una cola de espera para el elemento.

```
leer_elemento(X):
B: if LOCK(X) = "desbloqueado"
  then begin LOCK (X) ← "lectura bloqueada"
            no_de_lecturas(X) ← 1
            end
  else if LOCK(X) "lectura bloqueada"
  then no_de_lecturas(X) ← no_de_lecturas(X) + 1
  else begin wait (until LOCK(X) ="desbloqueado" y
  el administrador de bloqueo levante la transacción);
            go to B
end;

escribir_elemento(X):
if LOCK(X) = "desbloqueado"
  then begin LOCK(X) ← "bloqueo escritura"
  else begin
    wait(until LOCK(X) = "desbloqueado" and
    el administrador de bloqueo levante la transacción);
            go to B
end;

desbloquear_elemento(X):
if LOCK(X) = "escritura bloqueada"
  then begin LOCK(X) ← "desbloqueado";
    levanta una de las transacciones en espera, si existen
  end
else if LOCK(X) = "lectura bloqueada"
  then begin
    no_de_lecturas(X) ← no_de_lecturas(X) - 1;
    if no_de_lecturas(X) = 0
      then begin LOCK(X) = "desbloqueado";
    levanta una de las transacciones en espera, si existen
  end
end;
```

Figura 13.2 Operaciones de bloqueo y desbloqueo para bloqueos de dos modos (lectura-escritura o compartido exclusivo).

Cuando usamos el esquema de bloqueo multimodo, el sistema debe reforzar las siguientes reglas:

1. Una transacción T debe asegurar la operación bloqueo_lectura(X) o bloqueo_escritura(X) antes de que se realice cualquier operación leer_elemento(X) en T.

2. Una transacción T debe asegurar la operación bloqueo_escritura(X) antes de que se relice en T cualquier operación escribir_elemento(X).

3. Una transacción debe asegurar que se lleve a cabo la operación desbloquear(X) después de que se completen en T todas las operaciones leer_elemento(X) y escribir_elemento(X).

4. Una transacción T no liberará una operación bloqueo_lectura(X) si ya retiene un bloqueo de lectura (compartido) o escritura (exclusivo) sobre el elemento X. Como veremos después, esta regla puede modificarse.

5. Una transacción T no liberará una operación bloqueo_escritura(X) si ya retiene un bloqueo de lectura (compartido) o escritura (exclusivo) sobre el elemento X.

6. Una transacción T no liberará una operación desbloquear(X) a menos que ya retenga un bloqueo de lectura (compartido) o escritura (exclusivo) sobre el elemento X.

13.1.2 Garantía de Seriabilidad Mediante Bloqueo de Dos Fases

Una transacción se dice seguir el **protocolo de bloqueo de dos fases** si *todas* las operaciones de bloqueo (bloqueo_lectura, bloqueo_escritura) preceden a la *primer* operación de desbloqueo en la transacción. Tal transacción puede dividirse en dos fases: una **fase de expansión** (o **crecimiento**), durante la cual pueden adquirirse pero no liberarse nuevos bloqueos sobre elementos; y una de **fase de contracción**, durante la cual existen bloqueos que pueden liberarse pero no adquirirse. Si se permite la actualización de bloqueos, esta definición no cambia. Sin embargo, si se permite el retroceso de bloqueos, la definición debe cambiarse ligeramente, debido a que *todos los retrocesos deben efectuarse en la fase de contracción*. Por lo tanto, una operación bloqueo_lectura(X) que retrocede a un bloqueo de escritura sobre X puede aparecer sólo en la fase de contracción de la transacción.

Las transacciones T1 y T2 de la Figura 13.3(a) no siguen el protocolo de bloqueo de dos fases. Esto se debe a que la operación bloqueo_escritura(X) sigue a la operación desbloquea(Y) en T1, de manera similar la operación bloqueo_escritura(Y) sigue a la operación desbloquea(X) en T2. Si reforzamos el bloqueo de dos fases, la transacción puede reescribirse como T1' y T2', como se muestra en la Figura 13.4. Ahora, el calendario mostrado en la Figura 13.3(c) no es permitido para T1' y T2' bajo las reglas de bloqueo descritas anteriormente. Esto se debe a que T1' liberará su bloqueo_escritura(X) *antes* de desbloquear al elemento Y; consecuentemente, cuando T2' libere bloqueo_lectura(X), se forza a esperar hasta que T1' libere su desbloqueo(X) en el calendario.

Puede probarse que, si cada transacción en un calendario sigue el protocolo de bloqueo de dos fases, se garantiza que el calendario es serializable. El mecanismo de bloqueo, reforzando las reglas de bloqueo de dos fases, también reforza la seriabilidad.

El bloqueo de dos fases puede limitar la cantidad de concurrencia que puede ocurrir en un calendario. Esto se debe a que una transacción T puede no ser capáz de liberar un elemento X después de que lo usa si T debe bloquear un elemento adicional Y posteriormente; o recíprocamente, T debe bloquear al elemento adicional Y antes de que lo necesite pudiendo así liberar a X. Por lo tanto, X debe permanecer bloqueado por T hasta que todos los elementos que necesita la transacción hayan sido bloqueados; sólo

entonces puede liberarse a X por T. Mientras tanto, la otra transacción que busca acceso a X puede forzarse a esperar, aún a pesar de que T siga usando a X; recíprocamente, si Y es bloqueado antes de ser necesario, otra transacción que busque acceso a Y se forza a esperar aunque T no esté siendo usado aún. Este es el precio de garantizar la seriabilidad de todos los calendarios sin tener que checarlos.

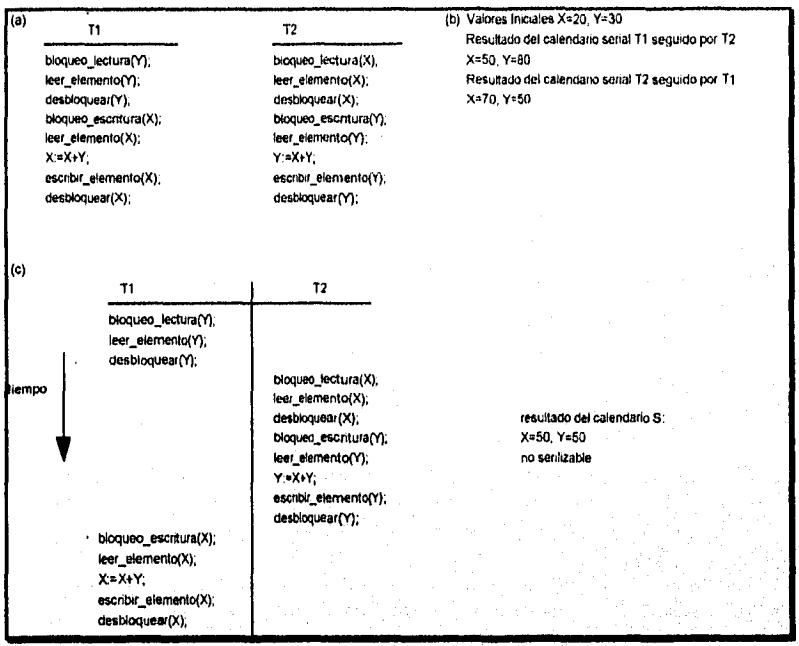


Figura 13.3 Transacciones que no obedecen el bloqueo de dos fases. (a) Dos transacciones T1 y T2. (b) Resultados de posibles calendarios seriales de T1 y T2. (c) Un calendario no serializable S que usa bloqueos.

T1	T2
bloqueo_lectura(Y);	bloqueo_lectura(X);
leer_elemento(Y);	leer_elemento(X);
bloqueo_escritura(X);	bloqueo_escritura(Y);
desbloqueo(Y);	desbloqueo(X);
leer_elemento(X);	leer_elemento(Y);
X:=X+Y;	Y:=X+Y;
escribir_elemento(X);	escribir_elemento(Y);
desbloquear(X);	desbloquear(Y);

Figura 13.4 Transacciones T1' y T2', las cuales son las mismas que T1 y T2 de la Figura 13.3 pero la cual sigue el protocolo de bloqueo de dos fases.

Bloqueo de Dos Fases, Básico, Conservador y Estricto. Existe un número de variantes del bloqueo de dos fases (2PL). La técnica recién descrita se conoce como **2PL básico**. Una variante conocida como **2PL conservador** (o **2PL estático**) requiere que una

transacción bloquee a todos los elementos que accesa *antes de que la transacción inicie su ejecución*, **predeclarando** su juego de lectura y escritura.

La lectura de una transacción es el conjunto de todos los elementos que leen en la transacción, y la escritura es el conjunto de todos los elementos que escriben en la transacción. Si cualquiera de los elementos necesarios predeclarados no puede bloquearse, la transacción no bloquea a ningún elemento; sino que, espera hasta que todos los elementos estén disponibles para el bloqueo. El 2PL conservador es un protocolo de bloqueo-libre.

En la práctica, la variante más popular de 2PL es el **2PL estricto**, el cual garantiza los calendarios estrictos. En esta variante, una transacción T no libera a ninguno de sus bloqueos hasta que finaliza o aborta. Por lo tanto, ninguna otra transacción puede leer o escribir un elemento que está escrito por T a menos que T finalice, llevando a un calendario estricto de recuperabilidad. Notemos la diferencia entre 2PL conservador y estricto; el primero debe bloquear a todos sus elementos *antes de que inicie*, mientras que el último no desbloquea a ninguno de sus elementos hasta *después de que finaliza* (por finalización o aborto). El 2PL estricto no es libre de bloqueos a menos que se combine con 2PL conservador.

13.2 Control de Concurrencia Basado en el Ordenamiento de Etiquetas

El uso de bloqueos con el protocolo de dos fases, nos permite garantizar la seriabilidad de los calendarios. El orden de las transacciones equivalente al calendario serial se basa en el orden en el cual las transacciones en ejecución bloquean a los elementos que requieren. Si una transacción necesita un elemento que ya está bloqueado, puede forzar a esperar hasta que se libere. Un acercamiento diferente que garantiza la seriabilidad involucra usar etiquetas para ordenar la ejecución de la transacción equivalente a un calendario serial.

13.2.1 Etiquetas

Una **etiqueta** es un identificador único creado por el DMBS para identificar una transacción. Típicamente, los valores de la etiqueta se asignan en el orden en el cual las transacciones se envían al sistema, así una etiqueta puede concebirse como *la hora de inicio de una transacción*. Nos referiremos a la etiqueta de una transacción T como **TS(T)**. Las técnicas de control de concurrencia basadas en etiquetas no usan bloqueos.

Las etiquetas pueden generarse de varias formas. Una posibilidad es usar un contador que se incremente cada vez que se asigne su valor a una transacción. Las etiquetas de la transacción se enumeran 1, 2, 3... en este esquema. Un contador tiene un valor máximo finito, así el sistema debe resetear periódicamente al contador a cero cuando ninguna transacción se ejecuta por algun periodo de tiempo. Otra forma de implementar las etiquetas es usar el valor actual del reloj del sistema y asegurarse que no se generen dos valores de etiqueta durante el mismo periodo del reloj.

13.2.2 Algoritmo de Ordenamiento de Etiquetas

La idea de este esquema es ordenar las transacciones basado en sus etiquetas. Un calendario en el cual las transacciones participan es serializable, y el calendario serial equivalente tiene a las transacciones en orden de sus etiquetas. A esto se le llama **ordenamiento por etiqueta** (TO). En el bloqueo de dos fases, un calendario es serializable siendo equivalente a *algun* calendario serial permitido por los protocolos de bloqueo; en el ordenamiento por etiqueta, el calendario es equivalente al orden serial *particular* que corresponde al orden de las etiquetas de transacciones. El algoritmo debe asegurarse de que, para cada elemento accesado por más de una transacción en el calendario, el orden en el cual el elemento se accesa no viola la seriabilidad del calendario. Para lograrlo, el algoritmo **básico TO** asocia a cada elemento X de la base de datos dos valores de etiqueta (TS):

1. **read_TS(X)**: **Etiqueta de lectura** del elemento X; esta es la etiqueta más grande entre todas las etiquetas de transacciones que han leído exitosamente a X.

2. **write_TS(X)**: **Etiqueta de escritura** del elemento X; es la etiqueta más grande de todas las etiquetas de transacciones que han escrito exitosamente al elemento X.

Cada vez que una transacción T intenta liberar una operación leer_elemento(X) o escribir_elemento(X), el algoritmo básico de TO compara la etiqueta de T con la etiqueta de lectura y escritura de X para asegurarse que no se viola el orden de ejecución de la transacción. Si el orden es violado por la operación, entonces la transacción T violará al calendario serial equivalente, abortando así a T. Entonces T se submite al sistema como una nueva transacción con una etiqueta *nueva*. Si T es abortada y regresada, también deberá hacerlo cualquier transacción T1 que pueda haber usado un valor escrito por T. De manera similar, cualquier transacción T2 que pueda haber usado un valor escrito por T1, y así sucesivamente. A este efecto se le conoce como **retroceso en cascada** y es uno de los problemas asociados con TO básico, ya que los calendarios producidos no son recuperables. El algoritmo de control de concurrencia debe checar si en los siguientes dos casos se viola el orden de las etiquetas de las transacciones:

1. La transacción T libera una operación escribir_elemento(X):

   a. Si read_TS(X) > TS(T) o si write TS(X) > TS(T), entonces abortar y regresar a T y rechazar la operación. Esto debe hacerse porque alguna transacción con una etiqueta mayor que TS(T) -y por lo tanto *después* de T en el orden de etiqueta- ya ha leído o escrito el valor del elemento X antes de que T tuviera oportunidad de escribir a X, violando así el orden de etiqueta.

   b. Si la operación en la parte a no ocurre, entonces ejecuta la operación escribir_elemento(X) de T y coloca escribir_TS(X) a TS(T).

2. La transacción T libera una operación leer_elemento(X):

   a. Si write TS(X) > TS(T), entonces abortar y regresar T y rechazar la operación. Esto debe hacerse porque alguna transacción con una etiqueta mayor que TS(T) -y por lo tanto *después*de T en el orden de etiquetas- ya ha escrito el valor del elemento X antes de que T tuviera oportunidad de leer X.

   b. Si write_TS(X) ≤ TS(T), entonces ejecutar la operación leer_elemento(X) de T y colocar leer_TS(X) al máximo de TS(T) y el actual read_TS(X).

Por lo tanto, el algoritmo básico TO checa cada vez que ocurren dos *operaciones conflictivas* en el orden incorrecto, y rechaza a la última de las dos operaciones abortando la transacción que la liberó. Los calendarios producidos por TO básico garantizan ser seriables de conflictos. Una modificación del algoritmo conocida como **regla de escritura de Thomas**, no forza la seriabilidad de conflicto; pero rechaza menos operaciones de escritura, modificando los chequeos de la operación escribir_elemento(X) como sigue:

a. Si read_TS(X) > TS(T), entonces aborta y regresa a T y rechaza la operación.

b. Si write TS(X) > TS(T) entonces no ejecuta la operación de escritura pero continua el procesamiento. Esto se debe a que alguna transacción con etiqueta mayor que TS(T) ya ha escrito el valor de X. Por lo tanto, debemos ignorar la operación escribir_elemento(X) de T porque ya está fuera de tiempo y es obsoleta. Notemos que cualquier conflicto producido por esta situación sería detectado por el caso a.

c. Si no ocurre ninguna de las condiciones de a y b, entonces ejecutemos la operación escribir_elemento(X) de T y colocamos write_TS(X) a TS(T).

El protocolo de ordenamiento por etiquetas, como el de dos fases, garantiza la seriabilidad de los calendarios. Sin embargo, algunos calendarios son posibles bajo cada potocolo que no se permiten bajo el otro. Por lo tanto, ningún protocolo permite que *todos los calendarios seriales* sean *posibles*. Sin embargo, puede ocurrir el reinicio cicílico si una transacción es continuamente abortada y reinicializada.

El algoritmo TO básico reforza el conflicto de seriabilidad, pero no asegura calendarios recuperables; y por lo tanto no garantiza calendarios sin cascada o estrictos. Una variante del TO básico llamada **TO estricto** asegura que todos los calendarios sean estrictos y serializables. En esta variante, una transacción T que libera un leer_elemento(X) o escribir_elemento(X) tal que TS(T) > write_TS(X) tiene su operación de lectura o escritura *retrasada* hasta que la transacción T' que *escribió* el valor de X ha finalizado o abortado. Para implementar este algoritmo, es necesario simular el bloqueo de un elemento X que haya sido escrito por la transacción T' hasta que T' haya finalizado o abortado. Este algoritmo no ocasiona deadlock, ya que T espera a T' sólo si TS(T) > TS(T).

13.3 Técnicas de Control de Concurrencia Multiversión

Otros protocolos para el control de concurrencia mantienen los valores anteriores de un elemento cuando este se actualiza. A esto se le conoce como técnicas **de control de concurrencia multiversión**, debido a que se mantienen diferentes versiones (valores) de un elemento. Cuando una transacción requiere acceso a un elemento, se elige una versión apropiada para mantener la seriabilidad del calendario en ejecución, si es posible. La idea es que algunas operaciones de lectura que se rechazarían en otras técnicas puedan seguir siendo aceptadas por la lectura de *una versión anterior* del elemento para mantener la seriabilidad. Cuando una transacción escribe un elemento, escribe una *nueva versión* y retiene a la anterior. En general, los algoritmos de control de concurrencia multiversión usan el concepto de seriabilidad de vistas en lugar del conflicto de seriabilidad.

Una desventaja obvia de las técnicas multiversión es que se necesita de más almacenamiento para mantener varias versiones de los elementos de la base de datos. Sin embargo, las versiones anteriores pueden tener que mantenerse de cualquier forma -por ejemplo, para propósitos de recuperación. Además, algunas aplicaciones de bases de datos requieren que se almacenen versiones anteriores para mantener un histórico de la evolución de los valores de los elementos. El caso extremo es una *base de datos temporal*, la cual registra todos los cambios y tiempos en que ocurrieron. En tales casos, no hay penalidad adicional para las técnicas multiversión, ya que las versiones anteriores ya están registradas.

13.3.1 Técnica Multiversión Basada en el Orden de Etiquetas

En esta técnica multiversión, se guardan varias versiones X1, X2,..., Xk de cada elemento X por el sistema. Para cada versión, se guarda el valor de la versión Xi y las siguientes dos etiquetas:

1. read_TS(Xi): La **etiqueta de lectura** de Xi; es la más grande de todas las etiquetas de transacciones que leen exitosamente a la *versión* Xi.

2. write_TS(Xi): La **etiqueta de escritura** de Xi; es la etiqueta de la transacción que escribió el valor de la versión Xi.

Cada vez que a una transacción T se le permite ejecutar una operación escribir_elemento(X), una nueva versión X_{k+1} del elemento creado X, con ambos write_TS(X_{k+1}) y el read_TS(X_{k+1}) se colocan en TS(T). Cuando a una transacción T se le permite leer el valor de la versión Xi, el valor de read_TS(Xi) se coloca al mayor read_TS(Xi) y TS(T).

Para asegurar la seriabilidad, usaremos las siguientes dos reglas para controlar la lectura y escritura de elementos de datos:

1. Si la transacción T libera una operación escribir_elemento(X), y una versión i de X tiene el más alto write_TS(Xi) de todas las versiones de X que es también *menor o igual que* TS(T), y TS(T) < read_TS(Xi), entonces abortar y regresar la transacción T; de otro modo, crear una nueva versión Xj de X con read_TS(Xj) = write_TS(Xj) = TS(T).

2. Si la transacción T libera una operación leer_elemento(X), encontrar la versión i de X que tiene el más alto write_TS(Xi) de todas las versiones de X que es también *menor que o igual a* TS(T); entonces retornar el valor de Xi a la transacción T, y coloca el valor de leer_TS(Xi) al mayor TS(T) y el actual read_TS(Xi).

En el caso 1, la transacción T puede abortarse y regresarse. Esto sucede si T intenta escribir una versión de X que deba haberse leído por otra transacción T' cuya etiqueta es read_TS(Xi); sin embargo, T' ya tiene una versión de lectura de Xi, la cual fue escrita por la transacción con etiqueta igual a write_TS(Xi). Si ocurre este conflicto, T es regresado; de otro modo, se crea una nueva versión de X, escrita por la transacción T. Notemos que, si T es regresada, puede ocurrir el regreso en cascada. Por lo tanto, para asegurar la recuperabilidad, a una transacción T no se le permite finalizar hasta después que todas las transacciones que han escrito versiones que T ha leído han finalizado.

13.3.2 Bloqueo Multiversión de Dos Fases

En este esquema, existen tres modos de bloqueo para un elemento: lectura, escritura y certificar. Por lo tanto, el estado de un elemento X puede ser "bloqueo de lectura", "bloqueo de escritura" y "desbloqueo". En el esquema de bloqueo estandar con sólo bloqueos de lectura y escritura, un bloqueo de escritura es exclusivo. Podemos describir la relación entre bloqueos de lectura y escritura en el esquema estandar por medio de la **tabla de compatibilidad de bloqueos** mostrada en la Figura 13.6(a). Una entrada de *si* significa que, si una transacción T retiene el tipo de bloqueo especificado en el encabezado de la columna sobre el elemento X y si la transacción T' requiere el tipo de bloqueo especificado en el encabezado del renglón del mismo elemento X, entonces T' puede *obtener el bloqueo* porque los modos de bloqueo son compatibles. Por otro lado, una entrada de *no* en la tabla indica que los bloqueos no son compatibles, así T' debe esperar hasta que T' libere el bloqueo.

En el esquema de bloqueo estandar, una vez que una transacción obtiene un bloqueo de escritura sobre un elemento, ninguna otra transacción puede accesar a ese elemento. La idea detrás de este bloqueo es permitir que otra transacción T' lea un elemento X mientras que una transacción T retiene un bloqueo de escritura sobre X. Esto se logra permitiendo **dos versiones** de cada elemento X; una versión debe haber sido escrita por alguna transacción finalizada. La segunda versión X' se crea cuando una transacción T adquiere un bloqueo de escritura sobre el elemento. Otras transacciones pueden continuar leyendo la versión finalizada X mientras que T retiene el bloqueo de escritura. Ahora la transacción T puede cambiar el valor de X' cuando sea necesario, sin afectar el valor de la versión X finalizada. Sin embargo, una vez que T está por finalizar, debe obtener un **bloqueo**

certificado de todos los elementos que retienen bloqueos de escritura antes de que finalice. El certificado de bloqueo no es compatible con los de lectura, así la transacción puede tener que retrasar su finalización hasta que todos los elementos con bloqueo de escritura sean liberados por cualquier transacción de lectura. En este punto, la versión final de X se coloca en el valor de la versión X', la versión X' se descarta, y los certificados de bloqueo se liberan. La tabla de compatibilidad de bloqueo para este esquema se muestra en la Figura 13.6(b).

(a)

	Lectura	Escritura
Lectura	si	no
Escritura	no	no

(b)

	Lectura	Escritura	Certificado
Lectura	si	si	no
Escritura	si	no	no
Certificado	no	no	no

Figura 13.6 Compatibilidad de bloqueos: (a) Tabla de compatibilidad para el esquema de bloqueo estandar. (b) Tabla de compatibilidad para bloqueo multiversión de dos fases.

En este esquema de bloqueo multiversión de dos fases, la lectura puede proceder concurrentemente con una operación de escritura -un arreglo no permitido bajo los esquemas de bloqueo de dos fases estandar. El costo es que una transacción puede tener que retrasar su finalización hasta que obtenga bloqueos certificados exclusivos sobre todos los elementos que haya actualizado. Puede mostrarse que este esquema evita los abortos en cascada, ya que a las transacciones solo se le permite leer la versión X que fue escrita por una transacción finalizada. Sin embrago, pueden ocurrir deadlocks si se permite la conversión de un bloqueo de lectura a uno de escritura, y estos deben manejarse mediante variantes de las técnicas ya discutidas.

13.4 Técnicas de Control de Concurrencia por Validación

En todas las técnicas de control de concurrencia discutidas hasta ahora, se lleva a cabo un ligero chequeo *antes* de poder ejecutar una operación sobre la base de datos. Por ejemplo, en el bloqueo, se hace un chequeo para determinar si el elemento que está siendo accesado está bloqueado. En el ordenamiento de etiquetas, se checa la etiqueta de la transacción contra las etiquetas de lectura y escritura del elemento. Tales chequeos representan dolores de cabeza durante la ejecución de las transacciones, con el efecto de hacer lentas las transacciones.

En las técnicas de **control de concurrencia óptimas**, también conocidas como técnicas de **validación** o **certificación**, *no se hace* ningun chequeo mientras se ejecuta la transacción. En este esquema, las actualizaciones en la transacción *no* se aplican directamente a los elementos de la base de datos hasta que la transacción finaliza.

Durante la ejecución de la transacción, todas las actualizaciones se aplican a *copias locales* de los elementos que se guardan por la transacción. Al final de la ejecución de la

transacción, una **fase de validación** checa si cualquiera de las transacciones que actualizan viola la seriabilidad. Si no se viola la seriabilidad, la transacción finaliza y la base de datos se actualiza de las copias locales, de otro modo, se aborta la transacción y se reinicia posteriormente.

Existen tres fases para este protocolo de control de concurrencia:

1. La **fase de lectura**: Una transacción puede leer valores de elementos de la base de datos. Sin embargo, las actualizaciones se aplican sólo a copias locales de los elementos guardadas en el espacio de trabajo de la transacción.

2. La **fase de validación**: Se realiza un chequeo para asegurar que la seriabilidad no será violada si las actualizaciones de la transacción se aplican a la base de datos.

3. La **fase de escritura**: Si la fase de validación tiene éxito, la actualización de la transacción se aplica a la base de datos; de otro modo, se descartan las actualizaciones y se reinicia la transacción.

La filosofía de este método es hacer todos los chequeos una sola vez; por lo tanto, la ejecución de una transacción procede con un mínimo de problemas hasta alcanzar la fase de validación. Si existe interferencia entre las transacciones, la mayoría de las transacciones serán validadas con éxito. Sin embargo, si existe mucha interferencia, muchas transacciones que ejecutan su finalización tendrán sus resultados descartados y deberán reiniciarse posteriormente. Bajo estas circunstancias, las técnicas optimistas no trabajan bien. Se les llama técnicas "optimistas" porque suponen que ocurrirá poca interferencia y no hay necesidad de chequeo durante la ejecución de la transacción.

El protocolo optimista que describimos usa etiquetas de transacción y también requiere que se guarden en el sistema los juegos de escritura y de lectura. Además, para cada transacción necesitan guardarse los tiempos de inicio y finalización de alguna de las tres fases. El **juego de escritura** de una transacción es el conjunto de elementos escritos por la transacción, mientras que el **juego de lectura** es el conjunto de elementos leídos por la transacción. En su fase de validación de la transacción Ti, el protocolo checa que Ti no interfiera con ninguna transacción finalizada o con cualquier otra que esté en su fase de validación. La fase de validación de Ti checa que, para cada transacción Tj que finalice o esté en su fase de validación, se *retenga* una de las siguientes condiciones:

1. La transacción Tj completa su fase de escritura antes de que Ti inicie su fase de lectura.

2. Ti inicia su fase de escritura después de que Tj lo hace, y el juego de lectura de Ti no tiene elementos en común con el juego de escritura de Tj.

3. Ambos juegos de lectura y escritura de Ti no tienen elementos en común con el juego de escritura de Tj, y Tj finaliza su fase de lectura antes que Ti.

Si se retiene cualquiera de estas tres condiciones, no existe interferencia y Ti se valida con éxito. Si no se retiene ninguna de las tres condiciones, falla la validación de la transacción Ti y se aborta y reinicia posteriormente porque puede haber ocurrido alguna interferencia.

13.5 Granularidad de los Elementos de Datos

Todas las técnicas de control de concurrencia asumen que la base de datos estaba compuesta de un número de elementos. Puede elegirse alguno de los elementos para ser uno de la siguiente lista:

- Registro de la base de datos.
- Valor del campo de un registro.
- Bloque de disco.
- Archivo completo.
- Base de datos completa.

En la elección del tamaño del elemento deben considerarse varios intercambios. Discutiremos el tamaño del elemento dentro del contexto del bloqueo, a pesar de que pueden hacerse argumentos similares para otras técnicas de control de concurrencia.

Primero, notemos que mientras más grande sea el elemento, menor es el grado de concurrencia permitido. Por ejemplo, si el tamaño del elemento es un bloque de disco, una transacción T que necesite bloquear a un registro A debe bloquear al bloque completo que contenga a A. Esto se debe a que el bloqueo se asocia con el elemento X completo. Ahora, si otra transacción S quiere bloquear a un registro B diferente que reside en el mismo bloque X en un modo de bloqueo conflictivo, se forza a esperar hasta que la primera transacción libere el bloqueo sobre el bloque X. Si el tamaño del elemento fuera de un registro, la transacción S podría proceder como si hubiera bloqueado a un elemento diferente (registro B)que al bloqueado por T (registro A).

Por otro lado, mientras más pequeño sea el tamaño del elemento, existirán más elementos en la base de datos. Debido a que cada elemento se asocia con un bloqueo, el sistema tendrá un mayor número de bloqueos activos que manejar. Se realizarán más operaciones de bloqueo y desbloqueo, ocasionando un mayor problema. Además, se requerirá de mayor espacio de almacenamiento para la tabla de bloqueos. Para etiquetas, se requiere espacio para cada elemento read_TS y write_TS, y el problema de manejar un mayor número de elementos es similar al del caso de bloqueo.

Al tamaño de los elementos de datos se le llama **granularidad del elemento de datos**. La *granularidad fina* se refiere a pequeños elementos, mintras que la *granularidad tosca* se refiere a grandes elementos de datos. Dados los apartados anteriores, la pregunta es: Cuál

es el mejor tamaño? La respuesta es *depende del tipo de transacciones involucradas*. Si una transacción típica accesa un pequeño número de registros, es ventajoso tener granularidad de un registro. Por otro lado, si una transacción accesa muchos registros del mismo archivo, podría ser mejor tener granularidad de bloque o archivo de tal forma que la transacción considerará a todos los registros como un (o unos cuantos) elementos.

La mayoría de las técnicas de control de concurrencia tienen un tamaño uniforme. Sin embargo, se han propuesto algunas técnicas que permiten tamaños variables. En estas técnicas, el tamaño del elemento puede cambiarse a la granularidad que mejor convenga a las transacciones que se ejecuten en el sistema.

13.6 Algunos Otros Temas del Control de Concurrencia

En esta sección dicutiremos algunos otros puntos relevantes al control de concurrencia, los problemas asociados con la inserción y borrado de registros y el así llamado "problema fantasma", el cual puede ocurrir cuando se insertan registros. Los problemas que pueden ocurrir cuando una transacción saca algunos datos a una terminal antes de finzalizar, y posteriormente se aborta la transacción.

13.6.1 Inserción, Borrado y Registros Fantasma

Cuando se **inserta** un nuevo elemento a la base de datos, obviamente no puede accesarse hasta después de haber sido creado y que la operación de inserción ha finalizado. En un medio ambiente de bloqueo, puede crearse un bloqueo para el elemento y colocarse en modo exclusivo; basado en el protocolo de control de concurrencia, el bloqueo puede liberarse al mismo tiempo que otros bloqueos. Para un protocolo basado en etiquetas, las etiquetas de lectura y escritura del nuevo elemento se colocan en la etiqueta de la transacción creadora.

Una operación de **borrado** se aplica sobre un elemento existente. Para los protocolos de bloqueo, debe obtenerse un bloqueo exclusivo antes de que la transacción pueda borrar al elemento. Para el ordenamiento de etiquetas, el protocolo debe asegurarse que ninguna transacción posterior ha leído o escrito al elemento antes de permitir que una transacción lo borre.

Cuando un nuevo registro que está siendo insertado por alguna transacción T satisface una condición que un conjunto de registros accesó por otra transacción T' puede ocurrir un problema conocido como el **problema fantasma**. Por ejemplo, supongamos que la transacción T inserta a un nuevo EMPLEADO que pertenece al departamento número 5, mientras que la transacción T' accesa a todos los registros cuyo DNO=5 para agregarles sus SALARIOS. Si el orden serial equivalente es T seguido por T', entonces T' debe leer al nuevo empleado e incluir su salario en el cálculo de la suma. Para el orden serial equivalente T' seguido por T, el no se incluiría el nuevo salario. Notemos que, a pesar de que la transacción está en conflicto lógico, en el primer caso no existe registro en común entre las dos transacciones ya que T' puede haber bloqueado a todos los registros con

DNO=5 *antes* de que T insertara al nuevo registro. Esto se debe a que el registro que ocasiona el conflicto es un **registro fantasma** que ha aparecido de repente en la base de datos.

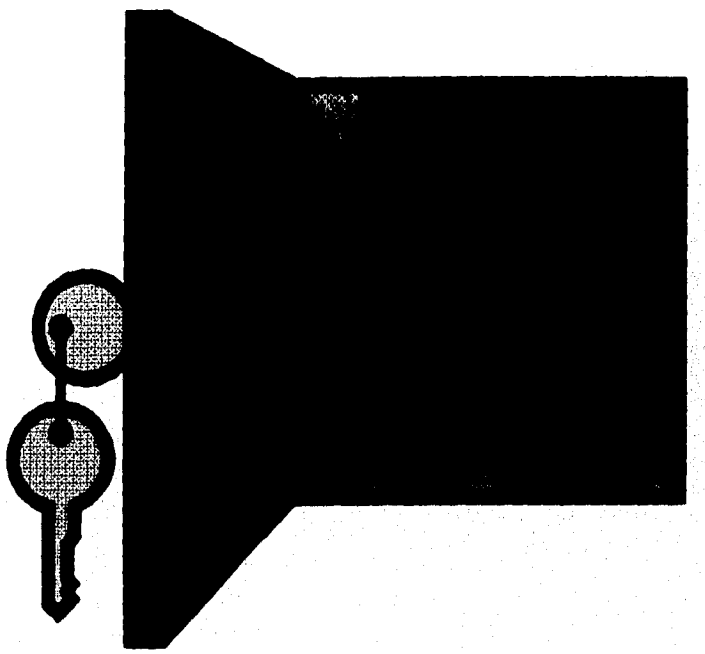
Una solución al registro fantasma se conoce como **índice de bloqueo**, y puede usarse junto con un protocolo de bloqueo de dos fases. Un índice incluye entradas que tienen el valor de un atributo, más un conjunto de apuntadores a todos los registros en el archivo con ese valor. Por ejemplo, un índice sobre el atributo DNO del archivo EMPLEADO incluye una entrada para cada valor de DNO distinto, más un conjunto de apuntadores a todos los registros EMPLEADO con ese valor. Si una entrada índice está bloqueada *antes* de que pueda accesarse al registro, entonces puede detectarse el conflicto del registro fantasma. Esto se debe a que la transacción T' requiriría un bloqueo de lectura sobre la *entrada índice* para DNO=5, y T requiriría un bloqueo de escritura sobre la misma entrada *antes* de poder colocar los bloqueos sobre los registros actuales. Debido a que el bloqueo del índice entra en conflicto, se detectaría el conflicto fantasma. Una técnica más general, llamada **bloqueo predicado**, bloquearía el acceso a todos los registros que satisfagan de manera similar a un *predicado arbitrario* (condición); sin embargo el bloqueo predicado ha probado ser difícil de implementar con eficiencia.

13.6.2 Transacciones Interactivas

Cuando las transacciones interactivas leen entradas y despliegan la salida a un dispositivo interactivo ocurren otros problemas, tal como la pantalla de una terminal, antes de haber finalizado. El problema es que un usuario puede ingresar un valor de un elemento a una transacción T que se basa en algun valor escrito a la pantalla por la transacción T', la cual puede que no haya aún finalizado. Esta dependencia entre T y T' no puede modelarse por el sistema de control de concurrencia, ya que sólo se basa en la interacción del usuario con las dos transacciones.

Una opción para trabajar con este problema es posponer el despliegue de las transacciones hasta que hayan finalizado.

CAPITULO 14



Seguridad y Autorización de la Base de Datos

14.1 Introducción a los Temas de Seguridad de Bases de Datos

14.1.1 Tipos de Seguridad

La seguridad de las bases de datos es un área muy amplia que incluye muchos temas, entre ellos a los siguientes:

- Temas legales y éticos que tienen que ver con el acceso correcto de cierta información. Alguna información puede juzgarse como privada y no accesarse legalmente por personal no autorizado. En los Estados Unidos, varios estados y el gobierno federal tienen leyes de privacidad de información.

- Temas políticos a nivel gubernamental, institucional o corporativo -por ejemplo, tasas de crédito y registros médicos personales.

- Los temas relacionados con sistemas tales como *niveles* en los cuales deben manejarse varias funciones de seguridad -por ejemplo, a nivel físico de hardware, sistema operativo, o DBMS.

- La necesidad en algunas organizaciones de identificar múltiples *niveles de seguridad* y categorizar los datos y usuarios basados en estas clasificaciones -por ejemplo, alto secreto, secreto, confidencial, y sin clasificación. Debe reforzarse la política de seguridad de la organización con respecto a permitir acceso a varias clasificaciones de datos.

En un sistema multiusuario, el DBMS debe proporcionar técnicas que permitan que ciertos usuarios o grupos de ellos accesen secciones seleccionadas de la base de datos sin ganar acceso al resto de ella. Esto es particularmente importante cuando dentro de la misma organización se usa por muchos usuarios una gran base de datos integrada. Información sensible tal como los sueldos de los empleados debe mantenerse confidencial a la mayoría de los usuarios. Un DBMS típicamente incluye un **subsistema de seguridad y autorización** que es responsable de preservar la seguridad de porciones de una base de datos contra acceso no autorizado.

Ahora es habitual referirnos a dos tipos de mecanismos de seguridad:

- **Mecanismos de seguridad discreta** se usan para otorgar privilegios a usuarios, incluyendo la capacidad de accesar archivos de datos específicos, registros, o campos en un modo especificado (tal como lectura, escritura, o actualización).

- **Mecanismos de seguridad por mandato** se usan para reforzar la seguridad multinivel clasificando los datos y usuarios en varias clases de seguridad (o niveles) y después implementar la política de seguridad apropiada para la organización. Por ejemplo, una política de seguridad típica es permitir a los usuarios un cierto nivel de clasificación para ver un sólo elemento al nivel más bajo.

Otro problema común de seguridad en todos los sistemas de cómputo es prevenir que personas no autorizadas accesen al sistema -ya sea que obtengan información o hagan cambios con malicia a alguna porción de la base de datos. El mecanismo de seguridad de un DBMS debe incluir provisiones para restringir el acceso al sistema completo. A esta función se le llama **control de acceso** y se maneja creando cuentas de usuario y passwords para controlar el proceso de entrada al DBMS.

Un tercer problema de seguridad asociado con las bases de datos es el control de acceso a una **base de datos estadística**, la cual se usa para proporcionar información estadística o resumida de valores basados en diferentes criterios. Por ejemplo, una base de datos de población puede proporcionar información acerca de grupos por edades, niveles de ingresos, tamaño de las casas, niveles de educación, y otros criterios. A los usuarios de bases estadisticas tales como el gobierno o firmas de investigación de mercados se les permite acceso a la base de datos para recuperar información acerca de una población pero no a la información confidencial detallada sobre individuos en específico. La seguridad en las bases estadísticas debe asegurar que no pueda accesarse la información sobre individuos. A veces es posible deducir ciertos hechos concernientes a individuos a partir de queries que involucran sólo resúmenes sobre grupos; consecuentemente esto no debe permitirse.

Otra ténica de seguridad es la **encripción de datos**, la cual se usa para proteger los datos sensibles que se transmiten via satélite o por algun otro tipo de red. La encripción puede usarse para proporcionar protección adicional a porciones sensibles de una base de datos. La información se **codifica** usando algun algoritmo. Un usuario no autorizado quien accese datos codificados tendrá dificultad para descifrarlos, pero a los usuarios autorizados se les proporcionan algoritmos de decodificación (o claves) para descifrar los datos. En aplicaciones militares las técnicas de encriptamiento son dificiles de codificar sin que haya sido desarrollada una llave.

14.1.2 Seguridad en la Base de Datos y el DBA

El DBA es la autoridad central para manejar un sistema de bases de datos. Las resposabilidades del DBA incluyen proporcionar privilegios a usuarios que necesiten usar el sistema y clasificar a usuarios y datos de acuerdo con la política de la organización. El DBA tiene una **cuenta privilegiada** en el DBMS, algunas veces llamada **cuenta del sistema**, la cual proporciona capacidades poderosas que no están disponibles para cuentas y usuarios comúnes. Esta cuenta es similar a la cuenta *raiz* o *superusuario* que se da a los administradores del sistema de cómputo, permitiendo acceso a comandos restringidos del sistema operativo. Los comandos privilegiados del DBA incluyen aquéllos que proporcionan y revocan privilegios en cuentas individuales, usuarios, o grupos de usuarios y realizan el siguiente tipo de acciones:

1. *Creación de la cuenta*: Esta acción crea una nueva cuenta y un password para un usuario o grupo de usuarios que les permita accesar al DBMS.

2. *Otorgamiento de privilegios*: Esta acción permite al DBA proporcionar ciertos privilegios a ciertas cuentas.

3. *Revocación de privilegios*: Esta acción permite al DBA revocar (cancelar) ciertos privilegios que fueron previamente dados a ciertas cuentas.

4. *Asignación del nivel de seguridad*: Esta acción consiste en asignar a las cuentas de usuarios al nivel de clasificación adecuado.

El DBA es responsable de la seguridad global del sistema de base de datos. La acción 1 de la lista anterior se usa para controlar el acceso al DBMS como un todo, mientras que las acciones 2 y 3 se usan para controlar las autorizaciones discretas de la base de datos, y la acción 4 se usa para controlar la autorización mandatoria.

14.1.3 Protección de Acceso, Cuentas de Usuarios, y Auditorías de Bases de Datos

Cada vez que una persona o grupo necesita accesar al sistema de base de datos, el individuo o grupo debe tener una cuenta de usuario. El DBA creará entonces un nuevo **número de cuenta** y **password** para el usuario si es que tiene necesidad legítima de usar la base de datos. Cada que necesita acceso el usuario debe **registrarse** en el sistema ingresando su número de cuenta y password. El DBMS checa que el número de cuenta y password sean válidos; si lo son, se le permite al usuario usar el DBMS y accesar a la base de datos. Los programas de aplicación también pueden considerarse como usuarios y requerirseles un password.

Es sencillo registrar a los usuarios de la base de datos y sus cuentas y passwords creando una tabla o archivo encriptado con los campos Número_Cuenta y Password. Esta tabla puede ser fácilmente mantenida por el DBMS. Cada vez que se crea una nueva cuenta, se inserta un nuevo registro a la tabla. Cuando se cancela una cuenta, debe eliminarse el registro correspondiente de la tabla.

El sistema debe registrar también todas las operaciones que fueron aplicadas sobre la base de datos por un usuario a través de cada **sesión de registro**, la cual consta de la secuencia de interacciones que un usuario realiza desde la hora de registro hasta la hora de salida. Cuando un usuario se registra, el DBMS puede detectar el número de cuenta y asociarlo con la terminal desde la cual se ingresó. Todas las operaciones aplicadas desde esa terminal se atribuyen al usuario de la cuenta hasta que éste la abandona. Es particularmente importante registrar las operaciones de actualización que se aplican a la base de datos de tal forma que, si la base de datos es manoseada, el DBA pueda descubrir qué usuario lo hizo.

Para registrar todas las aplicaciones aplicadas a la base de datos y del usuario en particular que hizo cada actualización, podemos modificar la bitácora del sistema. La bitácora del sistema incluye una entrada por cada operación aplicada a la base de datos que pueda

requerir de una recuperación debido a una falla en la transacción o caída del sistema. Se puede expander la bitácora de tal forma que incluya el número de cuenta del usuario y el id de la terminal que aplicó cada operación registrada en la bitácora. Si se sospecha de algun daño, se realiza una **auditoría de base de datos**, la cual consta de revisar la bitácora para examinar todos los accesos y operaciones aplicadas a la base de datos durante un cierto periodo de tiempo. Cuando se encuentra una operación ilegal o no autorizada, el DBA puede determinar el número de cuenta utilizado para realizar esta operación. Las auditorías de bases de datos son particularmente importantes en bases de datos sensibles que se actualizan por medio de transacciones y usuarios, tal como las bases bancarias que se actualizan por muchas sucursales. A las bitácoras que se usan principalmente con propósitos de seguridad se les llama **pista de auditoría**.

14.2 Control de Acceso Discreto Basado en Privilegios

El método típico de reforzar el **control de acceso discreto** en un sistema de base de datos se basa en proporcionar y revocar privilegios. Consideremos los privilegios desde el contexto de un DBMS relacional. En particular, discutiremos un sistema de privilegios algo similar al desarrollado originalmente por SQL. Muchos DBMSs relacionales actuales usan alguna variante de esta técnica. La idea principal es incluir instrucciones adicionales en el lenguaje query que permitan al DBA y usuarios selectos proporcionar y revocar privilegios.

14.2.1 Tipos de Privilegios Discretos

En SQL2, se usa el concepto de *identificador de autorización* para referir, burdamente hablando, a una cuenta de usuario. Basado en cuentas específicas, el DBMS debe proporcionar acceso selectivo a cada relación en la base de datos. Las operaciones también pueden controlarse; así, tener una cuenta no necesariamente da derecho al poseedor de la misma a toda la funcionalidad proporcionada por el DBMS. Informalmente, para usar el sistema de base de datos existen dos niveles de asignación de privilegios:

1. *El nivel de cuenta*: En este nivel, el DBA especifica los privilegios particulares que cada cuenta tiene independientemente de las relaciones en la base de datos.

2. *El nivel de relación*: En este nivel, podemos controlar el privilegio de accesar cada relación individual o vista en la base de datos.

Los privilegios a **nivel de cuenta** se aplican a las capacidades proporcionadas por la cuenta misma y pueden incluir el privilegio CREATE SCHEMA o CREATE TABLE, para crear un esquema o relación base; el privilegio CREATE VIEW; ALTER, para agregar o eliminar atributos de las relaciones; DROP, para borrar relaciones o vistas; MODIFY, para insertar, borrar o actualizar tuplas; y SELECT, para recuperar información usando un query. Notemos que estos privilegios de cuenta se aplican a la cuenta en general. Si cierta cuenta no tiene el privilegio CREATE TABLE, no pueden

crearse relaciones desde esa cuenta. Los privilegios a nivel de cuenta *no se* definen como parte de SQL2; se dejan para que los definan los implementadores del DBMS. En versiones anteriores de SQL, existió un privilegio CREATETAB para dar a una cuenta el privilegio de crear tablas (relaciones).

El segundo nivel de privilegios se aplica a relaciones individuales, ya sean relaciones base o virtuales (vistas). Estos privilegios *se definen* por SQL2. En la siguiente discusión, el término *relación* puede referir a una relación base o vista, a menos que se especifique explícitamente la una o la otra. Los privilegios a nivel de relación especifican a cada usuario las relaciones individuales sobre las cuales puede aplicarse cada tipo de comando. Algunos privilegios también refieren a columnas individuales (atributos) de relaciones. Los comandos de SQL2 proporcionan privilegios a *nivel de relación y atributo únicamente*. A pesar de que esto es algo general, dificulta la creación de cuentas con privilegios limitados. La proporción y revocación de privilegios generalmente sigue un modelo de organización para privilegios discretos conocidos como el **modelo de acceso matriz**, donde los renglones de la matriz M representan *sujetos* (usuarios, cuentas, programas) y las columnas representan *objetos* (relaciones, registros, columnas, vistas, operaciones). Cada posición M(i,j) en la matriz representa los tipos de privilegios (lectura, escritura, actualización) que el sujeto i retiene sobre el objeto j.

Para controlar la proporción y revocación de privilegios de relación, cada relación R en una base de datos se asigna a un **propietario de cuenta**, la cual es típicamente la cuenta que fue usada cuando la relación fue creada en primer lugar. Al propietario de una relación se le dan *todos* los privilegios sobre esa relación. En SQL2, el DBA puede asignar un propietario a un esquema completo creando el esquema y asociándolo al identificador de autorización apropiado con ese esquema, usando el comando CREATE SCHEMA. El propietario de la cuenta propiedad puede pasar privilegios sobre cualquiera de las relaciones propiedad de otros usuarios **proporcionando** privilegios a sus cuentas. En SQL pueden proporcionarse los siguientes tipos de privilegios sobre cada relación individual R:

- SELECT sobre R: Da a la cuenta el privilegio de recuperación. En SQL la cuenta puede usar la instrucción SELECT para recuperar tuplas de R.
- MODIFY sobre R: Da a la cuenta la capacidad de modificar tuplas de R. En SQL este privilegio se divide en UPDATE, DELETE, e INSERT para aplicar el comando correspondiente a R. Además, INSERT y UPDATE pueden especificar que sólo ciertos atributos de R pueden actualizarse por esa cuenta.
- REFERENCES sobre R: Da a la cuenta la capacidad de referenciar la relación R cuando se especifican argumentos de integridad. Este privilegio puede también restringirse a atributos específicos de R.

Notemos que para crear una vista, la cuenta debe tener el privilegio SELECT sobre *todas las relaciones* involucradas en la definición de la vista.

14.2.2 Especificación de Autorización por Uso de Vistas

El mecanismo de vistas es un mecanismo discreto en su propio derecho. Por ejemplo, si el propietario A de una relación R desea que otra cuenta B sea capáz de recuperar sólo algunos campos de R, entonces A puede crear una vista V de R que incluya sólo a esos atributos y luego proporcionar SELECT sobre V a B. Igualmente se aplica limitando a B a recuperar sólo ciertas tuplas de R; puede crearse una vista V' definiendo la vista por medio de un query que selecciona sólo aquéllas tuplas de R que A quiere permitir que B accese.

14.2.3 Revocación de Privilegios

En algunos casos es deseable proporcionar algun privilegio a un usuario temporalmente. Por ejemplo, el propietario de una relación puede desear proporcionar el privilegio SELECT a un usuario para una tarea específica y luego revocar ese privilegio una vez que finalice la tarea. Por lo tanto, es necesario un mecanismo para **revocar** los privilegios. En SQL se incluye un comando REVOKE con el propósito de cancelar privilegios.

14.2.4 Propagación de Privilegios y la OPCION GRANT (Proporciona)

Cada vez que el propietario A de una relación R proporciona un privilegio sobre R a otra cuenta B, puede darse el privielgio a B con o sin **GRANT OPTION**. Si se da GRANT OPTION, significa que B también puede proporcionar el privilegio sobre R a otras cuentas. Supongamos que a B se le da GRANT OPTION por A y que B proporciona el privilegio sobre R a una tercera cuenta C, también con GRANT OPTION. De esta manera, los privilegios sobre R pueden **propagarse** a otras cuentas sin el conocimieto del propietario de R. Si la cuenta propietaria de A revoca el privilegio proporcionado a B, todos los privilegios que B propagó basados en ese privilegio deben ser automáticamente revocados por el sistema. Por lo tanto, un DBMS que permite la propagación de privilegios debe registrar cómo fueron proporcionados para poder así revocarlos correcta y completamente.

Para limitar la propagación de privilegios se han desarrollado técnicas, a pesar de que no han sido implementadas en la mayoría de los DBMSs. Limitar la **propagación horizontal** a un entero i significa que a una cuenta B que se le da GRANT OPTION puede proporcionar el privilegio al menos a otras i cuentas. La **propagación vertical** es más complicada; limita la profundidad de los privilegios de proporción, Proporcionar un privilegio con propagación vertical de cero es equivalente a proporcionar el privilegio sin *GRANT OPTION*. Si la cuenta A proporciona un privilegio a la cuenta B con propagación vertical puesta a un entero $i > 0$, significa que la cuenta B tiene GRANT OPTION sobre ese privilegio, pero B puede proporcionar el privilegio a otras cuentas sólo con propagación vertical *menor que j*. En efecto, la propagación vertical limita la secuencia de proporción de opciones que pueden darse desde una cuenta a la siguiente basado en una sola proporción del privilegio.

14.3 Control de Acceso Mandatorio para Seguridad Multinivel

La técnica de control de acceso discreto de proporcionar y revocar privilegios sobre relaciones ha sido tradicionalmente el mecanismo de seguridad de los sistemas de bases de datos. Este es un método de todo o nada: ya que un usuario tiene o no un cierto privilegio. En muchas aplicaciones, se necesita de una política de seguridad adicional que clasifique datos y usuarios basado en clases de seguridad. Este acercamiento, conocido como **control de acceso mandatorio**, se *combina* con los mecanismos de control discretos. Es importante recalcar que la mayoría de los DBMSs comerciales proporcionan mecanismos sólo para el control de acceso discreto. Sin embargo, existe la necesidad de seguridad multinivel en el gobierno, la milicia, y aplicaciones de inteligencia, así como en muchas aplicaciones industriales y corporativas.

Las **clases de seguridad** típicas usadas son: alto secreto (TS), secreto (S), confidencial (C) y no clasificada (U), donde TS es el nivel más alto y U el más bajo. Existen otros esquemas más complejos de clasificación, en los cuales las clases de seguridad se organizan en una matriz. Para ilustrar nuestra discusión, usaremos el sistema de clasificación de cuatro niveles, donde TS > S > C > U. El modelo comúnmente usado para seguridad multinivel, conocido como el modelo Bell-LaPadula, clasifica a cada *sujeto*(usuario, cuenta, programa) y *objeto*(relación, tupla, columna, vista, operación) en cualquiera de las clasificaciones de seguridad TS, S, C, U. Nos referiremos a la clasificación de un sujeto S como **clase(S)** y a la clasificación de un objeto O como **clase(O)**. Basado en las clasificaciones sujeto/objeto se forzan dos restricciones sobre el acceso de datos:

1. A un sujeto S no se le permite acceso de lectura a un objeto O a menos que clase(S) ≥ clase(O). Esto se conoce como *propiedad de seguridad sencilla*.

2. A un sujeto S no se le permite acceso de escritura a un objeto O a menos que clase(S) ≥ clase(O). A esta se le conoce como *propiedad* * (o *propiedad estrella*).

La primer restricción forza la regla obvia de que ningún sujeto puede leer un objeto cuya clasificación de seguridad es más alta que la del sujeto. La segunda restricción prohibe a un sujeto de escribir un objeto que tiene menos clasificación de seguridad que la del sujeto mismo. La violación de esta regla permitiría que la información fluyera de clasificaciones más altas a más bajas, lo cual viola un principio de seguridad básico.

Para incorporar nociones de seguridad multinivel en el modelo relacional, es común considerar valores de atributos y tuplas como objetos. Por lo tanto, cada atributo A se asocia con un **atributo de clasificación** C en el esquema, y a cada valor de atributo en una tupla se asocia con una clasificación de seguridad correpondiente. Además, en algunos modelos, se agrega a los atributos de la relación un atributo de **clasificación de tupla** TC para proporcionar una clasificación para cada tupla como un todo. Por lo tanto, un esquema de relación R **multinivel** con n atributos se representaría como:

R(A1, C1, C2, ..., An, Cn, TC)

donde cada Ci representa el atributo de clasificación asociado con el atributo Ai.

El valor del atributo TC en cada tupla proporciona una clasificación general para la tupla misma, mientras que cada Ci proporciona una clasificación de seguridad más fina para cada valor de atributo dentro de la tupla. El valor de TC dentro de una tupla t debe ser el *más alto* de todos los valores de clasificación de atributos dentro de t. La **llave aparente** de una relación multinivel es el conjunto de atributos que habrían formado la llave primaria en una relación regular (un solo nivel). Una relación multinivel parecerá contener diferentes datos a sujetos (usuarios) con diferentes niveles de clasificación. En algunos casos, es posible almacenar una sola tupla en la relación a un nivel de clasificación más alto y producir las tuplas correspondientes a un nivel de clasificación inferior a través de un proceso conocido como **filtrado**. En otros casos, es necesario almacenar dos o más tuplas a diferentes niveles de clasificación con el mismo valor de **llave aparente**. Esto conduce al concepto de **poliinstanciamiento**, donde varias tuplas pueden tener el mismo valor de llave aparente pero tener diferentes valores de atributos para usuarios a diferentes niveles de clasificación.

14.4 Seguridad en Bases de Datos Estadísticas

Las bases de datos estadísticas se usan principalmente para producir estadísticas sobre varias poblaciones. La base de datos puede contener información confidencial sobre muchos individuos, los cuales deben ser protegidos de acceso a usuarios. Sin embargo, se les puede permitir recuperar información estadística sobre las poblaciones, tal como promedios, conteos, sumas, y desviaciones estandar. Ilustraremos el problema con la relación mostrada en la Figura 14.3, la cual muestra una relación **PERSONA** con los atributos NOMBRE, IMSS, INGRESOS, DIRECCION, CIUDAD, ESTADO, CP, SEXO y ESTUDIOS.

Una **población** es un conjunto de tuplas de un archivo que satisfacen alguna condición de selección. Por lo tanto, cada condición de selección en la relación **PERSONA** especificará una población particular de tuplas **PERSONA**. Por ejemplo, la condición **SEXO='M'** especifica a la población masculina.

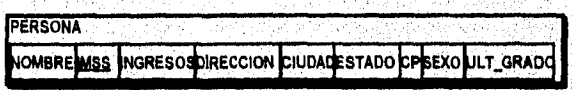


Figura 14.3 La relación **PERSONA**.

Los queries estadísticos involucran la aplicación de funciones estadísticas a una población de tuplas. Por ejemplo, podemos recuperar el número de individuos en una población o el promedio de ingresos en la población. Sin embargo, a los usuarios estadísticos no se les permite recuperar información individual, tal como el ingreso de una persona específica.

CAPITULO 15

Técnicas de Recuperación

15.1 Conceptos de Recuperación

15.1.1 Esbozo de Recuperación

La recuperación de fallas en las transacciones usualmente significa que la base de datos se *restaura* de algun estado en el pasado a un estado correcto -cercano a la hora de la falla- puede *reconstruirse* a partir de ese estado pasado. Para lograrlo, el sistema debe mantener información acerca de los cambios a elementos de datos durante la ejecución de la transacción fuera de la base de datos. Esta información típicamente se registra en la **bitácora del sistema**. Informalmente, una estrategia típica de recuperación puede globalizarse como sigue:

1. Si existe un daño extensivo a una gran porción de la base de datos debido a una falla catastrófica, tal como alguna falla en el disco, el método de recuperación restaura una copia pasada de la base de datos que fue *descargada* para archivamiento (típicamente en cinta) y reconstruye un estado más actualizado reaplicando o *rehaciendo* operaciones de transacciones finalizadas desde el inicio hasta la hora de la falla.

2. Cuando la base de datos no se daña fisicamente pero se vuelve inconsistente debido a fallas no catastróficas, la estrategia es revertir los cambios que ocasionaron la inconsistencia *haciendo* algunas operaciones para restaurar un estado consistente de la base de datos. En este caso no necesitamos una copia completa del archivo de la base de datos. En cambio, se registran las entradas en la bitácora del sistema y se consultan durante la recuperación.

Podemos distinguir dos técnicas principales de recuperación de fallas catastróficas en las transacciones. Las técnicas de **actualización deferida** no actualizan a la base de datos hasta *después* que una transacción alcanza su punto final; después las actualizaciones se registran en la base de datos. Antes de finalizar, todas las actualizaciones de transacciones se registran en el espacio de trabajo de la transacción local. Durante la finalización, las actualizaciones se registran primero persistentemente en la bitácora y después se escriben a la base de datos. Si una transacción falla antes de alcanzar su punto final, no habrá cambiado a la base de datos de ninguna manera, así que no es necesario el UNDO. Puede ser necesario REHACER el efecto de las operaciones de una transacción finalizada desde la bitácora, debido a que su efecto puede haber no sido registrado en la base de datos. Por lo tanto, a la actualización deferida también se le conoce como el algoritmo **NO-UNDO/REDO**.

En las técnicas de **actualiación inmediata**, la base de datos puede actualizarse mediante algunas operaciones *antes* de que la transacción alcance su punto final. Sin embargo, antes de aplicarse a la base de datos, estas operaciones se registran en la *bitácora en disco* mediante escritura forzada posibilitando así la recuperación. Si una transacción falla después de registrar algunos cambios en la base de datos pero sin haber alcanzado su punto final, debe deshacerse el efecto de sus operaciones sobre la base de datos; esto es, debe retrocederse la transacción. En el caso general de actualización inmediata, se

requiere del *undo* y *redo* durante la recuperación, así se le conoce como el **algoritmo UNDO/REDO**. Una variante del algoritmo donde todas las actualizaciones se registran en la base de datos antes de que una transacción finalice requiere sólo del *undo*, así se le conoce como el **algoritmo UNDO/NO-REDO**.

15.1.2 Conceptos de Sistemas para Recuperación

El proceso de recuperación con frecuencia se relaciona con las funciones del sistema operativo -en particular, el buffereo y cache de páginas de disco en memoria principal. Típicamente, una o más páginas que incluyen al elemento de datos a actualizar son **cacheados** a un buffer en memoria principal y luego actualizados en memoria antes de volver a ser escritos en disco. El cacheo de páginas en disco es tradicionalmente una función del sistema operativo, pero debido a su importancia en la eficiencia de los procedimientos de recuperación, es algunas veces manejado por el DBMS llamando a rutinas de bajo nivel del sistema operativo. Para propósitos de recuperación en general, es conveniente considerar que cada elemento corresponde a una página de disco.

Típicamente se guarda una colección de buffers en memoria, llamados **cache DBMS**, bajo el control del DBMS con el propósito de retener elementos de la base. Esta puede ser una tabla de entradas <nombre del elemento, localidad del buffer>. Cuando el DBMS requiere acción sobre algun elemento, primero checa en el directorio y determina si el elemento está en el cache. Si no lo está, el elemento debe localizarse en el disco, copiando al cache las páginas de disco apropiadas. Para hacerle espacio al nuevo elemento puede ser necesario **limpiar** algunos de los buffers caches. Para el limpiado de buffers pueden usarse algunas estrategias, reemplazamiento de páginas del sistema operativo, tales como el más recientemente usado (LRU) o primero en entrar primero en salir (FIFO).

Asociado con cada elemento en el cache está un **bit sucio**, el cual puede incluirse en la entrada del directorio, para indicar si o no ha sido modificado el elemento. Cuando se lee el elemento del disco a buffer, el directorio del buffer se actualiza con el nuevo nombre del elemento, y el bit sucio se pone en 0 (cero). Tan pronto como se modifica el elemento, el bit sucio del directorio correspondiente a la entrada se pone en 1 (uno). Cuando se limpia a un elemento, se regresa a disco sólo si su bit sucio es 1.

Cuando se limpia un elemento modificado a disco se emplean dos estrategias principales. La primera, conocida como **actualización en el lugar**, escribe el elemento en la *misma localidad del disco*, sobreescribiendo el valor anterior del elemento sobre el disco. Por lo tanto, se mantiene en disco una copia sencilla de cada elemento. La segunda, conocida como **sombra**, escribe un nuevo elemento en una localidad diferente del disco, pudiendo mantener así varias copias de un elemento. En general, al valor del elemento antes de la actualización se le llama **imágen anterior (BFIM)**, y al valor después de la actualización se le llama la **imágen posterior (AFIM)**. En la sombra, se guardan el BFIM y el AFIM; por lo tanto, no es estrictamente necesario mantener una bitácora de recuperación.

Cuando se usa la actualización en lugar, es necesario usar una bitácora de recuperación. En este caso, el mecanismo de recuperación debe asegurar que el BFIM del elemento se registra en la entrada de la bitácora adecuada y se limpia a disco antes de que el BFIM sea sobreescrito con el AFIM. A este proceso se le conoce generalmente como **bitácora adelantada**. Antes de poder describir un protocolo de bitácora adelantada, es necesario distinguir entre dos entradas: aquéllas necesarias para el UNDO, y las necesarias para el REDO. Una ***entrada tipo* REDO** es una entrada que corresponde a una operación de escritura de alguna transacción que incluye al nuevo valor (AFIM) del elemento; tal entrada es necesaria si la técnica de recuperación debe *rehacer* el efecto de la operación desde la bitácora colocando el valor del elemento en la base de datos a su AFIM. Las **entradas tipo UNDO** incluyen entradas escribir_elemento que incluyen el valor anterior (BFIM) del mismo; tal entrada es necesaria si la técnica de recuperación debe *deshacer* el efecto de la operación desde la bitácora colocando el valor del elemento en la base de datos de vuelta a su BFIM. Además, cuando es posible el retroceso en cascada, las entradas leer_elemento en la bitácora se consideran entradas tipo UNDO.

Cuando se usa la actualización en el lugar, para permitir la recuperación, antes de aplicar cambios a la base de datos se requiere que las entradas se registren de manera permanente en la bitácora en disco. Por ejemplo, consideremos el siguiente protocolo **registro adelantado (WAL)** para un algoritmo de recuperación que requiere UNDO y REDO:

1. La imágen anterior de un elemento no puede sobreescribirse por su imágen posterior en disco hasta que todos los registros de tipo UNDO para la actualización han sido forzados a escribirse en disco.

2. La operación final de una transacción no puede completarse hasta que todos los registros REDO y UNDO de esa transacción hayan sido forzados a escritura en disco.

Para facilitar el proceso de recuperación, el subsistema de recuperación del DBMS mantiene un número de listas relacionadas a las transacciones que están siendo procesadas en el sistema. Estas incluyen una lista de las **transacciones activas** que han iniciado pero que aún no han finalizado, una lista de todas las **transacciones finalizadas** desde el último chequeo, y una lista de **transacciones abortadas** desde el último chequeo. El mantenimiento de estas listas hace que el proceso de recuperación sea más eficiente.

15.1.3 Retroceso de Transacción

Si una transacción falla por cualquier razón después de actualizar a la base de datos, puede ser necesario **regresar** o **deshacer** la transacción. Cualquier elemento que haya sido modificado por la transacción debe regresar a su valor anterior (BFIM). Las entradas de bitácora se usan para recuperar los valores anteriores de los elementos a regresarse.

Si una transacción T es regresada, cualquier transacción S que tenga que leer el valor de algun elemento X escrito por T también debe regresarse. De manera similar, una vez que S es regresada, cualquier transacción R que haya leído el valor de algun elemento Y escrito

por S también deberá regresarse; y así sucesivamente. A este fenómeno se le llama **retroceso en cascada**, y puede ocurrir cuando el protocolo de recuperación asegure calendarios *recuperables* pero no asegure calendarios *estrictos* o *sin casacada*. El retroceso en cascada, con justa razón, puede ser consumidor de tiempo. Esto es por lo cual la mayoría de los mecanismos están diseñados para que *nunca sea requerido* tal retroceso en cascada.

La Figura 15.1 muestra un ejemplo donde se requiere el retroceso en cascada. Las operaciones de lectura y escritura de tres transacciones individuales se muestran en la Figura 15.1(a). La Figura 15.1(b) muestra la bitácora al momento de la caída del sistema para la ejecución de un calendario en particular de estas transacciones. Los valores de los elementos A, B, C y D los cuales se usan por las transacciones, se muestran a la derecha de las entradas a la bitácora. Supongamos que los valores originales de los elementos, mostrados en la primera línea son A=30, B=15, C=40 y D=20. Al momento de la falla, la transacción T3 no había alcanzado su conclusión y debe regresarse. Las operaciones WRITE de T3, marcadas por un * en la Figura 15.1(b), son las operaciones de T3 que se deshacen durante el retroceso de la transacción. La Figura 15.1(c) muestra gráficamente las operaciones de las diferentes transacciones junto con el eje en el tiempo.

Ahora debemos checar el retroceso en cascada. De la Figura 15.1(c) vemos que la transacción T2 lee el valor del elemento B que fue escrito por la transacción T3; esto puede determinarse también examinando la bitácora. Debido a que T3 es regresada, ahora T2 debe regresarse también. Las operaciones WRITE de T2 marcadas por ** en la bitácora, son las únicas que se deshacen. Notemos que sólo se necesitan deshacer las operaciones escribir_elemento durante el retroceso de la transacción; las operaciones leer_elemento se registran en la bitácora sólo para determinar si es necesario el retroceso en cascada de otras transacciones.

Si *nunca* se requiere del retroceso de transacciones por el método de recuperación (debido a que el método garantiza calendarios sin cascada o estrictos), podemos registrar *infomación más limitada* en la bitácora del sistema. Tampoco existe la necesidad de registrar ninguna operación de leer_elemento en la bitácora, ya que estas sólo son necesarias para determinar el retroceso en cascada.

15.2 Técnicas de Recuperación Basadas en Actualización Diferida

La idea detrás de las técnicas de actualización diferida es posponer cualquier actualización a la base de datos hasta que la transacción finalice su ejecución exitosamente. Durante la ejecución de la transacción, las actualizaciones se registran sólo en la bitácora y en el espacio de trabajo de la transacción. Después de que la transacción alcanza su punto final y se forza a la bitácora a escribir en disco, las actualizaciones se registran en la base de datos misma. Si una transacción falla antes de alcanzar su punto final, no existe la necesidad de deshacer ninguna operación, ya que la transacción no ha afectado a la base de datos de ninguna manera.

Podemos establecer un protocolo de actualización deferido como sigue:

1. Una transacción no puede cambiar a la base de datos hasta que alcanza su punto final.

2. Una transacción no alcanza su punto final hasta que todas sus operaciones de actualización se registren en la bitácora *y* se force su escritura a disco.

Notemos que el paso 2 de este protocolo es una variante del protocolo de registro adelantado (WAL). Ya que la base de datos nunca se actualiza hasta después que la transacción finaliza, no hay necesidad de deshacer ninguna operación. Por lo tanto, a esta técnica se le conoce como el algoritmo **NO-UNDO/REDO**. El REDO es necesario en caso de que el sistema falle después de que la transacción finaliza pero antes de que todos sus cambios sean registrados en la base de datos. En este caso, las operaciones de transacción se rehacen a partir de las entradas en la bitácora.

Usualmente, en sistemas multiusuario el método de recuperación de la falla está muy relacionado con el de control de concurrencia. Primero discutiremos los sistemas de recuperación monousuario, donde no se necesita control de concurrencia, de tal forma que podamos entender el proceso de recuperación independientemente de cualquier método de control de concurrencia. Después discutiremos cómo puede afectar el proceso de recuperación el control de concurrencia.

15.2.1 Recuperación Usando Actualización Diferida en Medio Ambiente Monousuario

En tal medio ambiente, el algoritmo de recuperación puede ser sencillo. El algoritmo RDU_S (recuperación usando actualización diferida en medio ambiente monousuario) usa el procedimiento **REDO**, dado a continuación, para rehacer ciertas operaciones escribir_elemento; funciona como sigue:

PROCEDIMIENTO RDU_S Usa dos listas de transacciones: las transacciones finalizadas desde el último chequeo, y las transacciones activas (al menos una caerá dentro de esta categoría, ya que el sistema es monousuario). Se aplica la operación REDO a todas las operaciones escribir_elemento de las transacciones finalizadas en la bitácora *en el orden en el cual fueron escritas a ella.* Reinicia las transacciones activas.

El procedimiento **REDO** se define como sigue:

REDO(OP_ESCRITURA) Rehacer una operación escribir_elemento consiste en examinar su entrada de bitácora [escribir_elemento, T,X,nuevo_valor] y colocar el valor del elemento X en la base de datos a un nuevo_valor el cual es la imágen posterior (AFIM).

Se requiere que la operación REDO sea **idempotente**; esto es, que ejecutarla una y otra vez sea equivalente a ejecutarla una sola vez. En efecto, el proceso de recuperación

completo debe ser idempotente. Esto se debe, a que si el sistema fuera a fallar durante el proceso de recuperación, el siguiente intento de recuperación podría rehacer ciertas operaciones escribir_elemento que ya habían sido reefectuadas durante el primer proceso de recuperación. El resultado de recuperarse de una caída *durante la recuperación* debe ser el mismo que el resultado de recuperarse *cuando no hay caída durante la recuperación!*

Notemos que la transacción en la lista activa no habrá tenido efecto en la base de datos debido al protocolo de actualización diferida y se ignora completamente por el proceso de recuperación. *Simplemente se regresa*, debido a que ninguna de sus operaciones se reflejaron en la base de datos. Sin embargo, la transacción debe reiniciarse ahora, ya sea automáticamente por el proceso de recuperación o manualmente por el usuario.

La Figura 15.2 muestra un ejemplo de recuperación en un medio ambiente monousuario, donde la primer falla ocurre durante la ejecución de la transacción T2, como se muestra en la Figura 15.2(b). Un proceso de recuperación rehará la entrada [escribir_elemento, T1,D,20] en la bitácora reseteando el valor del elemento D a 20 (su nuevo valor). Las entradas [write,T2,...] en la bitácora se ignoran por el proceso de recuperación ya que T2 no ha finalizado. Si ocurre una segunda falla durante la recuperación de la primera, se repite el mismo proceso desde el inicio hasta el final, con resultados idénticos.

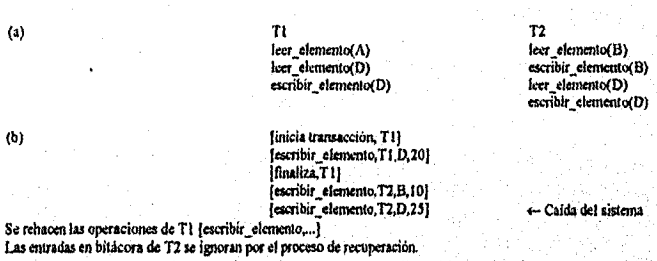


Figura 15.2 Recuperación usando actualización diferida en un medio ambiente monousuario. (a) Las operaciones de lectura y escritura de dos transacciones. (b) Bitácora del sistema al momento de la caída.

15.2.2 Actualización Diferida con Ejecución Concurrente en un Medio Ambiente Multiusuario

Para sistemas multiusuario con control de concurrencia, el proceso de recuperación puede ser más complejo, dependiendo de los protocolos usados para el control de concurrencia. En muchos casos, los procesos de control de concurrencia y de recuperación son relacionadas. En general, mientras más grado de concurrencia se quiera lograr, más difícil se vuelve la tarea de recuperación.

Consideremos un sistema en el cual el control de concurrencia usa bloqueo de dos fases y previene el deadlock preasignando todos los bloqueos a elementos necesarios por una transacción antes de que inicie se ejecución. Para combinar el método de actualización

diferido con esta técnica de control de concurrencia, podemos registrar todos los bloqueos sobre elementos en efecto *hasta que la transacción alcance su punto final.* Después de ello, pueden liberarse los bloqueos. Esto asegura calendarios estrictos y serializables. Asumir que se incluyan entradas en la bitácora, un posible algoritmo de recuperación para este caso, al cual llamamos RDU_M (recuperación usando actualización deferida en un medio ambiente multiusuario). Este procedimiento usa el REDO.

PROCEDIMIENTO RDU_M Usa dos listas mantenidas por el sistema: las transacciones finalizadas **T** desde el último chequeo, y las transacciones activas **T'**. REHACER todas las operaciones WRITE de las transacciones finalizadas en la bitácora, *en el orden en el cual fueron escritas a ella.* Las transacciones activas y las que no finalizaron se cancelan y deben resubmitirse.

La Figura 15.3 muestra un calendario posible de la ejecución de transacciones. Cuando fue tomado el chequeo en t1, la transacción T1 ha finalizado, mientras que las transacciones T3 y T4 no. Antes de que el sistema caiga en t2, T3 y T2 finalizaron pero no T4 y T5. De acuerdo al método RDU_M, no hay necesidad de rehacer las operaciones escribir_elemento de la transacción T1 -o cualquier transacción finalizada antes del último chequeo t1. Las operaciones escribir_elemento de T2 y T3 deben rehacerse, debido a que ambas transacciones alcanzaron sus puntos finales después del último chequeo. Recordemos que la bitácora se forza a escritura antes de finalizar una transacción. Las transacciones T4 y T5 se ignoran: se cancelan o retroceden debido a que ninguna de sus operaciones escribir_elemento fueron registradas en la base de datos bajo el protocolo de actualización deferida.

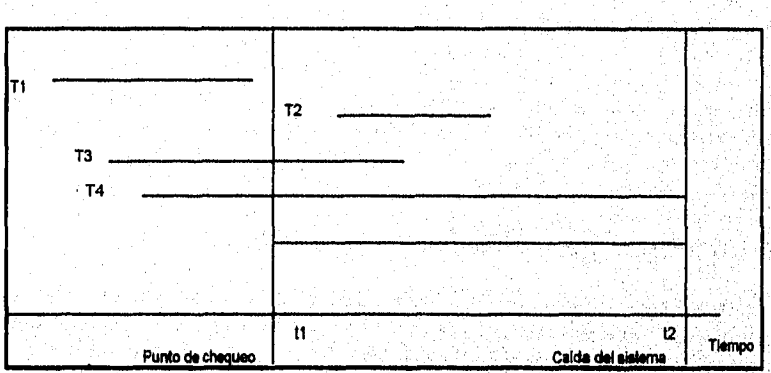


Figura 15.3 Ejemplo de recuperación en un medio ambiente multiusuario.

Podemos hacer más eficiente el algoritmo NO-UNDO/REDO *más eficiente* notando que, si un elemento X ha sido actualizado más de una vez por transacciones finalizadas, sólo es necesario rehacer *la última actualización de X* de la bitácora durante la recuperación, ya que en cualquier caso las otras actualizaciones serían sobreescritas por este último REDO. En este caso, iniciamos desde *el fin de la bitácora*; luego, cada que un elemento se rehace, se agrega a una lista de elementos rehechos. Antes de que REDO se aplique a un

elemento, se checa la lista; si el elemento aparece en la lista, no se rehace, ya que su último valor ya ha sido recuperado.

Una desventaja del método descrito es que limita la ejecución actual de transacciones porque *todos los elementos permanecen bloqueados hasta que la transacción alcance su punto final.* El principal beneficio del método es que las operaciones de la transacción nunca se deshacen, por dos razones:

1. Una transacción no registra sus cambios en la base de datos hasta que alcanza su punto final -esto es, hasta completar su ejecución exitosamente. Por lotanto, una transacción nunca retrocede debido a una falla en la ejecución de la transacción.

2. Una transacción nunca leerá el valor de un elemento que se escribe por una transacción sin finalizar, debido a que los elementos permanecen bloqueados hasta que una transacción alcanza su punto final. Por lo tanto, no ocurrirá un retroceso en cascada.

La Figura 15.4 muestra un ejemplo de recuperación para un sistema multiusuario que utiliza el método de recuperación y concurrencia recién descrito.

15.2.3 Acciones en la Transacción que No Afectan a la Base de Datos

En general, una transacción tendrá acciones que no afectan a la base de datos, tal como la generación e impresión de mensajes o reportes a partir de información recuperada de la base de datos. Si una transacción falla antes de finalizar, podemos desear que el usuario no obtenga estos reportes, ya que la transacción ha fallado en completarse. Por lotanto, tales reportes deben generarse sólo *después de que la transacción alcanza su punto final.* Un método común de trabajar con tales acciones es liberar los comandos que generan reportes pero mantenerlos como tareas batch. Las tareas batch se ejecutan sólo después de que la transacción alcanza su punto final. Si la transacción no alcanza su punto final debido a una falla, se cancelan las tareas batch.

15.3 Técnicas de Recuperación Basadas en Actualización Inmediata

En estas técnicas, cuando una transacción libera un comando de actualización, la base de datos puede actualizarse "inmediatamente", sin necesidad de esperar que la transacción alcance su punto final. En muchas de estas técnicas, una operación de actualización debe seguir registrada en la bitácora (en disco) *antes* de que se aplique a la base de datos de tal forma que se pueda recuperar, en caso de falla, usando un protocolo de registro de escritura adelantada.

(a)

T1	T2	T3	T4
leer_elemento(A)	leer_elemento(B)	leer_elemento(A)	leer_elemento(B)
leer_elemento(D)	escribir_elemento(B)	escribir_elemento(A)	escribir_elemento(B)
escribir_elemento(D)	leer_elemento(D)	leer_elemento(C)	leer_elemento(A)
	escribir_elemento(D)	escribir_elemento(C)	escribir_elemento(A)

(b)

[inicia_transaccion, T1]	
[escribir_elemento,T1, D,20]	
[finaliza,T1]	
[punto chequeo]	
[inicia_transacción,T4]	
[escribir_elemento,T4, B,15]	
[escribir_elemento,T4, A,20]	
[finalizar,T4]	
[inicia_transacción,T2]	
[escribir_elemento,T2, B,12]	
[inicia_transacción,T3]	
[escribir_elemento,T3, A,30]	
[escribir_elemento,T2, D,25]	← Caída del sistema

T2 y T3 se ignoran porque no alcanzaron su finalización.
T4 se rehace porque su punto de finalización es después del chequeo del sistema.

Figura 15.4 Recuperación usando actualización diferida con transacciones concurrentes. (a) Operaciones de lectura y escritura de cuatro transacciones. (b) Bitácora del sistema al momento de la caída.

Cuando se permite la actualización inmediata, deben tomarse provisiones para *deshacer* el efecto de operaciones de actualización sobre la base de datos, debido a que una transacción puede fallar después de que ha aplicado algunas actualizaciones a la base de datos. Por lo tanto, los esquemas de recuperación basados en actualización inmediata deben incluir la capacidad de retroceder una transacción deshaciendo el efecto de sus operaciones escribir_elemento.

En general, podemos distinguir dos categorías principales de algoritmos de actualización. Si la técnica de recuperación asegura que todas las actualizaciones de una transacción se registran en la base de datos en disco *antes de que la transacción finalice*, nunca habrá necesidad de **REHACER** ninguna operación de transacciones finalizadas. A tal algoritmo se le llama **UNDO/NO-REDO**. Por otro lado, si se le permite finalizar a la transacción antes de que todos sus cambios sean escritos a la base de datos, teniendo al algoritmo más general de recuperación, conocido como **UNDO/REDO**. Esta es la técnica más compleja.

15.3.1 Recuperación UNDO/REDO Basada en Actualización Inmediata en un Medio Ambiente Monousuario

Primero consideremos un sistema monousuario de tal forma que podamos examinar el proceso de recuperación separado del control de concurrencia. Si ocurre una falla en un sistema monousuario, la transacción en ejecución al momento de la falla puede haber registrado algunos cambios en la base de datos. El efecto de tales operaciones debe deshacerse como parte del proceso de recuperación. Por lo tanto, el algoritmo de recuperación necesita un procedimiento UNDO, para deshacer el efecto de ciertas

operaciones escribir_elemento que se han aplicado a la base de datos siguiendo la examinación de su bitácora del sistema. El algoritmo de recuperación RIU_S (recuperación usando actualización inmediata en un medio ambiente monousuario) también usa el procedimiento REDO.

PROCEDIMIENTO RIU_S

1. Usar dos listas de transacciones mantenidas por el sistema; las transacciones finalizadas desde el último chequeo, y las transacciones activas.

2. Deshacer todas las operaciones escribir_elemento de la transacción *activa* de la bitácora, usando el procedimiento *UNDO*.

3. Rehacer todas las operaciones escribir_elemento de las transacciones *finalizadas* de la bitácora, en el orden en el cual fueron escritas, usando el procedimiento REDO.

El procedimiento *UNDO* se define como sigue:

UNDO (OP_ESCRITURA) Deshacer una operación escribir_elemento consiste en examinar la bitácora [escribir_elemento,T,X,valor_anterior,nuevo_valor] y poner el valor del elemento X en la base de datos al valor_anterior el cual es su imágen anterior (BFIM). Deshacer operaciones escribir_elemento de una o más transacciones de la bitácora debe proceder en el *orden inverso* en el cual se escribieron las operaciones a la bitácora.

15.3.2 Actualización Inmediata UNDO/REDO con Ejecución Concurrente

Cuando se permite la ejecución concurrente, el proceso de recuperación depende de los protocolos usados para el control de concurrencia. El procedimiento RIU_M (recuperación usando actualizaciones inmediatas para un medio ambiente multiusuario) indica una técnica de recuperación para transacciones concurrentes con actualización inmediata. Supongamos que la bitácora incluye chequeos y que el protocolo de control de concurrencia produce *calendarios estrictos* -como por ejemplo, el protocolo estricto de bloqueo de dos fases. Un calendario estricto no permite que una transacción lea o escriba un elemento a menos que la transacción que escribió por última vez al elemento haya finalizado. Sin embargo, pueden ocurrir deadlocks en el bloqueo estricto de dos fases, requiriendo así deshacer las transacciones. Para un calendario estricto, deshacer una operación requiere cambiar al elemento a su valor anterior (BFIM).

PROCEDIMIENTO RIU_M

1. Usar dos listas de transacciones mantenidas por el sistema: las transacciones finalizadas desde el último chequeo y las transacciones activas.

2. Deshacer todas las operaciones escribir_elemento de las transacciones *activas* (no finalizadas), usando el procedimiento UNDO. Las operaciones deben deshacerse en el orden inverso en el cual fueron escritas a la bitácora.
3. Rehacer todas las operaciones escribir_elemento de las transacciones *finalizadas* desde la bitácora, en el orden en el cual fueron escritas.

15.4 Paginación Shadow

Este esquema de recuperación no requiere el uso de una bitácora en un medio ambiente monousuario. En un ambiente multiusuario, se puede requerir de una bitácora si es que es necesaria para el método de control de concurrencia. La paginación shadow considera a la base de datos conformada por páginas de disco de tamaño fijo (o bloques de disco) -digamos, n- para propósitos de recuperación. Se construye una **tabla página** (o **directorio**) con n entradas, donde la iésima entrada de la tabla página apunta a la iésima página en el disco. La tabla página se mantiene en memoria principal si no es demasiado grande, y todas las referencias -lecturas o escrituras- a páginas de la base de datos en disco pasan a través de la tabla página. Cuando una transacción inicia su ejecución, la **tabla página actual** -cuyas entradas apuntan a las páginas más recientes o actuales en disco- se copia a la **tabla página shadow**. La tabla página shadow se guarda en disco mientras que la tabla de la página actual se usa por la transacción.

Durante la ejecución de la transacción, la tabla página shadow *nunca* se modifica. Cuando se efectúa una operación escribir_elemento, se crea una copia nueva de la página de la base de datos modificada, pero la copia anterior de esa página *no se sobreescribe*. En su lugar, la nueva página se escribe en otra parte -en algún bloque de disco previamente inutilizado. La entrada de la página actual se modifica para apuntar al nuevo bloque en disco, mientras que la tabla página sombra no se modifica y continúa apuntando al bloque anterior sin modificación. La Figura 15.5 ilustra los conceptos de una tabla página shadow y una tabla página actual. Para las páginas actualizadas por transacción, se guardan dos versiones. La versión anterior es referenciada por la tabla página shadow, y la nueva versión por la tabla página actual.

Para recuperarse de una falla durante la ejecución de una transacción, es suficiente liberar las páginas modificadas de la base de datos y descartar a la página actual. El estado de la base de datos antes de la ejecución de la transacción está disponible a través de la página tabla shadow, y que el estado se recupera reinstando a la tabla página shadow de tal forma que se convierta en la página actual una vez más. Así la base de datos retorna a su estado previo a la transacción que estaba ejecutándose cuando ocurrió la falla, y cualquier página modificada se descarta. La finalización de una transacción corresponde a descartar la tabla página shadow anterior y liberar las tablas página anteriores que referencia. Debido a que la recuperación no involucra ni deshacer ni rehacer elementos, esta técnica puede categorizarse como NO-UNDO/NO-REDO.

La ventaja de la paginación shadow es que facilita el deshacer el efecto de la transacción ejecutada. No hay necesidad de deshacer o rehacer ninguna operación de las

transacciones. En un medio ambiente multiusuario con transacciones concurrentes, deben incorporarse bitácoras y chequeos a la técnica de paginación shadow. Una desventaja de la paginación shadow es que las páginas de la base de datos actualizadas cambian de localidad en el disco. Esto dificulta el mantener relacionadas a las páginas de la base de datos en disco sin complejas estrategias de manejo de almacenamiento. Además, si la tabla página es grande, es significativa la desventaja de escribir tablas páginas shadow a disco como transacciones finalizadas. Una mayor complicación es cómo manejar la **colección de basura** cuando una transacción finaliza. Las páginas anteriores referenciadas por la página shadow que han sido actualizadas deben liberarse y agregarse a una lista de páginas libres para uso futuro. Después de que la transacción finaliza, estas páginas no son más necesarias, y la tabla página actual reemplaza a la tabla página shadow para convertirse en la tabla página válida.

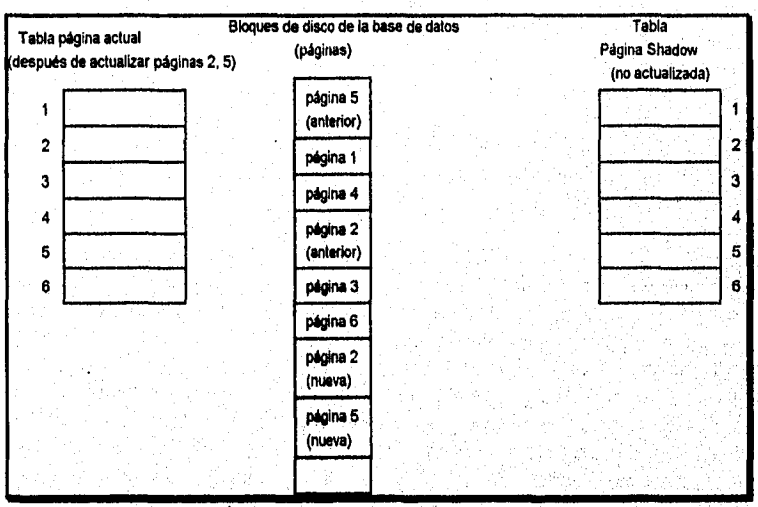


Figura 15.5 Paginación shadow.

15.5 Transacciones de Recuperación en Multibases de Datos

Hasta ahora, hemos asumido implícitamente que una transacción accesa a una sola base de datos. En algunos casos una sola transacción, llamada **transacción multibase**, puede requerir el acceso a varias bases de datos. Aún estas bases de datos pueden almacenarse en diferentes tipos de DBMSs; por ejemplo, algunos DBMSs pueden ser relacionales, mientras que otros jerárquicos o de red, En tal caso, cada DBMS involucrado en la transacción multibase tendrá su propia técnica de recuperación y manejo de transacciones independiente de los demás. Esta situación es algo similar al caso del manejador de bases de datos distribuídas, donde parte de la base de datos reside en diferentes sitios que se conectan a través de una red de comunicaciones.

Para mantener la atomicidad de una transacción multibase, es necesario tener un mecanismo de recuperación a dos niveles. Es necesario un **manejador de recuperación global**, o **coordinador**, además de los manejadores de recuperación local. El coordinador usualmente sigue un protocolo llamado **protocolo de finalización de dos fases**, cuyas dos fases pueden establecerse como sigue:

FASE 1: Cuando todas las bases participantes señalan al coordinador que la parte de la transacción multibase que involucra a cada una ha finalizado, el coordinador envía un mensaje "prepararse para finalizar" a cada participante para alistarlo de la finalización de la transacción. Cada base de datos participante recibe un mensaje que forzará la escritura de la bitácora de todos los registros a disco y luego enviará una señal "listo para finalizar" u "OK" al coordinador. Si la escritura forzada en disco falla o por alguna razón la transacción local no puede finalizar, la base participante envía una señal "no puede finalizar" o "No OK" al coordinador. Si el coordinador no recibe una respuesta de una base de datos dentro de cierto intervalo de tiempo, asume una respuesta "No OK".

FASE II: Cuando todas las bases participantes responden "OK", la transacción es exitosa, y el coordinador envía una señal de "finalización" para la transacción a las bases participantes. Debido a que todos los efectos locales de la transacción han sido registrados en las bitácoras de las bases participantes, ahora es posible la recuperación de una falla. Cada base de datos finaliza una transacción escribiendo una entrada [finalización] para la transacción en la bitácora y de ser necesario actualizando permanentemente a la base de datos. Por otro lado, si una o más de las bases de datos participantes tienen una respuesta de "No OK" al coordinador, la transacción ha fallado, y el coordinador envía un mensaje de "retroceder" o DESHACER el efecto local de la transacción a cada base participante. Esto se logra deshaciendo las operaciones de la transacción, usando la bitácora.

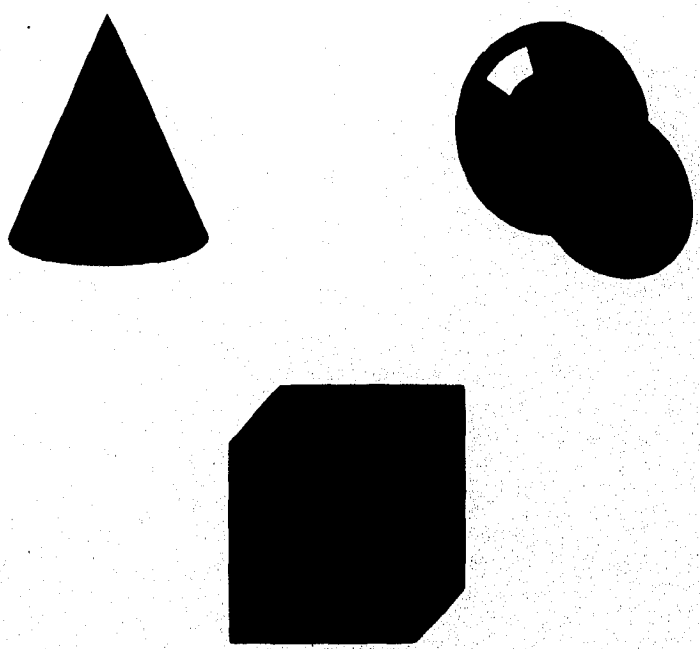
El efecto neto del protocolo de finalización de dos fases es que todas las bases participantes finalizan el efecto de la transacción o ninguna de ellas lo hace. En caso de que cualquiera de los participantes -o el coordinador- falle, siempre es posible recuperar a un estado donde cualquier transacción finalice o sea retrocedida. Una falla durante o antes de la fase 1 usualmente requiere que la transacción sea retrocedida, mientras que una falla en la fase 2 significa que una transacción exitosa puede recuperarse y finalizarla.

15.6 Respaldo y Recuperación de Fallas Catastróficas

Hasta ahora, todas las técnicas que hemos discutido se aplican a fallas no catastróficas. Se ha supuesto que se mantiene la bitácora del sistema en el disco y que no se pierde como resultado de una falla. De manera similar, la tabla página shadow debe almacenarse en disco para permitir la recuperación cuando se use paginación shadow. Las técnicas de recuperación que hemos discutido usan las entradas en la bitácora del sistema o tabla página shadow para recuperarse de la falla regresando a la base de datos a un estado consistente.

El manejador de recuperación de un DBMS debe estar equipado para el manejo de fallas más catastróficas como destrucción de discos. La técnica principal para manejar tales desperfectos es el **respaldo de la base de datos**. La base de datos completa y la bitácora se copian periódicamente a medios de almacenamiento baratos tales como cintas magnéticas. La bitácora del sistema es usualmente mucho más pequeña que la base de datos y por lo tanto puede respaldarse más frecuentemente.

CAPITULO 16



Bases de Datos Orientadas a Objetos

16.1 Panorama de los Conceptos Orientados a Objetos

El término **orientado a objeto** -abreviado OO- tiene sus orígenes en los lenguajes de programación orientados a objetos. Los conceptos OO se aplican ahora en áreas de bases de datos, ingeniería de software, inteligencia artificial, y sistemas de cómputo en general. Los lenguajes de programación OO tienen sus raíces en el lenguaje SIMULA, el cual fue propuesto a finales de los 60's. En SIMULA, el concepto de *clase* junto con la estructura interna de un objeto en una declaración de clase. Subsecuentemente, los investigadores propuesieron el concepto de *tipo de dato abstracto*, el cual esconde las estructuras de datos y especifica todas las operaciones externas posibles que pueden aplicarse a un objeto, llevando al concepto de *encapsulación*. El lenguaje de programación SMALLTALK, desarrollado en XERORX PARK en los 70's fue uno de los primeros lenguajes en incorporar explícitamente conceptos OO adicionales, tales como el paso de mensajes y herencia. Se conoce como un lenguaje de programación OO *puro*, significando que fue explícitamente diseñado para ser orientado a objetos. Esto contrasta con los lenguajes de programación *hibridos*, los cuales incorporan conceptos OO a un lenguaje ya existente. Un jemplo de este último es C++, el cual incorpora conceptos OO al popular lenguaje de programación C.

Los objetos en un lenguaje de programación OO existen sólo durante la ejecución del programa. Una BDOO proporciona capacidades de tal forma que los objetos puedan crearse para existir permanentemente, o *persistir*, y ser compartidos por numerosos programas. Por lo tanto, BDOO almacenan *objetos persistentes* de forma permanente en almacenamiento secundario, y permiten compartir estos objetos entre múltiples programas y aplicaciones. Esto requiere de otras características bien conocidas de sistemas administradores de bases de datos, tales como los mecanismos de indexado, control de concurrencia, y recuperación. Un sistema BDOO se interfaza con uno o más lenguajes de programación para proporcionar capacidades de objetos persistentes y compartidos.

Una meta de BDOO es mantener una correspondencia directa entre el mundo real y los objetos de la base de datos de tal manera que los objetos no pierdan su integridad e identidad y puedan ser identificados y operados fácilmente. Por lo tanto, BDOO proporcionan un *identificador de objetos* único generado por el sistema (OID) para cada objeto. Podemos comparar esto con el modelo relacional, por ejemplo, donde cada relación debe tener una llave primaria cuyo valor identifique a cada tupla de manera única. En el modelo relacional, si cambia el valor de la llave primaria, la tupla tendrá una nueva identidad, aún a apesar de que siga representando a la misma identidad del mundo real. Alternativamente, un objeto del mundo real puede poseer diferentes llaves en diferentes relaciones, haciendo difícil asertar que las llaves verdaderamente son del mismo objeto (por ejemplo, el identificador de objeto puede estar representado como Emp_id en una relación y como Imss en otra).

Otra cataracterística de BDOO es que los objetos pueden tener una *estructura de complejidad arbitraria* que contiene toda la información significativa que describe al objeto. En contraste, en los sistemas de bases de datos tradicionales, la información acerca

de un objeto complejo se *esparce* sobre muchas relaciones de registros, llevando a la pérdida de correspondencia directa entre un objeto del mundo real y su representación en la base de datos.

La estructura interna de un objeto incluye la especificación de *variables de instancia*, las cuales retienen los valores que definen el estado interno del objeto. Por lo tanto, una variable de instancia es similar al concepto de *atributo*, excepto que las variables de instancias pueden encapsularse dentro del objeto y así no ser necesariamente visibles a usuarios externos. Las variables instancia también pueden ser tipos de datos arbitrariamente complejos. Los sistemas orientados a objetos permiten la definición de operaciones o funciones que pueden aplicarse a objetos de un tipo particular. En efecto, algunos modelos OO insisten que todas las operaciones que un usuario puede aplicar a un objeto deben estar predefinidas. Esto forza a una *encapsulación* completa de los objetos. Este acercamiento rígido ha sido disminuido en la mayoría de modelos OO, debido a que implica que cualquier recuperación sencilla requiere de una operación predefinida.

Para llevar a cabo la encapsulación, se define una operación en dos partes. A la primera se le llama la *firma o interface*, especifica el nombre de la operación y argumentos (o parámetros). La segunda, llamada el *método* o *cuerpo*, especifica la *implementación* de la operación. Las operaciones pueden invocarse pasando un *mensaje* a un objeto, el cual incluye el nombre de la operación y los parámetros. El objeto ejecuta entonces el método para esa operación. Esta encapsulación permite la modificación de la estructura interna de un objeto y la implementación de sus operaciones sin la necesidad de distinguir los programas externos que invocan a estas operaciones. Por lo tanto, la encapsulación proporciona para una forma de dato e independencia de operación.

Otro concepto clave en sistemas OO involucra la representación de *relaciones* entre objetos. La insistencia sobre la encapsulación completa en los primeros modelos de datos OO llevó al argumento de que las relaciones no deben representarse explícitamente, en lugar de ello deben ser descritas definiendo métodos apropiados que localicen a los objetos relacionados. Sin embargo, este acercamiento no trabaja muy bien para bases de datos complejas con relaciones múltiples, debido a que es útil identificar estas relaciones y hacerlas visibles a los usuarios. Muchos modelos de datos OO no permiten la representación de relaciones a través de referencias -esto es, colocando los OID de objetos relacionados dentro de un mismo objeto.

Algunos sistemas OO proporcionan capacidades de trabajar con *múltiples versiones* del mismo objeto -una característica que es esencial en diseño y aplicaciones de ingeniería. Por ejemplo, una versión antigua de un objeto que representa un diseño probado y verificado debe retenerse hasta que se prueba y verifica la nueva versión. Una nueva versión del objeto complejo puede incluir sólo unas cuantas versiones nuevas de sus objetos componentes, mientras que otros componentes permanecen sin cambio. Además de permitir el versionamiento, las BDOO deben permitir idealmente un *esquema de evolución*, lo cual ocurre cuando se cambian declaraciones de tipo o cuando se crean nuevos tipos o relaciones.

Otro concepto OO es el *operador de polimorfismo*, el cual se refiere a una capacidad de operación que se aplica a diferentes tipos de objetos; en tal situación, una *operación nombre* puede referir a varias *implementaciones* distintas, dependiendo del tipo de objetos al que se aplica. A esta característica también se le llama *operador de sobrecarga*. Por ejemplo, una operación para calcular el área de un objeto geométrico puede diferir en su método (implementación), dependiendo si el objeto es de tipo triángulo, círculo o rectángulo. Esto puede requerir el uso de *enlace retardado* del nombre de la operación al método apropiado al momento del run-time, cuando el tipo de objeto al que se aplica la operación se vuelve conocido.

16.2 Identidad, Estructura de Objeto, y Tipos de Constructores

En esta sección discutiremos el concepto de identidad de objeto, y presentaremos las operaciones típicas de estructurado para definir el valor de la estructura de un objeto. Estas operaciones se llaman con frecuencia **constructores de tipo**. Definen las operaciones básicas de estructurado que combinadas pueden formar objetos de complejidad arbitraria.

16.2.1 Identidad del Objeto

Un sistema de BDOO proporciona una **identidad única** a cada objeto independiente almacenado en la base de datos. Esta identidad única se implementa a través de un **identificador de objetos** único, generado por el sistema, u OID. El valor de un OID no es visible para el usuario externo, pero se usa internamente por el sistema para identificar a cada objeto de manera única y crear y manejar referencias entre objetos.

La principal propiedad requerida por un OID es el ser **inmutable**; esto es, el valor de un OID de un objeto en particular no debe cambiar. Esto preserva la identidad del objeto del mundo real que está siendo representado. También es deseable que cada OID se use una sóla vez; esto es que, aún si el objeto es eliminado de la base de datos, su OID no se asigne a otro objeto. Estas dos propiedades implican que el OID no debe depender de ningún valor de atributo del objeto, debido a que el valor del atributo puede cambiar. También se considera inapropiado basar el OID en la dirección física de almacenamiento del objeto, debido a que una reorganización física de los objetos de la base de datos podría cambiar a los OIDs. Sin embargo, algunos sistemas deben usar la dirección física como OID para incrementar la eficiencia de recuperación del objeto. Si cambia la dirección física, puede colocarse un *apuntador indirecto* en la dirección anterior, lo cual da la nueva localidad física del objeto. Un sistema de BDOO debe de tener algún mecanismo de generar OIDs con la propiedad de inmutabilidad.

Algunos modelos OO requieren que todo se represente como objeto, ya sea como un sólo valor o un objeto complejo; por lo tanto, cada valor básico, tal como un entero, cadena, o valor Booleano, tiene un OID. Esto permite que dos valores básicos tengan distintos OIDs, lo cual puede ser útil en algunos casos. Por ejemplo, el valor entero 50 puede usarse algunas veces como peso en kilogramos, y otras como la edad de una persona.

Entonces, pudieran crearse dos objetos básicos con distintos OIDs, y ambos objetos tendrían el mismo valor básico de 50. Aún cuando es útil como modelo teórico, no es muy práctico, debido a que puede llevar a la generación de demasiados OIDs. Por tanto, la mayoría de los sistemas de BDOO permiten la representación de objetos y **valores**. Cada objeto debe tener un OID inmutable, mientras que un valor no tiene OID y está el solo.

16.2.2 Estructura de Objetos

En las BDOO, pueden construirse los valores (o estados) de objetos complejos a partir de otros objetos usando ciertos **constructores de tipo**. Una forma de representar tales objetos es visualizarlos como un triple (i, c, v) donde i es un *identificador de objeto* único (el OID), c es un *constructor* (esto es, un indicador de cómo se construye el valor del objeto), y v es el *valor* del objeto (o estado). Dependiendo del modelo de datos o del sistema OO, pueden haber varios constructores. Los tres constructores básicos son **átom, tuple** y **set**. Otros constructores comúmente usados incluyen **list** y **array**. También existe un **dominio** D que contiene todos los valores atómicos que están directamente disponibles en el sistema. Estos incluyen típicamente enteros, números reales, cadenas de caracteres, Booleanos, fechas, y otros tipos de datos que el sistema soporte directamente.

Un valor de objeto v se interpreta en base al valor del constructor c en la tripleta (i, c, v) que representa al objeto. Si c = atom, el valor de v es atómico del dominio D de valores básicos soportados por el sistema. Si c = set, el valor de v es un conjunto de identificadores $\{i_1, i_2, ..., i_n\}$, los cuales son los identificadores (OIDs) para un conjunto de objetos que son físicamente del mismo tipo. Si c = tupla, el valor de v es una tupla de la forma $<a_1:i_1, a_2:i_2, ..., a_n:i_n>$ donde cada ai es el nombre de un atributo (algunas veces llamado *nombre de variable de instancia* en terminología OO) y cada ij es un identificador de objeto (OID). Si c = list, el valor de v es una *lista organizada* de identificadores de objetos $[i_1, i_2, ..., i_n]$ del mismo tipo. Para c = array, el valor es un arreglo de identificadores de objetos.

El modelo anterior permite el anidamiento arbitrario de set, listas, tuplas, y otros constructores. Todos los valores en objetos no atómicos se refieren a otros objetos por sus identificadores. Por lo tanto, el único caso donde un valor actual aparece es en los objetos atómicos. No es práctico generar un OID para cada valor, así la mayoría de los sistemas permiten OIDs y *valores estructurados*. Los valores pueden estructurarse usando el mismo tipo de constructores como objetos, excepto cuando el valor *no tiene* OID.

A los constructores de tipo **set, list, array** y **bag** se les llama **colección de tipos** o **volúmen de tipos**, para distinguirlos de los tipos básicos y tuplas. Una lista es similar a un conjunto excepto que los OID de la lista están *ordenados*, y podemos referirnos al primero, segundo, o iésimo objeto de la lista. Una bolsa también es similar a un conjunto, excepto que permite la existencia de valores duplicados. La principal característica de una colección de tipos es que un valor será una *colección de objetos* que pueden ser no estructurados (tales como un conjunto o bolsa) o estructurados (tales como una lista o arreglo).

Supongamos que en el siguiente ejemplo todo es un objeto, incluyendo los valores básicos. También asumamos que tenemos los constructores atom, set y tupla. Ahora representaremos algunos objetos de la base de datos relacional de la Figura 6.6, usando el modelo precedente. Usaremos i_1, i_2, i_3... para establecer identificadores únicos generados por el sistema. Un objeto se define por la tripleta (OID, constructor de tipo, valor). Consideremos los siguientes objetos:

o_1 = (i_1, atom, Houston)

o_2 = (i_2, atom, Bellaire)

o_3 = (i_3, atom, Sugarland)

o_4 = (i_4, atom, 5)

o_5 = (i_5, atom, Investigación)

o_6 = (i_6, atom, 22-MAY-78)

o_7 = (i_7, set, {i_1, i_2, i_3})

o_8 = (i_8, tupla, <DNOMBRE:i_5, DNUMERO:i_4, GTE:i_9, LOC:i_7, EMPLEADOS:i_{10}, FECINIGTE:i_6>)

o_9 = (i_9, tupla, <GERENTE:i_{12}, FECINIGTE:i_6>)

o_{10} = (i_{10}, set{i_{12}, i_{13}, i_{14}})

o_{11} = (i_{11}, set{i_{15}, i_{16}, i_{17}})

Los primeros cinco objetos (o_1-o_5) listados son valores atómicos. Existen muchos objetos similares, uno para cada distinto valor atómico constante en la base de datos. Estos objetos atómicos son los que pueden ocasionar problemas, debido al excesivo uso de OIDs, si este modelo se implementa de manera directa. El objeto o_7 es un objeto conjunto valuado que representa el conjunto de localidades del departamento 5; el conjunto {i_1, i_2, i_3} se refiere a los objetos atómicos con valores {Houston, Bellaire, Sugarland}. El objeto o_8 es una tupla valuada que representa al departamento 5, y tiene los atributos DNOMBRE, DNUMERO, GTE, LOC, etc. Los primeros dos atributos DNOMBRE y DNUMERO tienen a los objetos atómicos o_5 y o_4 como sus valores. El atributo GTE tiene a la tupla o_9 como su valor, el cual en cambio tiene dos atributos. El valor del atributo gerente es el objeto cuyo OID es i_{12}, el cual es el objeto empleado que maneja al departamento, mientras que el valor de FECINIGTE es otro objeto atómico cuyo valor es una fecha. El valor del atributo EMPLEADOS de o_8 es un conjunto de objetos con OID = i_{10}, cuyo valor es el conjunto de OIDs para los empleados que trabajan para el DEPTO (objetos i_{12}, i_{13}, i_{14}). De manera similar, el valor del atributo PROYECTOS de o_8 es un

conjunto de objetos con OID = i_{11}, cuyo valor es el conjunto de OIDs de los proyectos que están controlados por el departamento número 5 (objetos i_{15}, i_{16}, i_{17}).

En este modelo, un objeto puede representarse como una estructura gráfica, debido a que puede construirse aplicando repetidamente los tres constructores básicos. La gráfica que representa a un objeto o_i puede construirse creando primero un nodo para el objeto mismo. Crear un nodo para cada valor básico en el dominio D de todos los valores atómicos. Si el objeto tiene un valor atómico, dibujamos un arco dirigido del nodo que representa a o_i al nodo que representa su valor básico. Si el valor del objeto es construido, dibujamos arcos dirigidos del nodo objeto a un nodo que represente el valor construido. En general, no debe tener ciclos la gráfica de un *objeto individual*, ya que indicaría que un objeto se contiene a sí mismo como componente. Sin embargo, la gráfica que representa a un *tipo de objetos* puede tener ciclos que representen relaciones recursivas. Por ejemplo, un objeto EMPLEADO puede referirse a su SUPERVISOR, el cual también es de tipo EMPLEADO. Sin embargo, en la gráfica de objetos individuales EMPLEADO, el supervisor debe ser un objeto empleado distinto, y por lo tanto no existir ciclos. La Figura 16.1 muestra la gráfica del objeto ejemplo DEPTO dado anteriormente.

El modelo precedente permite dos tipos de definiciones en una comparasión de igualdad de los valores de dos objetos. Se dice que dos objetos tienen **valores idénticos** si las gráficas que representan sus valores son idénticas en cada aspecto, incluyendo los OIDs en cada nivel. Otra definición débil de igualdad es cuando dos objetos tienen **valores iguales**. En este caso, las estructuras gráficas deben mostrar ser iguales, y todos los valores atómicos correspondientes en las gráficas también deben ser iguales. Sin embargo, algunos nodos internos correspondientes a las dos gráficas pueden tener objetos con diferentes OIDs.

Para comparar la igualdad de objetos un ejemplo simple puede ilustar la diferencia entre las dos definiciones. Consideremos los siguientes objetos o_1, o_2, o_3, o_4, o_5 y o_6:

o_1 = (i_1, tupla, <a_1:i_4, a_2:i_6>)
o_2 = (i_2, tupla, <a_1:i_5, a_2:i_6>)
o_3 = (i_3, tupla, <a_1:i_4, a_2:i_6>)
o_4 = (i_4, atom, 10)
o_5 = (i_5, atom, 10)
o_6 = (i_6, atom, 20)

Los objetos o1 y o2 tienen valores *iguales*, debido a que sus valores a nivel atómico son los mismos pero los valores son alcanzados a través de distintos objetos o4 y o5. Sin embargo, los valores de los objetos o1 y o2 son *idénticos*. Similarmente, o4 y o5 son iguales pero no idénticos, debido a que tienen diferentes OIDs.

conjunto de objetos con OID = i_{11}, cuyo valor es el conjunto de OIDs de los proyectos que están controlados por el departamento número 5 (objetos i_{15}, i_{16}, i_{17}).

En este modelo, un objeto puede representarse como una estructura gráfica, debido a que puede construirse aplicando repetidamente los tres constructores básicos. La gráfica que representa a un objeto o_i puede construirse creando primero un nodo para el objeto mismo. Crear un nodo para cada valor básico en el dominio D de todos los valores atómicos. Si el objeto tiene un valor atómico, dibujamos un arco dirigido del nodo que representa a o_i al nodo que representa su valor básico. Si el valor del objeto es construido, dibujamos arcos dirigidos del nodo objeto a un nodo que represente el valor construido. En general, no debe tener ciclos la gráfica de un *objeto individual*, ya que indicaría que un objeto se contiene a sí mismo como componente. Sin embargo, la gráfica que representa a un *tipo de objetos* puede tener ciclos que representen relaciones recursivas. Por ejemplo, un objeto EMPLEADO puede referirse a su SUPERVISOR, el cual también es de tipo EMPLEADO. Sin embargo, en la gráfica de objetos individuales EMPLEADO, el supervisor debe ser un objeto empleado distinto, y por lo tanto no existir ciclos. La Figura 16.1 muestra la gráfica del objeto ejemplo DEPTO dado anteriormente.

El modelo precedente permite dos tipos de definiciones en una comparasión de igualdad de los valores de dos objetos. Se dice que dos objetos tienen **valores idénticos** si las gráficas que representan sus valores son idénticas en cada aspecto, incluyendo los OIDs en cada nivel. Otra definición débil de igualdad es cuando dos objetos tienen **valores iguales**. En este caso, las estructuras gráficas deben mostrar ser iguales, y todos los valores atómicos correspondientes en las gráficas también deben ser iguales. Sin embargo, algunos nodos internos correspondientes a las dos gráficas pueden tener objetos con diferentes OIDs.

Para comparar la igualdad de objetos un ejemplo simple puede ilustar la diferencia entre las dos definiciones. Consideremos los siguientes objetos o_1, o_2, o_3, o_4, o_5 y o_6:

o_1 = (i_1, tupla, <a_1:i_4, a_2:i_6>)
o_2 = (i_2, tupla, <a_1:i_5, a_2:i_6>)
o_3 = (i_3, tupla, <a_1:i_4, a_2:i_6>)
o_4 = (i_4, atom, 10)
o_5 = (i_5, atom, 10)
o_6 = (i_6, atom, 20)

Los objetos o1 y o2 tienen valores *iguales*, debido a que sus valores a nivel atómico son los mismos pero los valores son alcanzados a través de distintos objetos o4 y o5. Sin embargo, los valores de los objetos o1 y o2 son *idénticos*. Similarmente, o4 y o5 son iguales pero no idénticos, debido a que tienen diferentes OIDs.

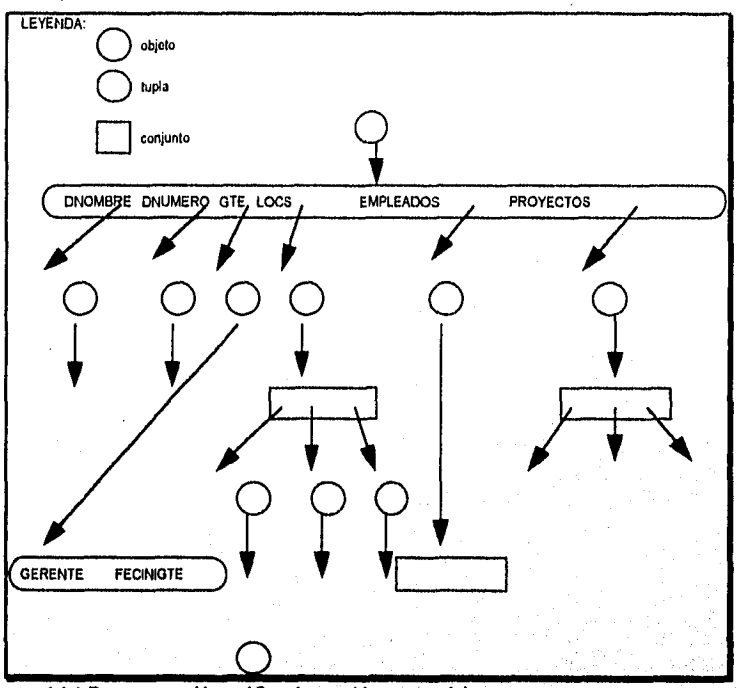


Figura 16.1 Representación gráfica de un objeto complejo.

16.2.3 Constructores de Tipo

Un **lenguaje de definición de datos** OO (OODDL) que incorpora los constructores de tipo precedentes puede usarse para definir objetos tipo de una aplicación de base de datos en particular. Por lo tanto, estos constructores de tipo pueden usarse para definir las estructuras de datos de un *esquema de base de datos* OO. La Figura 16.2 muestra cómo se pueden declarar tipos Depto y Empleado basado en los objetos de la Figura 16.1. También definimos a un tipo Fecha como una tupla (en vez de un valor atómico). Usaremos las instrucciones tupla, conjunto, y lista para los constructores de tipo, así como los tipos de datos estandar (entero, cadena, real, etc) de tipos atómicos.

Los atributos que refieren a otros objetos -tales como el departamento de Empleado o proyectos de Departamento- son básicamente **referencias**, y por lo tanto sirven para representar *relaciones* entre los tipos de objetos. Una relación binaria puede representarse en una dirección, o puede tener una *referencia inversa*. La última representación facilita transversar las relaciones en ambas direcciones. Por ejemplo, el atributo empleados de Depto tiene como su valor a un *conjunto de referencias* a objetos de tipo Empleado; estos son los empleados que trabajan para el departamento. El atributo de relación inversa dept de empleado refiere al departamento en particular para el que trabaja un empleado.

Posteriormente veremos que algunos OODBMS permiten que las inversas sean explícitamente declaradas, para asegurar que las referencias inversas son consistentes.

```
define type     Empleado:
        tuple(  nombre:     string,
                imss:       string,
                fechanac:   date,
                sexo:       char,
                dept:       Depto);

define type     Date:
tuple(          año:        integer,
                mes:        integer,
                día:        integer);

define type     Depto:
tuple(          dnombre:    string,
                dnumero:    integer,
                gte:        tuple (     gte:        Empleado,
                                        fecini:     Date),
                locs:       set(string),
                empleados:  set(Empleado),
                proyectos:  set(proyecto) );
```

Figura 16.2 Usando OODDL para definir tipos Empleado, Fecha, y Departamento.

16.3 Encapsulación de Operaciones, Métodos, y Persistencia

El concepto de encapsulación es una de las características principales de los lenguajes y sistemas OO. También están relacionados los conceptos de tipos de datos abstractos y ocultamiento de información en los lenguajes de programación. En las bases de datos tradicionales, no se aplicó este concepto, debido a que es aconsejable el hacer la estructura de los objetos de la base de datos visibles a los usuarios y a programas externos que accesan las tuplas de relación y atributos por el uso de estas operaciones.

Los conceptos de ocultamiento y encapsulación de información pueden aplicarse a objetos de la base de datos. La idea principal es definir el comportamiento de un tipo de objeto basado en las operaciones que pueden aplicarse externamente a objetos de ese tipo. La estructura interna del objeto es oculta, y el objeto es sólo accesible a través de un número de operaciones predefinidas. Algunas operaciones pueden usarse para crear o destruir objetos; otras pueden actualizar el valor del objeto (o estado del objeto); y otras se usan para recuperar partes del valor del objeto o aplicar algunos cálculos al valor del objeto. Otras operaciones pueden realizar una combinación de recuperación, cálculo, y actualización. En general, puede especificarse la *implementación* de una operación en un *lenguaje de programación de propósito general* que proporciona flexibilidad y poder en la definición de operaciones.

Los usuarios externos del objeto están sólo informados de la **interface** del objeto, la cual define los nombres y argumentos (parámetros) de cada operación. La **implementación** del objeto está oculta de los usuarios externos; incluye la definición de las estructuras de datos internos del objeto y la implementación de las operaciones que accesan a estas estructuras. En terminología OO, a la parte interface de cada operación se le llama **signatura**, y a la

implementación de la operación se le llama **método**. Típicamente, se invoca a un método enviando un **mensaje** al objeto para ejecutar el método correspondiente. Notemos que, como parte del método de ejecución, puede invocarse un mensaje subsecuente a otro objeto, y este mecanismo puede usarse para retornar valores de los objetos al medio ambiente externo o a otros objetos.

Para aplicaciones de bases de datos, el requerimiento de que todos los objetos estén completamente encapsulados es demasiado rígido. Una forma de aligerar este requerimiento es dividir la estructura de un objeto en atributos **visibles** y **ocultos** (variables). Los atributos visibles pueden accesarse directamente leyendo los operadores externos, o mediante un lenguaje query de alto nivel. Los atributos ocultos de un objeto son completamente encapsulados, y sólo pueden accesarse a través de operaciones predefinidas. La mayoría de los OODBMS emplean lenguajes query de alto nivel para accesar atributos visibles, y varias propuestas intentan extender el lenguaje SQL para usarse con bases de datos OO.

En la mayoría de los casos, las operaciones que *actualizan* el estado de un objeto están encapsuladas. Esta es una forma de definir la actualización semántica de los objetos, dado que en muchos modelos de datos OO, son pocas las contensiones de integridad *programadas en métodos* que crean, borran y actualizan a los objetos. En tales casos, todas las operaciones de actualización se implementan mediante operaciones encapsuladas. El término **clase** se usa para referir a una definición de tipo de objeto, junto con las definiciones de operaciones para ese tipo. La Figura 16.3 muestra cómo puede extenderse el OODDL de la Figura 16.2 para definir clases. Se declara un número de operaciones para cada objeto de la clase, y se incluye la signatura (interface) de cada operación en la definición de la clase. Para cada operación en cualquier otra parte debe definirse un método (implementación), usando un lenguaje de programación. Las operaciones típicas incluyen la operación *constructor de objeto* -la cual se usa para crear un nuevo objeto- y la operación *destructor* -la cual se usa para destruirlo. También puede declararse un número de operaciones *modificadoras de objetos* para modificar varios atributos (variables) de un objeto.

Es aconsejable para un OODBMS estar estrechamente acoplado con un lenguaje de programación OO. El lenguaje de programación OO se usa para especificar la implementación de métodos. Un objeto es típicamente creado por algunos programas ejecutables, invocando a la operación constructora del objeto. No todos los objetos se almacenan de manera permanenete en la base de datos. Existen **objetos transitorios** en el programa ejecutable y desaparecen una vez que el programa finaliza. Los **objetos persistentes** se almacenan en la base de datos y persisten después de la terminación del programa. El mecanismo típico de persistencia involucra dar a un objeto un **nombre** único persistente a través del cual puede recuperarse por este y otros programas o haciendo al objeto *alcanzable* desde algun objeto persistente. Un objeto B se dice ser **alcanzable** desde un objeto A si una secuencia de referencias en la gráfica objeto conduce de un objeto A a un objeto B. Por ejemplo, todos los objetos de la Figura 16.1 son alcanzables

desde el objeto o8; por lo tanto, si o8 se hace persistente, todos los demás objetos de la Figura 16.1 también se vuelven persistentes.

```
define class      Empleado:
type
        tupla(    nombre:    string,
                  imss:      string,
                  fechanac:  Date,
                  sexo:      char,
                  dept:      Depto);

operations
edad(e: Empleado):              integer,
crea_nuevo_emp:                 Empleado,
destuye_emp(e:Empleado):        boolean;

define class      Depto
type
        tuple(    dnombre:   string,
                  dnumero:   integer,
                  gte:       tuple(   gerente:   Empleado,
                                      fecini:    Date),
                  locs:      set (string),
                  empleados: set (Empleado),
                  proyectos: set (Proyecto) )

operations
numero_de_emps(d:Depto):                integer,
crea_nuevo_depto:                       Depto,
destruye_depto (d:Depto):               boolean,
agrega_emp(d:Depto, e:Empleado):        boolean,
elimina_emp(d:Depto, e:Empleado):       boolean;
```

Figura 16.3 Usando OODDL para definir las clases Empleado y Depto.

Si primero creamos un objeto persistente llamado N, cuyo valor es un *conjunto* o *lista* de objetos de alguna clase C, podemos hacer objetos peristentes de C *agregándolos* al conjunto o lista, y así hacerlos alcanzables de N. Por lo tanto, N define una **colección persistente** de objetos de clase C. Por ejemplo, podemos definir una clase DeptoSet (véase Figura 16.4) cuyos objetos son del tipo **set**(Depto). Supongamos que se crea un objeto del tipo DeptoSet, y que se nombra TodosDep y así hecho persistente, como se ilustró en la Figura 16.4. Cualquier objeto Depto que se agregue al conjunto de TodosDep usando la operación agrega_dep se vuelve persistente en virtud de ser alcanzable de TodosDep.

Notemos la diferencia entre los modelos de base de datos estandar y OO en este respecto. En un modelo típico de bases de datos, tales como EER o el modelo relacional, se asume que *todos los objetos* son persistentes.

En el acercamiento típico OO, una declaración de clase EMPLEADO sólo especifica el tipo y operaciones de una clase de objetos. El usuario debe definir separadamente un objeto persistente de tipo set (EMPLEADO) o lista (EMPLEADO) cuyo valor es la *colección de referencias* a todos los objetos persistentes EMPLEADO, si así se desea, como se ilustró en la Figura 16.4. En efecto, si así se desea, es posible definir varias colecciones persistentes para la misma definición de clase. La principal razón de esta diferencia es que permite que los objetos transitorios y persistentes sigan el mismo tipo y

declaración de clase del OODDL y del lenguaje de programación OO. Debido a que nuestro interés principal es de aplicaciones de bases de datos, asumiremos que, para cada declaración de clase, el nombre de la clase refiere a las definiciones de *tipo* y *operaciones*, así como al *conjunto de todos los objetos persistentes* de esa clase.

```
define class DeptoSet
type
        set (Depto)

operations
        crea_depto_set:                         DeptoSet,
        destruye_depto_set (ds:DeptoSet):       boolean,
        agrega_dep(ds: DeptoSet, d:Depto):      boolean,
        elimina_dep(ds: DeptoSet, d:Depto):     boolean;

persistent name TodosDep: DeptoSet;

d:= crea_nuevo_depto;

b:= agrega_dep(TodosDep, d);
```

Figura 16.4 Creación de objetos persistentes por nombramiento y alcanzabilidad.

16.4 Tipo y Jerarquías de Clase y Herencia

Otra característica principal de los sistemas OO es que deben permitir tipos o jerarquías de clase y herencia. Las jerarquías y jerarquías de clase son conceptualmente diferentes; pero al final resultan conducir a estructuras que se comportan de manera similar, se toman con frecuencia para describir lo mismo. En nuestra presentación, primero discutiremos las jerarquías de tipo, y luego las jerarquías de clase. En esta sección utlizamos un *modelo OO diferente* -un modelo en el cual los atributos y operaciones se tratan de manera uniforme- debido a que pueden heredarse aributos y operaciones. Además, el término **clase** en esta sección se refiere al *conjunto de objetos* de un tipo en particular.

16.4.1 Jerarquías de Tipo y Herencia

En la mayoría de las aplicaciones de bases de datos, existen numerosos objetos del mismo tipo. Por lo tanto, las OODB deben proporcionar la capacidad de clasificar objetos basados en su tipo, como lo hacen los demás sistemas de bases de datos. Pero en las OODB, un requerimiento mayor es que el sistema permite la definición de nuevos tipos basados en otros tipos predefinidos, llevando a una **jerarquía de tipo**.

Típicamente, un tipo se define asignándole un nombre y luego un número de atributos (variables instancia) y operaciones (métodos). En algunos casos, a los atributos y operaciones en conjunto se les llaman *funciones*. Subsecuentemente, puede usarse un nombre de función para referirir al valor de un atributo o para referir a la implementación de una operación (método). En esta sección, usaremos el término **función** para referir a atributos *y* operaciones de un tipo de objeto.

Puede definirse un tipo asignándole un **nombre** y luego listando los nombres de sus **funciones**. En esta sección, cuando se especifica un tipo, usaremos el siguiente formato simplificado:

NOMBRE_TIPO: función, función, ..., función

Por ejemplo, un tipo que describe las características de una PERSONA puede definirse como sigue:

PERSONA: Nombre, Dirección, Fechanac, Edad, IMSS

En el tipo PERSONA, las funciones Nombre, Dirección, IMSS y Fechanac se implementan como atributos almacenados, mientras que la función Edad se implementa como un método que calcula la Edad a partir del valor del atributo Fechanac y la fecha actual.

El concepto de **subtipo** es útil cuando el diseñador o usuario debe crear un nuevo tipo que es similar, pero no idéntico a uno ya definido. El usbtipo entonces **hereda** todas las funciones del tipo predefinido, al cual llamaremos **supertipo**. Por ejemplo, supongamos que deseamos definir dos tipos nuevos EMPLEADO y ESTUDIANTE como sigue:

EMPLEADO: Nombre, Dirección, Fechanac, Edad, IMSS, Salario, Fechaini, Antiguedad
ESTUDIANTE: Nombre, Dirección, Fechanac, Edad, IMSS, Carrera, Prom

Debido a que ESTUDIANTE y EMPLEADO incluyen todas las funciones definidas para PERSONA *más* algunas funciones adicionales propias, podemos declararlas como **subtipos** de PERSONA. Cada uno heredará las funciones de PERSONA previamente definidas -llamadas Nombre, Dirección, Fechanac, Edad e IMSS. Para ESTUDIANTE, sólo es necesario definir las funciones Carrera y Prom, las cuales no se heredaron. Presumiblemente, Carrera puede definirse como un atributo almacenado, mientras que Prom puede implementarse como un método que accesa los valores de Cal que se almacenan internamente dentro de cada objeto ESTUDIANTE. Para EMPLEADO, las funciones Salario y Fechaini pueden almacenarse como atributos, mientras que Antiguedad puede ser un método calculado a partir de Fechaini.

La idea de definir un tipo involucra definir todas sus funciones e implementarlas ya sea como atributos o métodos. Cuando se define un subtipo, puede heredar a todas estas funciones y su implementación. Sólo necesitan definirse e implementarse las funciones que son específicas del subtipo, y por lo tanto no implementadas en el supertipo. Por lo tanto, podemos declarar EMPLEADO y ESTUDIANTE como sigue:

EMPLEADO **subtype-of** PERSONA: Salario, Fechaini, Antiguedad
ESTUDIANTE **subtype-of** PERSONA: Carrera, Prom

Declarando cada EMPLEADO y ESTUDIANTE como *subtipo de persona*, estamos especificando que cada uno hereda las funciones de PERSONA. En general, un subtipo incluye *todas* las funciones declaradas para su supertipo, *más* algunas funciones adicionales que son sólo específicas del tipo. Por lo tanto, es posible generar una **jerarquía de tipo** para mostrar las relaciones supertipo/subtipo entre los tipos declarados en el sistema.

Como otro ejemplo, consideremos un tipo que describe objetos en geometría plana, el cual puede definirse como sigue:

OBJETO_GEOMETRICO: Forma, Area, PuntoReferencia

Para el tipo OBJETO_GEOMETRICO, Forma se implementa como un atributo (sus valores pueden ser triángulo, rectángulo, círculo, etc), y Area es un método que se aplica para calcular el área. Ahora supongamos que definimos un número de subtipos para el tipo OBJETO_GEOMETRICO como sigue:

RECTANGULO **subtype-of** OBJETO_GEOMETRICO: Ancho, Altura
TRIANGULO **subtype-of** OBJETO_GEOMETRICO: Lado1, Lado2, Angulo
CIRCULO **subtype-of** OBJETO_GEOMETRICO: Radio

Notemos que la operación área puede implementarse por un método diferente para cada subtipo, debido a que el procedimiento es diferente para rectángulos, triángulos y círculos. De manera similar, el atributo PuntoReferencia puede tener un significado diferente para cada subtipo; podría ser el punto central de los objetos RECTÁNGULO y CIRCULO, y el vértice entre los dos lados del objeto TRIANGULO. Algunos sistemas OO permiten el **renombramiento** de funciones heredadas en diferentes subtipos para reflejar más estrechamente el significado.

Una forma alterna de declarar estos tres subtipos es especificar el valor del atributo Forma como una condición que debe satisfacerse para objetos de cada subtipo:

RECTANGULO **subtype-of** OBJETO_GEOMETRICO: (Forma = 'rectángulo'): Ancho, Altura

TRIANGULO **subtype-of** OBJETO_GEOMETRICO (Forma = 'triángulo'): Lado1, Lado2, Angulo

CIRCULO **subtype-of** OBJETO_GEOMETRICO (Forma = 'círculo'): Radio
Aquí, solo los objetos OBJETO_GEOMETRICO cuya Forma = 'rectángulo' son del subtipo RECTANGULO, y de manera similar para los otros dos subtipos. En este caso, todas las funciones del supertipo OBJETO_GEOMETRICO se heredan por cada uno de los tres subtipos, pero el valor del atributo forma se restringe a un valor específico de cada uno.

Notemos que las definiciones de tipos *no generan* objetos por sí mismas. Son sólo declaraciones de ciertos tipos; y como parte de esa declaración, se especifica la implementación de las funciones de cada tipo. En una aplicación de bases de datos, existen muchos objetos de cada tipo. Cuando se crea un objeto, pertenece típicamente a uno o más de estos tipos que han sido declarados. Por ejemplo, un objeto círculo es del tipo CIRCULO *y* OBJETO_GEOMETRICO (por herencia). Cada objeto también se vuelve miembro de una o más clases de objeto, las cuales se usan para agrupar en conjunto colecciones de objetos que son significativos para la aplicación de base de datos.

16.4.2 Jerarquías de Clase

Una **clase** es una colección de objetos que son significativos para alguna aplicación. En la mayoría de los OODB, la colección de objetos en una clase tiene el mismo tipo. Sin embargo, esta no es una condición necesaria. Por ejemplo, SMALLTALK, un lenguaje OO, así llamado *sin tipos*, permite que una clase contenga una colección de objetos sin importar su tipo. Este también puede ser el caso cuando otros lenguajes no orientados a objetos sin tipos, tales como LISP, se extienden con conceptos OO.

Una clase se define típicamente por su nombre y por una colección de objetos que se incluyen en la clase. Con frecuencia es útil tener la posibilidad de definir una **subclase** a partir de otra clase de objetos, donde a la última se le llama **superclase**. En este caso, la contensión es que cada objeto en la subclase debe ser miembro de la superclase. Algunos sistemas OO tienen una clase predefinida por el sistema (llamada la clase **RAIZ** o la clase **OBJETO**) que contiene a todos los objetos en el sistema. Entonces procede la clasificación especializando objetos en clases adicionales que son siginificativas para la aplicación, creando una **jerarquía de clases** para el sistema. Todas las clases del sistema y las definidas por el usuario son subclases de la clase OBJETO, directa o indirectamente.

Notemos que los constructores de tipo permiten que el valor de un objeto sea una colección de objetos, lo cual es esencialmente una clase. Por lo tanto, una clase de objetos cuyos valores se basan en el *constructor de conjuntos* define un número de colecciones, una correpondiente para cada objeto. Los objetos valuados por conjunto son miembros de otra clase. Esto permite esquemas de clasificación multinivel, donde un objeto en una clase define como su valor a una clase de objetos.

16.5 Objetos Complejos

Una de las motivaciones principales que llevó al desarrollo de sistemas OO fue el deseo de representar objetos complejos. Existen dos tipos principales de objetos complejos: estructurados y no estructurados. Un objeto complejo estructurado está hecho de objetos componentes ensamblados aplicando los constructores de tipo disponibles recursivamente a varios niveles. Un objeto complejo no estructurado típicamente es un tipo de dato que requiere una gran cantidad de almacenamiento, tales como un tipo de dato que representa una imágen o un gran objeto textual.

16.5.1 Objetos Complejos No Estructurados y Extensibilidad de Tipo

Una facilidad de **objeto complejo no estructurado** proporcionada por un DBMS permite el almacenamiento y recuperación de grandes objetos que son necesarios para alguna aplicación. Ejemplos típicos de tales objetos son las *imágenes bitmap* y *grandes cadenas de texto*; también conocidas como **grandes objetos binarios** o **BLOBs**, abreviado. Estos objetos son no estructurados en el sentido de que el DBMS no conoce cuál es su estructura -sólo la aplicación que las usa puede interpretar el significado de los objetos. Por ejemplo, la aplicación puede tener funciones para desplegar una imágen o investigar ciertas palabras en una gran cadena de texto. Los objetos se consideran complejos porque requieren de una gran área de almacenamiento y no son parte de los tipos de datos estándar proporcionados por DBMSs típicos. Debido a que el tamaño de los objetos es algo grande, un DBMS puede recuperar una porción del objeto y proporcionarlo al programa de aplicación antes de recuperar al objeto completo. El DBMS también puede usar buffereo y técnicas de cache para precargar porciones del objeto antes de que el programa de aplicación necesite accesarlos.

El software del DBMS no tiene la capacidad de procesar directamente condiciones de selección y otras operaciones basadas en valores de estos objetos, a menos que la aplicación proporcione el código para hacer las operaciones de comparasión necesarias para la selección. En un OODBMS, puede lograrse definiendo un nuevo tipo de dato abstracto para los objetos no interpretados y proporcionando los métodos de selección, comparasión y desplegado de tales objetos. Por ejemplo, consideremos objetos que son imágenes bit map de dos dimensiones. Supongamos que la aplicación necesita seleccionar de una colección de tales objetos sólo a aquéllos que incluyen un determinado patrón. En este caso, el usuario debe proporcionar el programa de reconocimiento de patrones como un objeto sobre el objeto de tipo bit map. El OODBMS entonces recupera un objeto de la base de datos y corre el método para reconocimiento de patrones sobre él para determinar si el objeto incluye al patrón requerido.

Debido a que un OODBMS permite la creación de nuevos tipos, y a que el tipo incluye estructura y operaciones, podemos visualizar un OODBMS como un **sistema de tipo extensible**. Podemos crear librerías de nuevos tipos definiendo su estructura y operaciones, incluyendo tipos complejos. Las aplicaciones pueden entonces usar o modificar estos tipos, en el último caso creando subtipos a partir de los tipos ya existentes en las librerías. Sin embargo, los internos del DBMS deben proporcionar el almacenamiento básico y capacidades de recuperación de objetos que requieren grandes cantidades de almacenamiento de tal forma que las operaciones puedan aplicarse de manera eficiente. Muchos OODBMSs proporcionan el almacenamiento y recuperación de grandes objetos no estructurados como cadenas de caracteres o cadenas de bits, los cuales pueden pasarse "como tal" al programa para su interpretación.

16.5.2 Objetos Estructurados Complejos

Un **objeto complejo estructurado** difiere de uno no estructurado en que la estructura del objeto se define por la aplicación repetida de los constructores de tipo proporcionados por el OODBMS. Por lo tanto la estructura del objeto se define y se conoce por el OODBMS. Como ejemplo, consideremos el objeto DEPTO mostrado en la Figura 16.1. En el primer nivel, el objeto tiene una estructura tupla con seis atributos: DNOMBRE, DNUMERO, GTE, LOC, EMPLEADOS, y PROYECTOS. Sin embargo, sólo dos de estos atributos - llamados, DNOMBRE y DNUMERO- tienen valores básicos; los otros cuatro tienen valores complejos y por lo tanto construyen el segundo nivel de la estructura del objeto complejo. Uno de estos cuatro (GTE) tiene una estructura tupla, y los otros tres (LOC, EMPLEADOS, PROYECTOS) tienen una estructura conjunto. En el tercer nivel, para un valor de tupla GTE, tenemos un atributo básico (FECINIGTE) y un atributo (GTE) que refiere a un objeto empleado, el cual tiene una estructura tupla. Para un conjunto LOC, tenemos un conjunto de valores básicos, pero para los conjuntos EMPLEADOS y PROYECTOS, tenemos conjuntos de objetos estructurados en tuplas.

Existen dos tipos de referencia semántica entre un objeto complejo y sus componentes en cada nivel. El primer tipo, al cual llamamos **propiedad semántica**, se aplica cuando los subobjetos de un objeto complejo están encapsulados dentro de él y por lo tanto se consideran parte del objeto complejo. El segundo tipo, al cual llamaremos **referencia semántica**, se aplica cuando los componentes del objeto complejo son por sí mismos objetos independientes pero que a veces pueden considerarse parte del objeto complejo. Por ejemplo, podemos considerar los atributos DNOMBRE, DNUMERO y LOC propiedad de un DEPTO, mientras que GTE, EMPLEADOS y PROYECTOS debe referenciarse debido a que representan subobjetos independientes. El primer tipo también es referido como *parte de* o *componente de* la relación; y al segundo tipo se le llama *asociado con* la relación, debido a que describe una asociación igual entre dos objetos independientes. La relación es parte de la construcción de objetos complejos y tiene la propiedad de que los objetos componentes están encapsulados dentro del objeto complejo y se consideran parte de la estructura interna del objeto. Sólo pueden accesarse por métodos de ese objeto, y se eliminan cuando el objeto mismo es eliminado. Por otro lado, un objeto complejo cuyos componentes se referencian se considera consistir de objetos independientes que pueden tener su propia identidad y métodos. Cuando un objeto complejo necesita accesar a sus componentes referenciados, debe hacerlo invocando los métodos apropiados de los componentes, debido a que no están encapsulados dentro del objeto. Además, un componente de un objeto referenciado puede referenciarse por más de un objeto complejo y por lo tanto no se borra automáticamente cuando se borra al objeto complejo.

Un OODBMS debe proporcionar opciones de almacenamiento para *agrupar* los componentes de objetos de un objeto complejo juntos en almacenamiento secundario para incrementar la eficiencia de operaciones que accesan al objeto complejo. En muchos casos, la estructura del objeto se almacena en páginas disco, las cuales pueden referir a

páginas adicionales que deban recuperarse. Esto se conoce como **ensamblaje de objetos complejos**.

16.6 Otros Conceptos OO

En esta sección daremos un panorama de algunos conceptos adicionales OO, incluyendo el polimorfismo (sobrecarga de operadores), herencia múltiple, versionamiento, y configuraciones.

16.6.1 Polimorfismo (Sobrecarga de Operadores)

Otra característica de los sistemas OO es que proporcionan el **polimorfismo** de operaciones, el cual es algunas veces referido como **sobrecarga de operadores**. Este concepto permite que el *mismo nombre del operador* o *símbolo* sean límite de dos o más *implementaciones* diferentes del operador, dependiendo del tipo de objetos a los cuales se aplica el operador. Un ejemplo sencillo de lenguajes de programación puede ilustrar este concepto. En algunos lenguajes, el operador símbolo "+" puede significar diferentes cosas cuando se aplica a operandos (objetos) de diferentes tipos. Si los operandos de "+" son de tipo *integer*, la operación invocada es una suma. Si los operandos de "+" son de tipo *conjunto*, la operación invocada es la unión. El compilador puede determinar qué operación ejecutar basado en los tipos de operandos proporcionados en el programa.

En OODB, puede ocurrir una situación similar. Usaremos el ejemplo OBJETO_GEOMETRICO para ilustrar el polimorfismo en BDOO. Supongamos que declaramos OBJETO_GEOMETRICO y sus subtipos como sigue:

OBJETO_GEOMETRICO: forma, Area, PuntoCentral

RECTANGULO **subtype-of** OBJETO_GEOMETRICO(Forma = 'rectángulo'): Ancho, Altura

TRIANGULO **subtype-of** OBJETO_GEOMETRICO(Forma = 'TRIÁNGULO'): Lado1, Lado2, Angulo)

CIRCULO **subtype-of** OBJETO_GEOMETRICO(Forma = 'círculo'): Radio

Aquí, se declara la función Area para todos los objetos del tipo OBJETO_GEOMETRICO. Sin embargo, la implementación del método de Area puede diferir para cada subtipo de OBJETO_GEOMETRICO. Una posibilidad es tener una implementación general para calcular el área de un OBJETO_GEOMETRICO generalizado (por ejemplo, escribiendo un algoritmo general para calcular el área de un polígono) y luego reescribir algoritmos más eficientes para calcular las áreas de objetos geométricos específicos, tales como el círculo, rectángulo, triángulo, etc. En este caso, la función Area se *sobrecarga* por diferentes implementaciones.

El OODBMS debe seleccionar ahora el método apropiado para la función Area basado en el tipo de objeto geométrico al cual se aplicó. En sistemas fuertemente tipeados, esto puede lograrse al momento de la compilación, debido a que los tipos de objeto deben ser conocidos. A esto se le llama **colindancia temprana**, debido a que la elección del método a usar puede hacerse al momento de la compilación. Sin embargo, en sistemas con tipeo débil o sin tipo (tales como SMALLTALK y LISP), el tipo del objeto al cual se aplica la función debe checar el tipo de objeto al momento de la corrida y luego invocar al método apropiado. Esto es con frecuencia referido como **colindancia tardía** de la función al método de implementación, debido a que la colindancia se hace al runtime.

16.6.2 Herencia Multiple y Selectiva

La **herencia múltiple** en un tipo de jerarquía ocurre cuando un cierto subtipo T es un subtipo de dos (o más) tipos diferentes y por lo tanto hereda las funciones (atributos y métodos) de ambos supertipos. Por ejemplo, podemos crear un subtipo GTE_INGENIERIA que es subtipo de GERENTE e INGENIERO. Esto lleva a la creación de una **red de tipo** en lugar de una jerarquía de tipo. Un problema que puede ocurrir con la herencia múltiple es que los dos supertipos de los cuales el subtipo hereda pueden tener funciones distintas del mismo nombre, creando una ambiguedad. Por ejemplo, GERENTE e INGENIERO pueden tener una función Salario. Si la función Salario se implementa mediante diferentes métodos en los supertipos GERENTE e INGENIERO, existe una ambiguedad sobre cuál de los dos es heredado por el subtipo GTE_INGENIERO. Es posible, sin embargo, que INGENIERO y GERENTE hereden Salario del mismo supertipo (tal como EMPLEADO) más alto en la red. En tal caso, no existe ambiguedad; el problema sólo llega si las funciones son distintas en los dos supertipos.

Existen varias técnicas para tratar con la ambiguedad en la herencia múltiple. Una solución es tener el chequeo de ambiguedad en el sistema cuando se crea el subtipo, y permitir que el usuario elija explícitamente cuál función va a heredarse en este momento. Otra solución es usar algún default del sistema. Una tercera solución es desaprobar la herencia múltiple si ocurre ambiguedad, en vez de forzar al usuario a cambiar el nombre de una de las funciones en uno de los supertipos. Algunos sistemas OO no permiten la herencia múltiple de ninguna forma.

La **herencia selectiva** ocurre cuando un subtipo hereda sólo algunas de las funciones de un supertipo. Otras funciones no se heredan. En este caso, puede usarse una cláusula EXCEPT para listar las funciones en un supertipo que *no son* heredadas por el subtipo. El mecanismo de herencia selectivo no se proporciona en OODB, pero se usa más frecuentemente en aplicaciones de inteligencia artificial.

16.6.3 Versiones y Configuraciones

Muchas aplicaciones que usan sistemas OO requieren de la existencia de varias **versiones** del mismo objeto. Por ejemplo, consideremos una aplicación para un medio ambiente de

ingeniería de software que almacena varios artefactos de software, tales como módulos de diseño, módulos de código fuente, información de configuración para describir qué módulos deben enlazarse para formar un programa complejo, y probar todos los casos posibles en el sistema. Comúnmente, conforme los requerimientos evolucionan se aplican *actividades de mantenimiento* a un sistema de software. El mantenimiento usualmente involucra cambiar algunos de los módulos de diseño e implementación. Si un sistema ya es operacional, y uno o más de los módulos deben cambiarse, el diseñador debe crear una **nueva versión** de cada uno de estos módulos para implementar los cambios. De manera similar, deben haberse generado nuevas versiones de los casos de prueba. Sin embargo, las versiones existentes no deben descartarse hasta que las nuevas versiones hayan sido completamente probadas y aprobadas; sólo entonces las nuevas versiones deben reemplazar a las anteriores.

Notemos que pueden existir más de dos versiones de un objeto. Por ejemplo, consideremos a dos programadores trabajando para actualizar el mismo módulo de software concurrentemente. En este caso, además del módulo original, son necesarias dos versiones. Los programadores pueden actualizar sus propias versiones del *mismo módulo* concurrentemente. Esto es con frecuencia referido como **ingeniería concurrente**. Sin embargo, se vuelve eventualmente necesario cargar juntas estas dos versiones de tal forma que la versión híbrida pueda incluir los cambios hechos por ambos programadores. Durante la unión, también es necesario asegurarse que sus cambios son compatibles. Esto necesita la creación de otra versión del objeto: una que sea resultado de la unión de las dos versiones actualizadas independientemente.

Como pudo verse en la discusión precedente, un OODBMS debe ser capáz de almacenar y manejar distintas versiones del mismo objeto. Varios sistemas proporcionan esta capacidad, permitiendo que la aplicación mantenga múltiples versiones de un objeto y se refiera explícitamente a versiones particulares cuando es necesario. Sin embargo, el problema de unir y reconciliar los cambios hechos a dos versiones diferentes se deja a los desarrolladores de la aplicación, quienes conocen la semántica. Algunos DBMSs tienen ciertas facilidades que pueden comparar las dos versiones con el objeto original y determinar si alguno de los cambios es incompatible, para asistir con el proceso de unión. Otros sistemas mantienen una **versión gráfica** que muestra las relaciones entre versiones. Cada que una versión v1 origina una copia de otra versión v, puede dibujarse un arco directo de v a v1. De manera similar, si se cargan dos versiones v2 y v3 para crear una nueva versión v4, se dibjuan arcos dirigidos de v2 y de v3 a v4. La gráfica de versión puede ayudar a los usuarios a entender las relaciones entre las distintas versiones, y puede usarse internamente por el sistema para manejar la creación y borrado de versiones.

Cuando se aplica el versionamiento a objetos complejos, surgen otros problemas que resolver. Un objeto complejo, tal como el sistema de software, puede constar de muchos módulos. Cuando se permite el versionamiento, cada uno de estos módulos puede tener un número diferente de versiones y una gráfica de versión. Una **configuración** del objeto complejo es una colección consistente de una versión de cada módulo arreglado de tal manera que las versiones de módulo en la configuración sean compatibles y juntos formen

una versión válida del objeto complejo. Una nueva versión o configuración del objeto complejo no tiene que incluir las nuevas versiones de cada módulo. Por lo tanto, ciertas versiones de módulo que no han sido cambiadas pueden pertenecer a más de una configuración del objeto complejo. Notemos que una configuración es una colección de versiones de *diferentes objetos* que en conjunto forman un objeto complejo, mientras que la gráfica de versión describe la colección de versiones del *mismo objeto*. Una configuración debe seguir el tipo de estructura de un objeto complejo; configuraciones múltiples del mismo objeto complejo son análogas a versiones múltiples de un componente de objeto.

16.7 Ejemplos de OODBMSs

Ahora ilustraremos los conceptos ya discutidos examinando OODBMSs. Se presentará un panorama del Sistema O2 de O2 Technology, así como de ObjectStore de Object Design Inc.

16.7.1 Panorama del Sistema O2

En nuestro panorama del Sistema O2, primero ilustraremos la definición de datos y luego consideraremos ejemplos de manipulación de datos en O2. Después de esto, haremos una breve discusión de la arquitectura del Sistema O2.

Definición de Datos en O2. En O2, el esquema define los tipos de objetos y clases del sistema. Se define un tipo de objeto usando los tipos atómicos proporcionados por O2 y aplicando los constructores de tipo. Los *tipos atómicos* incluyen Boolean, character, integer, real, string y bits. Los *constructores de tipo* incluyen tupla, lista, conjunto y conjunto único. El constructor de conjunto, cuando se especifica sólo, permite elementos duplicados (similar a lo que llamamos *bolsa*), mientras que el constructor *conjunto único* no (similar lo que llamamos *conjunto*). En O2, los métodos no se incluyen como parte de la definición del tipo. En cambio, una *definición de clase* consta de dos partes: el *tipo* de objetos que pueden ser miembros de la clase, y los *métodos* que pueden aplicarse a estos objetos.

En O2 existe una distinción entre valores y objetos. Un *valor* tiene un solo tipo y se representa a sí mismo. Un *objeto* pertenece a una clase y por lo tanto tiene un tipo y un comportamiento especificado por el método de la clase. Además, un objeto tiene una *identidad única* (OID) y un valor actual (o estado), mientras que un valor no tiene OID. Valores y objetos pueden tener tipos complejos, y los diferentes niveles del tipo complejo pueden ser sus valores o referir a otros objetos usando sus OIDs. El sistema O2 tiene un lenguaje, O2C, que puede usarse para definir clases, métodos y tipos, y para crear objetos y valores. Los objetos son *persistentes* o *transitorios*. Los valores son transitorios a menos que se vuelvan parte de un objeto persistente.

La Figura 16.5 muestra declaraciones posibles de clase y declaraciones en O2C para una porción de la base de datos UNIVERSIDAD. Para ilustrar cómo pueden declararse los

métodos hemos incluido a algunos. En O2 es posible declarar una nueva clase E' que hereda los tipos y métodos de otra clase E. En nuestra terminología, E' es una subclase de E, *y* el tipo de E' debe ser un subtipo de E. Por ejemplo, en la Figura 16.5, Estudiante y Facultad se declaran como subclases de Persona usando la instrucción **inherit** Persona en las definiciones de Estudiante y Facultad. Es posible **renombrar** los atributos o métodos heredados y **redefinir** la estructura de atributos heredados o la implementación de métodos heredados de una subclase. Por ejemplo, si deseamos renombrar al atributo imss de Persona como id_estudiante en la Subclase Estudiante de Persona, podemos incluir la siguiente instrucción en la definición de clase de Estudiante:

rename imss **as** id_estudiante

Para redefinir un método, primero lo renombramos, y luego incluimos una definición del método renombrado.

O2 también permite la **herencia múltiple**, donde una clase hereda el tipo y los métodos de dos o más clases. Si las dos clases tienen un atributo o un método con el mismo nombre, es mejor renombrar a estos dos de tal forma que sus nombres puedan distinguirse en la subclase. En la instrucción rename, podemos calificar al nombre del atributo o método con el nombre de la clase colocando _nombreclase al atributo o nombre del método.

En O2 las relaciones clase/subclase y herencia se especifican usando la instrucción **inherit** en la declaración de subclase. Por ejemplo, en la Figura 16.5, las clases Facultad y Estudiante heredan el tipo y métodos de la clase Persona, y cada una incluye sus atributos y métodos adicionales. La clase Facultad tiene los siguientes atributos adicionales: rango, salario, fofic, y ftel. Además, incluimos en la clase Facultad atributos valuados por conjunto -pertenece_a, cuota, y avisos- para representar a las relaciones PERTENECE, PI y CONSEJERO, respectivamente.

Manipulación de Datos en O2. Las aplicaciones de O2 pueden desarrollarse de dos formas usando lenguajes de programación. La primera es usar el lengueje query propio de O2, O2SQL, y el lenguaje de programación, O2C, para escribir programas de aplicación. La segunda es usar O2 como un sistema de almacenamiento de objetos persistentes para otro lenguaje independiente, tal como C++, y desarrollar las aplicaciones en ese lenguaje. O2 también cuenta con una interface al usuario llamado O2Look, la cual puede usarse para acelerar el desarrollo de la aplicación facilitando la interacción con O2C a través de una interface amigable con el usuario. Otra herramienta, llamada O2Tools, es un medio ambiente gráfico para el desarrollo de programas que incluye herramientas tales como un browser, un depurador, un shell que permite la edición y ejecución de comandos O2 e interacción con archivos Unix, y un manejador de espacio de trabajo que puede almacenar temporalmente objetos de bases de datos.

```
type Teléfono: tuple (          cod_area: integer,
                                número: integer);

type Fecha tuple      (         año: integer,
                                mes: integer,
                                día: integer);

class Persona
        type tuple (            imss: string,
                                nombre: tuple(      nombre: string,
                                                    inicial: string,
                                                    apellido: string),
                                dirección tuple(    número: integer,
                                                    calle: string,
                                                    no_dep: string,
                                                    ciudad: string,
                                                    estado: string,
                                                    cp: string),
                                fechanac: Fecha,
                                sexo: character)
        method
                                edad: integer
end

class Studen inherit Persona
        type tuple (            clase: string,
                                mayores_en: Depto,
                                menores_en: Depto,
                                regsitrado_en: set(Sección),
                                transcripto: set (tuple (       calif: character,
                                                                nclaif: real,
                                                                sección: Sección) ) )
        method
                promedio_calif: real,
                cambia_clase: boolean,
                cambia_carrera: (nueva_carrera: Depto); boolean
end
```

Figura 16.5 Declaraciones de clase en O2 para parte de la base de datos UNIVERSIDAD.

Primero ilustraremos el lenguaje O2C y su uso en la escritura de métodos para clases. Este lenguaje es un subconjunto del lenguaje C que ha sido extendido para manejar tipos O2, incluyendo operaciones para conjuntos y listas. La Figura 16.6(a) muestra la definición de algunos de los métodos que se declararon en las clases O2 de la Figura 16.5. El primer método, edad, calcula la edad de una persona a partir de la fecha de nacimiento y la fecha de hoy, usando sólo constructores de lenguaje C: Notemos que el uso de la instrucción **self** dentro de un método refiere al objeto para el cual se invocó el método. Así, la palabra *self* en el método edad refiere a un objeto de tipo Persona y tiene el tipo y métodos de tal objeto. El segundo método, promedio_cal, usa un ciclo for que itera sobre el conjunto de valores en el atributo transcrito de Estudiante para obtener la suma de las calificaciones numéricas y el número de registros transcritos; luego calcula y retorna el promedio. El tercer método es un ejemplo de actualización, y cambia la carrera de un estudiante. Este método también invoca a otros dos de la clase Depto -elimina_carrera y agrega_carrera- los cuales también se muestran en la Figura 16.6(a). Así se hace para asegurar que los valores del atributo inverso -carreras de la clase Depto- permanezcan consistentes con el valor del atributo en_carreras de la clase Estudiante. Esto proporciona un ejemplo de cómo pueden implementarse relaciones bidireccionales definiendo los métodos apropiados O2.

```
class Calif_Estudiante inherit Estudiante
        type tuple (        calificaciones: set (tuple (    colegio: string,
                                                            grado: string,
                                                            año: integer) );
end

class Facultad inherit Persona
        type tuple (        salario: real,
                            rango: string,
                            fofic: string,
                            ftel: string,
                            pertenece_a: set(Depto),
                            cuota: set(Cuotas),
                            consejos: set(Estudiante) )
        method
                            promueve_facultad,
                            da_aumento (porciento: real)
end

class Depto
        type tuple (        dnombre: string,
                            oficina: string,
                            dtel: string,
                            miembros: set (Facultad),
                            carreras: set (Estudiante),
                            presidente: Facultad,
                            cursos: set (Curso) )

        method
                            agrega_carrera(s: Estudiante),
                            elimina_carrera(s: Estudiante): boolean
end

class Sección
        type tuple (        num_sec: integer,
                            cto: cuarto,
                            año: Año,
                            estudiantes: set ( tuple (      stud:Estudiante,
                                                            calif: character) ),
                            curso: Curso,
                            maestro: Instructor)
        method
                    cambia_cal(s:Estudiante, g:character)
end

class Curso
        type tuple (        cnombre: string,
                            cnumero: string,
                            cdescripción: string,
                            secciones: set(Sección),
                            dept_ofrece:Depto)
        method
                    actualiza_descripción (nuevo_d: string)
end
```

Figura 16.5 (continuación)

En O2 existen dos formas de hacer a un objeto persistente. Una es hacer del objeto mismo una **raíz**, dándole un nombre a través de la instrucción **name**. En este caso, al objeto se le llama *raiz persistente.* La otra forma es hacer al objeto *alcanzable* desde una raíz persistente -por ejemplo, haciéndolo miembro de un objeto persistente valuado por conjunto o haciéndolo componente de un objeto complejo. En general, O2 no crea un objeto persistente que corresponda a cada declaración de clase; en lugar, deja que la tarea la haga el usuario de manera explícita. Por lo tanto, si es necesario un conjunto de todos

los objetos persistentes de tipo Persona, el programador debe crear una raiz persistente de tipo raiz (Persona) y darle un nombre, como se muestra en la Figura 16.6(b), donde al conjunto se le llama Todas_Per. Cualquier objeto que se agregue a ese conjunto, se vuelve automáticamente persistente, debido a que se vuelve *unido a una raiz persistente*. El objeto Persona "Franklin Wong" mostrado en la Figura 16.6(b) no era persistente cuando se creó, pero lo hace cuando se agrega al objeto Todas_Per. También se puede crear un objeto individual persistente sólo asignándole un nombre -sin que tenga que pertenecer a ningún conjunto persistente- como se ilustró con el objeto Persona llamado "John Smith" en la Figura 16.6(b).

O2 también tiene un lenguaje query, O2SQL, el cual puede usarse para recuperar un conjunto de valores a una clase transitoria en un programa O2C. O2SQL permite la creación de un nuevo tipo *en el vuelo* para un resultado de query. El lenguaje query tiene una estructura **SELECT...FROM...WHERE,** similar a la de SQL, pero es posible en O2SQL usar referencia funcional dentro del query para referir a los componentes y valores de objetos complejos. La referencia funcional usa la *notación punto* para referir a los componentes de objetos complejos; por ejemplo, escribiendo

s.en_carreras.dnombre

en Q1 de la Figura 16.7 primero localizamos al objeto DEPARTAMENTO relacionado a ESTUDIANTE s a través de la función en_carreras, y luego localizar el atributo dnombre de ese departamento.

En un query O2SQL, la cláusula SELECT define los datos a recuperarse y la estructura (tipo) de los valores en el reultado del query. La cláusula FROM especifica las colecciones de objetos referenciadas por el query y da nombres de variables que abarcan sobre objetos en esas colecciones. Una **colección** es tipicamente un conjunto de objetos. En la Figura 16.7 asumismos que Estudiante es el nombre de una colección de todos los objetos Estudiante persistentes. Finalmente, la cláusula WHERE especifica las condiciones para seleccionar los objetos individuales en el resultado del query. La Figura 16.7 muestra ejemplos de dos queries. El query Q1 recupera los nombres de todos los estudiantes de ciencias de la computación, y el resultado del query es un conjunto de tuplas con dos atributos: nombre y apellido. El query Q2 recupera las transcripciones de estudiantes en ciencias de la computación, y el resultado tiene un tipo complejo cuya estructura se especifica en la cláusula SELECT del query. La facilidad para definir un nuevo tipo de estructura para los valores en un resultado query no está disponible en todos los OODBMSs. Algunos sistemas sólo permiten la recuperación de objetos completos de una colección como tal, sin reestructurarlos para el resultado del query. El programador puede entonces reestructurar explícitamente estos objetos como objetos transitorios de nuevos tipos que se declaran en el programa del usuario.

```
(a)
method body edad: integer in class Persona
{       int a;
        Date d;
        d := today();
        a := d -> year - self -> fechanac ->año;
        If (d -> mes < self -> fechanac -> mes) ||
          ((d -> mes == self -> fechanac -> mes) && (d -> dia < self -> fechanac -> dia))
          --a;
        return a;
}

method body promedio_cal: real in class Estudiante
{       float suma := 0.0;
        int contador := 0;
        struct{
                char gr;
                float ncalif;
                o2_Seccion sec;
        } t;
for(t in self -> transcript) {
        suma += t -> ncalif; ++contador;
        }
return suma/contador  ;
}

method body elimina_carrera (s:Estudiante): boolean in class Depto
{       if (s in self -> carrera){
                self -> carrera -= set(s);
                return 1;
                }
        }
        else return 0;
}

method body agrega_carrera (s:Estudiante) in class Depto
{       self ->carrera += set(s);
}
```

Figura 16.6 Programación O2. (a) Declaraciones de métodos en O2 para algunos de los métodos incluidos en la Figura 16.5 (b) Creación de objetos persistentes en O2.

Una alternativa para usar el lenguaje O2C es usar un lenguaje independiente, tal como C++, en conjunto con el sistema O2. Para facilitar esto, O2 proporciona una **facilidad de exportación** que crea clases de C++ correpondientes a las declaraciones de clase de O2. La Figura 16.8 muestra una clase C++ que corresponde a la clase Persona declarada en la Figura 16.5. Los objetos O2 de la clase persistente Persona pueden recuperarse directamente en variables de programa C++ que se definen ser de la clase correspondiente C++, y estos pueden manipularse por código del programa C++.

Panorama de la Arquitectura del Sistema O2. En esta sección, daremos un breve panorama de la arquietectura del sistema O2. Una parte del sistema O2, llamada O2Engine, es responsable de mucha de la funcionalidad del DBMS. Esto incluye proporcionar soporte para almacenar, recuperar, y actualizar persistentemente objetos almacenados que pueden compartirse por múltiples programas. O2Engine implementa el control de concurrencia, recuperación, y mecanismos de seguridad que son típicos en sistemas de bases de datos. Además, O2Engine implementa un modelo de administración de transacción y mecanismos de evolución del esquema.

```
(b)
name Todas_Per set (Persona)
name John Smith: Persona;
run body {
        o2 Persona p := new Persona;

        *p = tuple (imss: "333445555",
                nombre: tuple(nombre:"Franklin", inicial: "T", apellido: "Wong"),
                dirección: tuple(número: 638, calle: "Voss Road", ciudad: "Houston",
                estado: "Texas", cp: "77079"),
                fechanac: tuple (año: 1945, mes: 12, día: 8),
                sexo: M);
        Todas_Per+=set(p);
John Smith -> imss = "123456789",
John Smith -> nombre: tuple (nombre:"John", inic: "B", apellido: "Smith"),
John Smith -> dirección: tuple(numero: 731, calle: "Fondren Road", ciudad: "Houston",
                        estado: "Texas", cp: "77036"),
John_Smith->fechanac: tuple(año: 1955, mes: 1, día: 9),
John_Smith->sexo: M;
}
```

Figura 16.6 Continuación

```
Q1:select tuple (fnombre:s.nombre.nombre,
                apellido:s.nombre.apellido)
        from s in Estudiante
        where s.mayores_in.dnombre = "Ciencias de la Computación"

Q2:select tuple (fnombre: s.nombre.nombre,
                apellido.s.nombre.apellido)
                transcript: select tuple (
                                cnombre: sc.sección.curso.cnombre,
                                no_sec:sc.sección.num_sec,
                                cuarto:sc.sección.cto,
                                año:sc.sección.año,
                                grado:sc.sección.grado)
                        from sc in sec)
from s in Estudiante, sec in s.transcribe
where s.mayores_in.dnombre = "Ciencias de la computación"
```

Figura 16.7 Dos queries en O2SQL

La implementación de O2Engine a nivel de sistema estaba basado en una *arquitectura cliente/servidor*, para acomodar la tendencia actual hacia sistemas de cómputo en red y distribuidos. El *componente servidor*, el cual puede ser un servidor de archivos, es un *servidor de página*; sólo trabaja con almacenamiento a nivel página (bloque de disco) y no conoce la estructura de los objetos. Su responsabilidad es recuperar páginas eficientemente cuando es instruido por un cliente, y mantener el control de concurrencia apropiado y recuperar información a nivel de página. En O2, el control de concurrencia utiliza bloqueos, y la recuperación se basa en una técnica de logging adelantado. El servidor también efectúa cierta cantidad de cacheo de paginación para reducir I/O del disco, y se accesa a través de una interface de llamada a procedimiento remoto (RPC) desde los clientes. Un *cliente* es típicamente una estación de trabajo; y la mayor parte de la funcionalidad de O2 se proporciona a nivel cliente.

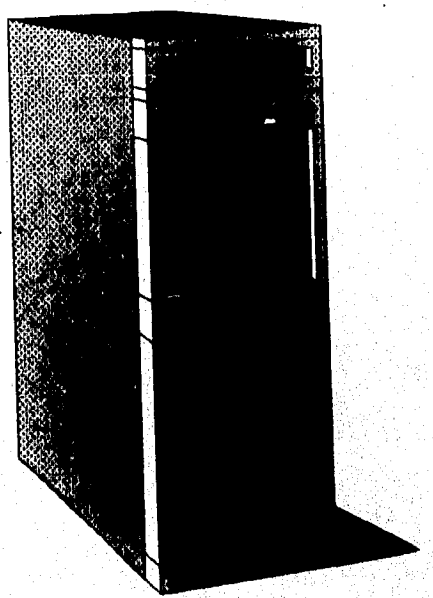
```
class Persona: o2_root {
public:
        char* imss;
        struct{
                char* nombre;
                char* inicial;
                char* apellido} nombre;
        struct{
                int numero;
                char* calle;
                char* no_dep;
                char* ciudad;
                char* estado;
                char* cp} dirección;
        struct{
                int año;
                int mes;
                int dia} fechanac;

        int edad();
}
```

Figura 16.8 Declaración de clase C++ correspondiente a la clase Persona de O2.

A nivel funcional, O2Engine tiene tres componentes principales. El *componente de almacenamiento*, al nivel más bajo, es una extensión de un sistema de almacenamiento llamado WiSS (Sistema de Almacenamiento Wisconsin). La implementación de esta capa se encuentra entre el cliente y el servidor. El servidor proporciona la administración del disco, almacenamiento y recuperación de páginas, control de concurrencia, y recuperación. El cliente procesa el cache de páginas y los bloqueos que hayan sido proporcionados por el servidor y los pone al servicio de los módulos funcionales de más alto nivel del cliente O2.

El siguiente componente funcional, llamado el *administrador de objetos*, tiene que ver con la estructura de los objetos y los valores, agrupando a los objetos relacionados en paginas de disco, indexando objetos, manteniendo la identidad del objeto, realizando operaciones sobre objetos etc.

CAPITULO 17



Bases de Datos Distribuidas y Arquitectura Cliente-Servidor

17.1 Introducción a los Conceptos de DBMS Distribuidos

Una **base de datos distribuida** es una colección de datos que pertenece lógicamente al mismo sistema pero físicamente se distribuye sobre los sitios de una red de cómputo. Las ventajas potenciales de los DDBSs incluyen las siguientes:

- *Naturaleza distribuida de algunas aplicaciones de bases de datos*: Muchas aplicaciones están *naturalmente distribuidas* sobre diferentes localidades. Por ejemplo, una compañía puede tener locales en diferentes ciudades, o un banco puede tener varias sucursales. Es natural que las bases de datos usadas en tales aplicaciones estén distribuidas sobre esas localidades. Muchos **usuarios locales** accesan sólo los datos en esta localidad, pero otros **usuarios globales** -tales como oficinas centrales de la compañía- pueden requerir acceso ocasional a datos almacenados en varias de estas localidades. Notemos que los datos de cada localidad típicamente describen un "minimundo" de ese sitio. La fuente de los datos y la mayoría de los usuarios y aplicaciones de la base de datos local residen físicamente en ese sitio.

- *Incremento de la rentabilidad y disponibilidad*: Estas son dos de las ventajas más comunes citadas para las bases de datos distribuidas. La **rentabilidad** está ampliamente definida como la probabilidad de que un sistema esté disponible en un momento en particular, mientras que la **disponibilidad** es la probabilidad de que un sistema esté continuamente disponible durante un intervalo de tiempo. Cuando los datos y el DBMS se distribuyen sobre varios sitios, un sitio puede fallar mientras que los demás continúan operando. Sólo no podrán accesarse los datos y el software existente en el lugar de la falla. Esto mejora la rentabilidad y la disponibilidad. Se logra mayor mejora *replicando* datos y software en más de un sitio. En un sistema centralizado, la falla en cualquier sitio hace que el sistema sea completamente inoperable para todos los usuarios.

- *Permitir el compartimiento de datos mientras se mantiene alguna medida de control local*: En algunos tipos DDBSs, es posible controlar los datos y el software en cada lugar. Sin embargo, ciertos datos pueden accesarse por usuarios en otros lugares remotos a través del DDBMS. Esto permite el compartimiento *controlado* de datos a través del sistema distribuido.

- *Mejora en el desempeño*: Cuando se distribuye una gran base de datos sobre distintos lugares, existen bases más pequeñas en cada sitio. Como resultado, los queries locales y las transacciones que accesan a los datos en un solo sitio tienen mejor desempeño debido al tamaño de las bases de datos locales. Además, cada sitio tiene un número de transacciones más pequeño en ejecución que si todas las transacciones se enviaran a una sola base de datos centralizada. Para transacciones que involucran acceso a más de un sitio, puede proceder el procesamiento en paralelo en los diferentes sitios, reduciendo el tiempo de respuesta.

La distribución lleva al incremento de complejidad en el diseño e implementación del sistema. Para lograr las ventajas potenciales ya mencionadas, el DDBMS debe ser capáz de proporcionar las siguientes *funciones adicionales* a las del DBMS centralizado:

- Capacidad de accesar sitios remotos y transmitir queries y datos entre los distintos sitios a través de una red de comunicaciones.
- Capacidad de registrar la distribución y replicación de datos en el catálogo del DDBMS.
- Capacidad de planear estrategias de ejecución de queries y transacciones que accesen datos de más de un sitio.
- Capacidad de decidir cuál copia de un elemento de dato replicado se debe accesar.
- Capacidad de mantener la consistencia de copias de un elemento de dato replicado.
- Capacidad de recuperar de caídas de sitios individuales y de nuevos tipos de fallas tales como la falla en el enlace de comunicaciones.

Estas funciones por sí mismas incrementan la complejidad de un DDBMS sobre un DBMS centralizado. Antes de darnos cuenta de la ventaja potencial de la distribución, debemos encontrar soluciones satisfactorias a estos problemas. Incluir toda esta funcionalidad adicional es difícil de completar, y encontrar soluciones óptimas es ir un paso más allá. Cuando consideramos el diseño de una base de datos distribuida aparecen complejidades adicionales. Específicamente, debemos decidir cómo distribuir los datos sobre los sitios y qué datos replicar, si es que existen.

A nivel físico de **hardware**, los siguientes factores distinguen a un DDBS de un sistema centralizado:

- Existen varias computadoras, llamadas **sitios** o **nodos**.
- Como se muestra en la Figura 17.1, estos sitios deben estar conectados por algún tipo de **red de comunicaciones** para transmitir datos y comandos entre sitios.

Los sitios pueden localizarse en la proximidad -digamos, dentro del mismo edificio o grupo de edificios adyacentes- y conectarse a través de una **red de área local**, o pueden estar geográficamente distribuidos sobre grandes distancias y conectados a través de una **gran red**. Las redes de área local típicamente usan cables, mientras que las grandes redes usan líneas telefónicas o satélites. También es posible usar una combinación de los dos tipos de redes.

Las redes pueden tener diferentes **topologías** que definen las trayectorias de comunicación entre los sitios. Por ejemplo, pueden existir enlaces directos entre los sitios 1 y 2 y entre

los sitios 2 y 3 pero no entre 1 y 3; en tal caso, la comunicación entre los sitios 1 y 3 debe pasar por el sitio 2. El tipo y topología de la red utilizada pueden tener un efecto significativo en el desempeño y por lo tanto en las estrategias de procesamiento de queries distribuidos y diseño de la base de datos distribuida. En cuestiones de arquitectura de alto nivel, no importa qué tipo de red se use; sólo importa que cada sitio sea capáz de comunicarse, directa o indirectamente, con cada sitio. En lo que resta de este capítulo, asumiremos que existe algún tipo de red de comunicaciones entre los sitios, sin importar la topología en particular.

17.2 Panorama de la Arquitectura Cliente-Servidor

La **arquitectura cliente-servidor** ha sido desarrollada para trabajar con los nuevos ambientes de cómputo en los cuales un gran número de computadoras personales, estaciones de trabajo, servidores de archivos, impresoras, y demás equipo se conecta junto a través de una red. La idea es definir **servidores especializados** con funcionalidades específicas. Por ejemplo, es posible conectar un grupo de estaciones de trabajo sin disco o computadoras personales como clientes a un *servidor de archivos* que mantiene los archivos de los usuarios del cliente. Otra máquina podría asignarse como un *print server* conectándose a varias impresoras; y en adelante, todas las requisiciones de impresión serían canalizadas a esta máquina. De esta forma, pueden accesarse los recursos proporcionados por servidores especializados por muchos clientes. La idea puede llevarse al software, con software especializado -tales como un DBMS o CAD- almacenados en servidores específicos, y este software hacerlo accesible a varios clientes.

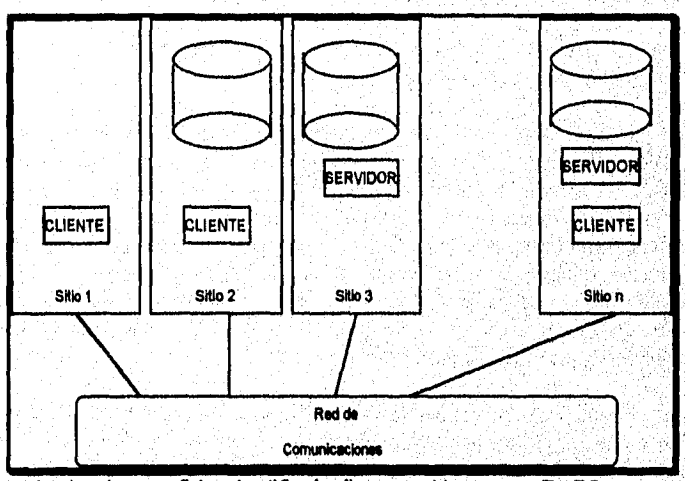


Figura 17.1 Arquitectura física simplificada cliente-servidor para un DDBS.

La arquitectura cliente-servidor se está incorporando en paquetes DBMS comerciales conforme se mueven al soporte de distribución. La idea es dividir el DBMS en dos niveles

-cliente y servidor- para reducir su complejidad. Esto se ilustra en la Figura 17.1; algunos sitios pueden correr únicamente el software del cliente (estos podrían ser las máquinas sin disco, tales como en el sitio 1, o máquinas con disco, tales como en el sitio 2), mientras que otros sitios pueden dedicarse como servidores que corran únicamente el software del servidor, tales como el sitio 3. Existen otros sitios que pueden soportar ambos módulos cliente y servidor, tales como el sitio n en la Figura 17.1.

Aún no ha sido establecido cómo dividir la funcionalidad del DBMS entre cliente y servidor. Se han propuesto diferentes puntos de vista. Una posibilidad es incluir la funcionalidad de un DBMS centralizado a nivel servidor. Un número de productos DBMS relacionales han tomado esta idea, donde un **servidor SQL** se proporciona a los clientes. Cada cliente debe entonces formular los queries SQL apropiados y proporcionar la interface del usuario y funciones del lenguaje de programación. Debido a que SQL es un estándar relacional, varios servidores SQL, posiblemente de varios vendedores, pueden aceptar comandos SQL. El cliente también puede ofrecer un diccionario de datos que incluya información sobre la distribución de datos entre los distintos servidores SQL, así como módulos para descomponer un query global en un número de queries locales que pueden ejecutarse en distintos sitios. La interacción entre cliente y servidor podría proceder como sigue durante el procesamiento de un query SQL:

1. El cliente parsea un query y lo descompone en un número de queries independientes de sitio. Cada query es enviado al servidor del sitio apropiado.

2. Cada servidor procesa el query local y envía la relación resultante al sitio del cliente.

3. El sitio del cliente combina los resultados de los subqueries para producir el resultado del query originalmente submitido.

En este punto de vista, el servidor SQL ha sido también llamado **procesador de base de datos** (DP) o **back-end**, mientras que al cliente se le ha llamado **procesador de aplicación** (AP) o **front-end**. La inetarcción entre cliente y servidor puede especificarse por el usuario a nivel de cliente o a través de un módulo cliente especializado. Por ejemplo, el usuario puede saber qué datos se almacenan en cada servidor, descomponer una requisición de query en subqueries de sitio manualmente, y submitir los subqueries individuales a los diferentes sitios. Las tablas resultantes pueden combinarse explícitamente por un usuario a nivel de cliente. La alternativa es tener el módulo cliente encargado de estas acciones automáticamente.

Otro acercamiento, tomado por algunos OODBMSs, divide al software en módulos entre el cliente y el servidor de manera más integrada. Por ejemplo, el *nivel servidor* puede incluir a la parte del DBMS responsable de manejar el almacenamiento de datos en páginas de disco, control de concurrencia local y recuperación, buffereo y cache de páginas de disco, y otras funciones similares. Mientras que, el *nivel cliente* puede manejar la interface con el usuario, funciones de diccionario de datos, interacción del DBMS con compiladores de lenguajes de programación, estructuramiento de objetos complejos de los datos en los

buffers, y otras funciones semejantes. En este acercamiento, la interacción cliente-servidor está más acoplada y se hace internamente por los módulos del DBMS, en vez de por los usuarios.

En un DDBMS típico, es personalizable la división de los módulos del software en tres niveles:

- El software **del servidor** es responsable del manejo local de datos en un sitio, casi como el DBMS centralizado.
- El software del **cliente** es responsable de la mayoría de las funciones de distribución; accesa la información de distribución del catálogo del DDBMS y procesa todas las requisiciones que requieren acceso a más de un lugar.
- El **software de comunicaciones** (algunas veces junto con un **sistema de operación distribuida)** proporciona la comunicación primitiva que se usa por el cliente para transmitir comandos y datos entre los distintos sitios cuando es necesario. Este no es estrictamente parte del DDBMS, pero proporciona lo esencial de comunicaciones y servicios.

El **cliente** es responsable de generar un plan de ejecución distribuida para un query o transacción multisitio y supervisar la ejecución distribuida enviando comandos a los servidores. Estos comandos incluyen queries locales y transacciones a ejecutarse, así como comandos para transmitir datos a otros clientes o servidores. Por lo tanto, el software del cliente debe incluirse en cualquier sitio donde se submiten queries. Otra función controlada por el cliente es la de asegurar la consistencia de copias de un elemento de datos empleando técnicas de control de concurrencia distribuida (o global). El cliente también debe asegurar la atomicidad de transacciones globales realizando recuperaciones globales cuando fallan ciertos sitios. La Figura 17.2 muestra una vista lógica de un DDBS; aquí, clientes y servidores se muestran sin especificar el sitio sobre el cual reside cada uno.

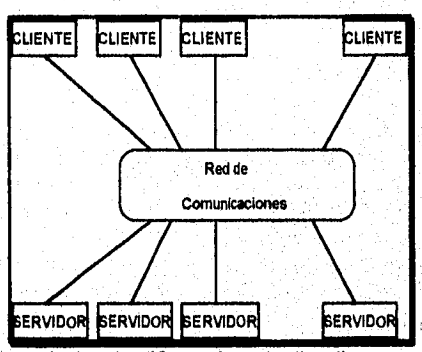


Figura 17.2 Arquitectura lógica simplificada de un DDBS cliente-servidor.

Una posible función del cliente es *ocultar* los detalles de distribución de datos del usuario; esto es, le permite al usuario escribir queries globales y transacciones como si la base estuviera centralizada, sin tener que especificar los sitios en los cuales residen los datos referenciados en el query o la transacción. A esta propiedad se le llama **transparencia de distribución**. Algunos DDBMSs no proporcionan la transparencia de distribución, requieren que los usuarios se preocupen de los detalles de la distribución de datos. En este caso, los usuarios deben especificar los sitios en los cuales residen los datos que se referencian en queries globales y transacciones. En la primer situación, un usuario típicamente *agrega el nombre del sitio* a cualquier referencia a una relación, archivo, o registro. En la última situación, se presenta al usuario un esquema que no incluye ninguna información de distribución, y el DDBMS registra los sitios en los cuales se localizan los datos en el **catálogo del DBMS**. Usando esta información, el software del cliente puede romper un query en un número de subqueries que pueden ejecutarse en distintos sitios, y planear cómo transmitir resultados de subqueries a otros sitios para mejor procesamiento y la producción del resultado. Los DDBMS que proporcionan transparencia de distribución facilitan al usuario la especificación de queries y transacciones, pero requieren de un software más complejo. Los DDBMSs que no proporcionan transparencia de distribución dejan al usuario la responsabilidad de especificar el sitio de cada relación o archivo, llevando a un software más sencillo.

17.3 Fragmentación de Datos, Replicación, y Técnicas de Localización para el Diseño de Bases de Datos Distribuidas

En esta sección discutiremos las técnicas empleadas para descomponer a la base de datos en unidades lógicas, llamadas **fragmentos**, que pueden asignarse para almacenamiento en varios lugares. También discutiremos el uso de **aplicación de datos**, lo cual permite que ciertos datos se almacenen en más de un sitio, y el proceso de **localización** de fragmentos -o réplicas de fragmentos- para almacenamiento en varios lugares. Estas técnicas se usan durante el proceso de **diseño de bases de datos distribuidas**. La información concerniente a la fragmentación de datos, localización, y replicación se almacena en un **catálogo global del sistema** que se accesa por el software del cliente cuando es necesario.

17.3.1 Fragmentación de Datos

En un DDBS, deben tomarse decisiones en relación a cuál sitio debe usarse para almacenar qué porciones de la base de datos. Por ahora, asumiremos que *no existe replicación*; esto es, cada archivo -o porción del archivo- se almacena en un solo lugar. También usaremos de la terminología de *bases de datos relacionales*; conceptos similares aplicados a otros modelos de datos. Asumiremos que estamos empezando con un esquema de base de datos relacional y debemos decidir cómo distribuir las relaciones sobre los distintos sitios. Para ilustrar nuestra discusión, usaremos el esquema de la base de datos relacional de la Figura 6.5.

Antes de decidir cómo distribuir los datos, debemos determinar las *unidades lógicas* de la base de datos que van a distribuirse. Las unidades lógicas más simples son las relaciones

mismas; esto es, cada relación *completa* va a almacenarse en un sitio en particular. En nuestro ejemplo, debemos decidir un sitio para almacenar cada una de las relaciones EMPLEADO, DEPTO, PROYECTO, TRABAJA_EN y DEPENDIENTE de la Figura 6.5. En muchos casos, sin embargo, una relación puede dividirse en unidades lógicas más pequeñas para su distribución. Por ejemplo, consideremos la base de datos COMPAÑIA de la Figura 6.6, y asumamos que existen tres sitios de cómputo -uno para cada departamento de la compañía. Por supuesto, en una situación actual, existen mucho más tuplas en las relaciones. Podemos desear almacenar la información de la base de datos relacionada con cada departamento en el sitio de ese departamento. Para hacerlo asi, necesitamos particionar cada relación usando una técnica llamada fragmentación horizontal.

Fragmentación horizontal. Un **fragmento horizontal** de una relación es un subconjunto de las tuplas en esa relación. Las tuplas que pertenecen al fragmento horizontal se especifican por una condición sobre uno o más atributos de la relación. Con frecuencia, sólo se involucra a un solo atributo. Por ejemplo, podemos definir tres fragmentos horizontales en la relación EMPLEADO de la Figura 6.6 con las siguientes condiciones: (DNO = 5), (DNO = 4) y (DNO = 1); cada fragmento contiene las tuplas de EMPLEADO que trabajan para un departamento en particular. De manera similar, podemos definir tres fragmentos horizontales para la relación PROYECTO de la Figura 6.6 con las condiciones (DNUM = 5), (DNUM = 4) y (DNUM = 1); cada fragmento contiene a las tuplas PROYECTO controladas por un departamento en particular. La fragmentación horizontal divide una relación "horizontalmente" agrupando renglones para crear subconjuntos de tuplas, donde cada subconjunto tiene un cierto significado lógico. Estos fragmentos pueden entonces asignarse a diferentes sitios en el sistema distribuido.

Fragmentación Vertical. Un **fragmento vertical** de una relación guarda solo ciertos atributos de la relación. Por ejemplo, podemos desear fragmentar la relación EMPLEADO en dos fragmentos verticales. El primer fragmento que incluya la información personal - NOMBRE, FECHANAC, DIRECCION y SEXO- y la segunda incluye información relacionada con el trabajo -IMSS, SALARIO, SUPERIMSS, DNO. Esta fragmentación vertical no es muy apropiada porque, si los dos fragmentos se almacenan de forma separada, no podremos poner a las tuplas del empleado original juntas de regreso, ya que no existe *ningún atributo en común* entre los dos fragmentos. Es necesario incluir al *atributo llave primaria* en *cada* fragmento vertical de tal forma que la relación completa pueda reconstruirse de los fragmentos. Por lo tanto, debemos de agregar el atributo IMSS al fragmento de información personal. La fragmentación vertical divide una relación "verticalmente" por columnas.

Notemos que cada fragmento horizontal en una relación R puede especificarse por la operación σCi(R) del álgebra relacional. Un conjunto de fragmentos horizontales cuyas condiciones C1, C2, ..., Cn incluyen a todas las tuplas en R -esto es, a cada tupla en R satisface (C1 O C2 O ...O Cn)- se le llama **fragmentación horizontal completa** de R. En muchos casos una fragmentación horizontal completa es también **disjunta**; esto es, ninguna tupla en R satisface (Cy y Cj) para cualquier i ≠ j. Nuestros dos ejemplos

anteriores de fragmentación horizontal para las relaciones EMPLEADO y PROYECTO fueron completas y disjuntas. Para reconstruir a la relación R a partir de una fragmentación horizontal *completa*, necesitamos aplicar la operación UNION a los fragmentos.

Un fragmento vertical en una relación R puede especificarse por una operación $\pi Li(R)$ del álgebra relacional. Un conjunto de fragmentos verticales cuya proyección lista L1, L2, ..., Ln incluye a todos los atributos en R pero que sólo comparte al atributo llave primaria de R se le llama **fragmentación vertical completa** de R. En este caso la lista de proyección satisface las siguientes dos condiciones:

- $L1 \cup L2 \cup ... \cup Ln = ATTRS(R)$.
- $Li \cap Lj = PK(R)$ para cualquier $n \neq j$, donde ATTRS(R) es el conjunto de atributos de R y PK(R) es la llave primaria de R.

Para reconstruir la relación R a partir de una fragmentación vertical *completa*, apliquemos a los fragmentos la operación OUTER UNION. Notemos que también podríamos aplicar la operación FULL OUTER JOIN y obtener el mismo resultado para una fragmentación vertical completa. Los dos fragmentos verticales de la relación EMPLEADO con listas de proyección L1 = {IMSS, NOMBRE, FECHANAC, DIRECCION, SEXO} y L2 = {IMSS, SALARIO, SUPERIMSS, DNO} constituyen una fragmentación vertical completa de EMPLEADO.

Dos fragmentos horizontales que no son completos ni disjuntos son aquéllos definidos en la relación EMPLEADO de la Figura 6.5 por las condiciones (SALARIO > 5000) y (DNO = 4); pueden no incluir todas las tuplas de empleados, y pueden incluir tuplas comúnes. Dos fragmentos verticales que no son completos son aquéllos definidos por la lista de atributos L1 = {NOMBRE, DIRECCION} y L2 = {IMSS, NOMBRE, SALARIO}; estas listas violan ambas condiciones de una fragmentación vertical completa.

Fragmentación Mixta. Podemos mezclar ambos tipos de fragmentación, produciendo una **fragmentación mixta**. Por ejemplo, podemos combinar las fragmentaciones horizontal y vertical de la relación EMPLEADO dadas anteriormente en una fragmentación mixta que incluya seis fragmentos. En este caso la relación original puede reconstruirse aplicando en el orden apropiado las operaciones UNION y OUTER UNION. En general, puede especificarse un **fragmento** de una relación R por una combinación de operaciones SELECT-PROJECT $\pi L(\sigma C(R))$. Si C = TRUE y $L \neq ATTRS(R)$, obtenemos un fragmento vertical, y si $C \neq TRUE$ y L = ATTRS(R), obtendremos un fragmento horizontal. Finalmente, si $C \neq TRUE$ y $L \neq ATTRS(R)$, obtendremos un fragmento mixto. Notemos que una relación puede considerarse a sí misma un fragmento con C = TRUE y L = ATTRS(R). En la siguiente discusión, el término *fragmento* se usa para referir a una relación o a cualquiera de los tipos de fragmentos precedentes.

Un **esquema de fragmentación** de una base de datos es una definición de un conjunto de fragmentos que incluyen a *todos* los atributos y tuplas en la base de datos y que satisfacen la condición de que la base de datos completa puede reconstruirse a partir de los fragmentos aplicando alguna secuencia de operaciones OUTER UNION y UNION. También algunas veces es útil -aunque no necesario- tener todos los fragmentos disjuntos excepto por la repetición de las llaves primarias entre fragmentos verticales (o mixtos). En el último caso, toda la replicación y distribución de fragmentos se especifica claramente en un estado subsecuente, separado de la fragmentación.

Un **esquema de localización** describe la localización de fragmentos a sitios del DDBS; por lo tanto es un mapeo que especifica cada fragmento del sitio en el cual se almacena. Si un fragmento se almacena en más de un sitio, se dice estar **replicado**.

17.3.2 Replicación y Localización de Datos

La replicación es útil para mejorar la disponibilidad de datos. El caso más extremo es la replicación de la *base de datos completa* en cada sitio del sistema distribuido, creando así una base de datos distribuida **completamente replicada**. Esto puede mejorar la disponibilidad remarcadamente debido a que el sistema puede continuar operando tanto como al menos un sitio lo haga. También mejora el desempeño de recuperación de queries globales, debido a que el resultado de tal query puede obtenerse localmente desde cualquier sitio; por lo tanto, un query de recuperación puede procesarse en el sitio donde es submitido, si es que ese sitio incluye un módulo servidor. La desventaja de la replicación completa es que alenta de manera drástica las operaciones de actualización, debido a que debe realizarse una sóla actualización lógica sobre cada copia de la base de datos para mantener las copias consistentes. La replicación completa hace del control de concurrencia y técnicas de recuperación más costosas de lo que serían si no hubiera replicación.

El otro extremo de la replicación completa involucra el **no tener replicación**; esto es, cada fragmento se almacena en un solo sitio. En este caso todos los fragmentos *deben* estar disjuntos, excepto por la repetición de llaves primarias entre fragmentos verticales (o mixtos). A esto también se le llama **localización no redundante**.

Entre estos dos extremos, tenemos un amplio espectro de **replicación parcial** de los datos; esto es, algunos fragmentos de la base de datos pueden estar replicados mientras que otros no. El número de copias de cada fragmento puede ir desde una hasta el total de los sitios en el sistema distribuido. A la descripción de los fragmentos de replicación se le llama algunas veces **esquema de replicación**.

Cada fragmento -o cada copia de un fragmento- debe de asignarse a un sitio en particular en el sistema distribuido. A este proceso se le llama **distribución de datos** (o **localización de datos**).

La elección de sitios y el grado de replicación depende de las metas de desempeño y disponibilidad del sistema y de los tipos y frecuencias de transacciones submitidas en cada sitio. Por ejemplo, si se requiere de alta disponibilidad y las transacciones pueden submitirse a cualquier sitio y la mayoría de las transacciones son sólo de recuperación, una base de datos completamente replicada es una buena elección. Sin embargo, si ciertas transacciones que accesan partes en particular de la base de datos son la mayor parte del tiempo submitidas a un sitio en particular, el conjunto correspondiente de fragmentos puede localizarse únicamente en ese sitio. Los datos que se accesan en múltiples sitios pueden replicarse en esos sitios. Si se efectúan muchas actualizaciones, puede ser útil el limitar la replicación. Encontrar una solución óptima o aún buena para la localización de datos distribuidos es un problema de optimización compeljo.

17.3.3 Ejemplo de Fragmentación, Localización y Replicación

Ahora consideremos un ejemplo de fragmentación y distribución de la base de datos Compañía de las Figuras 6.5 y 6.6. Supongamos que la compañía tiene tres sitios de cómputo -uno para cada departamento. Los sitios 2 y 3 son para los departamentos 5 y 4, respectivamente. En cada uno de estos sitios, esperamos acceso frecuente a la información de EMPLEADO y PROYECTO de los empleados *que trabajan en ese departamento* y de los proyectos *controlados por ese departamento*. Después, asumimos que estos sitios accesan principalmente a los atributos NOMBRE, IMSS, SALARIO y SUPERIMSS de EMPLEADO. El sitio 1 se usa por las oficinas centrales de la compañía y accesa toda la información de empleados y proyectos regularmente, además de registrar la información de DEPENDIENTE para propósitos de seguridad.

De acuerdo a estos requerimientos, la base de datos completa de la Figura 6.6 puede almacenarse en el sitio 1. Para determinar los fragmentos a replicar en los sitios 2 y 3, podemos primero fragmentar horizontalmente a EMPLEADO, PROYECTO, DEPTO y DEP_LOCS por número de departamento -llamado DNO, DNUM, y DNUMERO, respectivamente, en la Figra 6.5. Ahora podemos fragmentar verticalmente los fragmentos resultantes de EMPLEADO para incluir sólo los atributos [NOMBRE, IMSS, SALARIO, SUPERIMSS, DNO]. La Figura 17.3 muestra los fragmentos mixtos EMPD5 Y EMPD4, los cuales incluyen a las tuplas de EMPLEADO que satisfacen la condición DNO = 5 y DNO = 4, respectivamente. Los fragmentos horizontales de PROYECTO, DEPARTAMENTO y DEP_LOCS se fragmentan de manera similar por número de departamento. Todos estos fragmentos -almacenados en los sitios 2 y 3- se replican debido a que también están almacenados en la oficinas centrales del sitio 1.

Ahora debemos fragmentar la relación TRABAJA_EN y decidir qué fragmentos de TRABAJA_EN almacenar en los sitios 2 y 3. Estamos enfrentando el problema de que ningún atributo de TRABAJA_EN indica directamente el departamento al cual pertenece la tupla. En efecto, cada tupla en TRABAJA_EN relaciona un empleado e a un proyecto p. Podríamos fragmentar TRABAJA_EN basado en el departamento d en el cual e trabaja o basado en el departamento d' que controla a p. La fragmentación se vuelve fácil si tenemos un argumento estableciendo que d=d' para todas las tuplas TRABAJA_EN -esto

es, si los empledos sólo pueden trabajar en proyectos controlados por el departamento para el que trabajan. Sin embargo, en nuestra base de datos de la Figura 6.6 no existe tal argumento. Por ejemplo, la tupla TRABAJA_EN <333445555, 10, 10.0> relaciona a un empleado que trabaja para el deparatmento 5 con un proyecto controlado por el departamento 4. En este caso podríamos fragmentar TRABAJA_EN basado en el departamento para el cual trabaja el empleado y el departamento que controla el proyecto, como se muestra en la Figura 17.4.

En la Figura 17.4, la unión de fragmentos G1, G2 y G3 da todas las tuplas de TRABAJA_EN para empleados que trabajan *para el departamento 5*. De manera similar, la unión de los fragmentos G4, G5 y G6 da todas las tuplas TRABAJA_EN para empleados que *trabajan para el departamento 4*. Por otro lado, la unión de fragmentos G1, G4 y G7 da todas las tuplas de TRABAJA_EN de proyectos *controlados por el departamento 5*. La condición de cada uno de los fragmentos G1 a G9 se muestra en la Figura 17.4. Las relaciones que representan relaciones M:N, tales como TRABAJA_EN, con frecuencia tienen varias fragmentaciones lógicas posibles. En nuestra distribución de la Figura 17.3, elegimos incluir todos los fragmentos que pueden unirse a una tupla de EMPLEADO o PROYECTO en los sitios 2 y 3. Por lo tanto, colocamos los fragmentos de unión G1,G2, G3, G4 y G7 en el sitio 2 y a la unión de los fragmentos G4, G5, G6, G2 y G8 en el sitio 3. Notemos que los fragmentos G2 y G4 son replicados en ambos sitios. Esta estrategia de localización permite la unión entre los fragmentos locales de EMPLEADO o PROYECTO en el sitio 2 o 3 y el fragmento local de TRABAJA_EN para efectuarse por completo localmente. Esto demuestra claramente la complejidad del problema de la fragmentación de grandes bases de datos.

17.4 Tipos de Sistemas de Bases de Datos Distribuidas

El término *sistema administrador de bases de datos distribuido* puede describir varios sistemas que difieren uno del otro en muchos aspectos. El principal punto que tales sistemas tienen en común es el hecho de que los datos y el software se distribuyen sobre múltiples sitios conectados por alguna forma de red de comunicación. En esta sección discutiremos un número de DDBMSs y el criterio y factores que hacen a algunos de estos sistemas diferentes.

El primer factor a considerar es el **grado de homogeneidad** del DDBMS. Si todos los servidores (o DBMSs locales) usan software idéntico y todos los clientes usan software idéntico, al DDBMS se le llama **homogeneo**; de otro modo, se le llama **heterogéneo**. Otro factor relacionado con el grado de homogeneidad es el **grado de autonomía local**. Por otro lado, si se permite *acceso directo* a un servidor por transacciones locales, el sistema tiene algún grado de autonomía local.

(a)

EMPD5

NOMBRE	MINIC	APELLIDO	IMSS	SALARIO	SUPERIMSS	DNO
John	B	Smith	123456789	30000	333445555	5
Franklin	T	Wong	333445555	40000	888665555	5
Ramesh	K	Narayan	666884444	38000	333445555	5
Joyce	A	English	453453453	25000	333445555	5

DEP5

DNOMBRE	DNUMERO	GTEIMSS	FECINIGTE
Investigación	5	333445555	22-MAY-88

DEP5_LOCS

DNUMERO	LOCALIDAD
5	Bellaire
5	Sugarland
5	Houston

TRABAJA_EN5

EIMSS	PNO	HORAS
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0

PROYS5

PNOMBRE	PNUMERO	PLOC	DNUM
ProductoX	1	Bellaire	5
ProductoY	2	Sugarland	5
ProductoZ	3	Houston	5

(b)

EMPD4

NOMBRE	MINIC	APELLIDO	IMSS	SALARIO	SUPERIMSS	DNO
Alicia	J	Zelaya	999887777	25000	987654321	4
Jennifer	S	Wallace	987654321	43000	888665555	4
Ahmad	V	Jabbar	987987987	25000	987654321	4

DEP4

DNOMBRE	DNUMERO	GTEIMSS	FECINIGTE
Administración	4	987654321	01-JUN-90

DEP5_LOCS

DNUMERO	LOCALIDAD
4	Stafford

TRABAJA_EN4

EIMSS	PNO	HORAS
333445555	10	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	10	10.0
987654321	20	10.0

PROYS4

PNOMBRE	PNUMERO	PLOC	DNUM
Computación	10	Stafford	4
Nuevos beneficios	30	Stafford	4

Figura 17.3 Localización de fragmentos en sitios. (a) Fragmentos de relación en el sitio 2. (b) Fragmentos de Relación en el sitio 3.

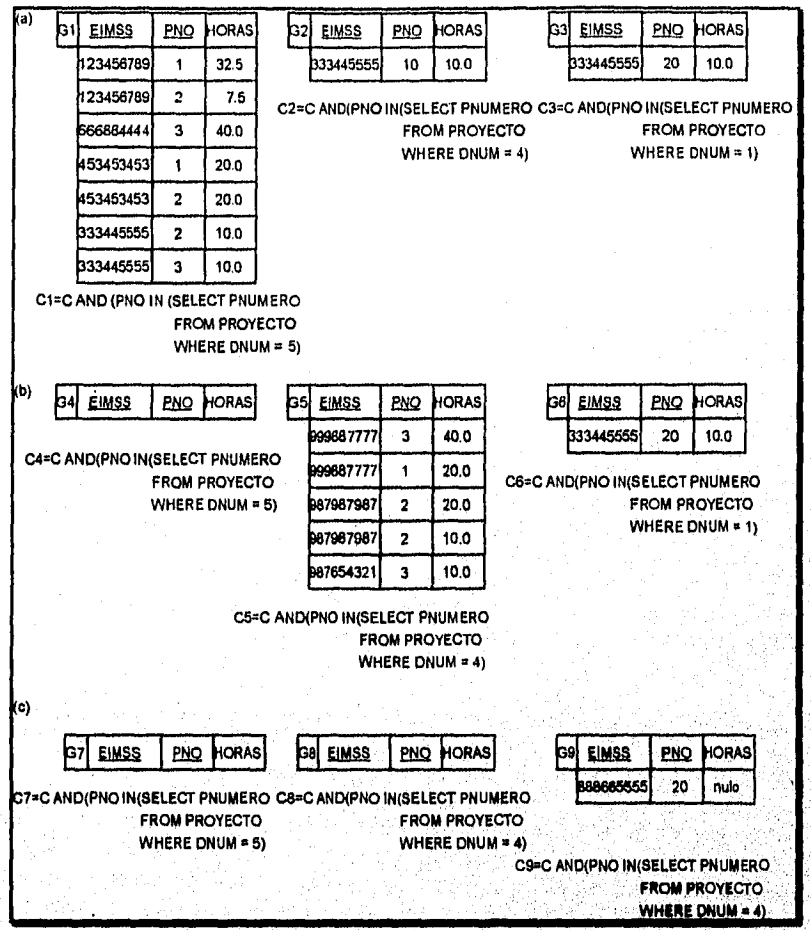


Figura 17.4 Fragmentos completos y disjuntos de la relación **TRABAJA_EN**. (a) Fragmentos de TRABAJA_EN para empleados que trabajan en el departamento 5. (b) Fragmentos de TRABAJA_EN para empleados que trabajan en el departamento 4. (c) Fragmentos de TRABAJA_EN para empleados que trabajan en el departamento 1.

En un extremo del espectro de la autonomía, tenemos un DDBMS que "se parece a" un DBMS centralizado para el usuario. Existe un solo esquema conceptual, y todo el acceso al sistema se obtiene a través de un cliente, así no existe autonomía local. En el otro extremo encontramos un tipo de DDBMS llamado **DDBMS federado** (o **sistema multibase de datos**). En tal sistema, cada servidor es un DBMS independiente autónomo y centralizado que tiene a sus propios usuarios locales, transacciones locales, y el DBA tiene un grado de *autonomía local* muy alto. Cada servidor puede autorizar acceso a una

porción en particular de su base de datos especificando un **esquema de exportación**, el cual especifica la parte de la base de datos que puede accesarse por cierto tipo de usuarios no locales. Un cliente dentro de tal sistema es esencialmente una interface adicional a diferentes servidores (DBMSs locales) que permiten que un usuario multibase de datos (o global) accese a los datos almacenados en varias de estas *bases de datos autónomas*. Notemos que un sistema federado es un híbrido entre sistemas centralizados y distribuidos; es un sistema centralizado para usuarios locales autónomos y es un sistema distribuido para usuarios globales.

En un sistema multibase de datos heterogéneo, un servidor puede ser un DBMS relacional, otro un DBMS de red, y un tercero un DBMS jerárquico; en tal caso es necesario tener un sistema canónico de lenguaje e incluir traductores de lenguaje en el cliente para traducir subqueries del lenguaje canónico al lenguaje de cada servidor.

Un tercer aspecto que puede emplearse para categorizar bases de datos distribuidas es el **grado de transparencia de la distribución**, o alternativamente, el **grado de integración del esquema**. Si el usuario ve un esquema integrado sencillo sin ninguna información concerniente a la fragmentación, replicación o distribución, el DDBMS se dice tener un *alta grado de transparencia en la distribución* (o esquema de integración). Por otro lado, si el usuario ve toda la fragmentación, localización y replicación, el DDBMS *no tiene transparencia de distribución* y ningún esquema de integración. En el último caso, el usuario debe referir a copias de fragmentos específicos en sitios específicos cuando formula un query, pero agregando el nombre del sitio antes del nombre de la relación o fragmento.

Esto es parte del complejo problema del **nombramiento** en sistemas distribuidos. En el caso de que un DDBMS no proporcione transparencia de distribución, es cuestión del usuario *especificar sin ambiguedades* el nombre de una relación en particular o copia del fragmento. Esta tarea es más severa en un sistema multi base de datos, debido a que cada servidor (DBMS local) fue desarrollado presumiblemente de manera independiente, y como resultado pueden haberse usado nombres conflictivos en diferentes servidores. En el caso de que un DBMS que proporciona un esquema integrado, el nombramiento se vuelve un problema interno del sistema, debido a que el usuario es proveído de un solo esquema. El DDBMS debe de almacenar *todas las correspondencias* entre los objetos del esquema integrado y los objetos distribuidos a través de los distintos DPs en el **catálogo de distribución**.

17.5 Procesamiento de Queries en Bases de Datos Distribuidas

Ahora daremos un panorama de cómo un DDBMS procesa y optimiza un query. Primero discutiremos los costos de comunicación para el procesamiento de un query distribuido; luego una operación especial, llamada *semijoin*, que se usa para optimizar algunos tipos de queries en un DDBMS.

17.5.1 Costos de Transferencia de Datos para el Procesamiento de Queries Distribuidos

En un sistema distribuido, existen varios factores que complican el procesamiento del query. El primero es el costo de la transferencia de datos sobre la red. Estos datos incluyen archivos intermedios que se transfieren a otros lados para mayor procesamiento, así como los archivos resultantes que pueden necesitar transferirse a un sitio donde es necesario el resultado del query. A pesar de que estos costos pueden no ser muy altos si los sitios se conectan a través de una red LAN de alto desempeño, se vuelve poco significativo en otros tipos de redes. Por lo tanto, la optimización de algoritmos de queries considera la meta de reducir la *cantidad de datos a transferir* como un criterio de optimización en la elección de una estrategia de ejecución del query distribuido.

Ilustraremos esto con dos ejemplos de queries. Supongamos que las relaciones EMPLEADO y DEPTO de la Figura 6.5 están distribuidos como se muestra en la Figura 17.5. Supondremos que ninguna relación es fragmentada. De acuerdo a la Figura 17.5, el tamaño de la relación EMPLEADO es de 100 * 10,000 = 10 ↑ 6 bytes, y el tamaño de la relación DEPTO es de 35 * 100 = 3500 bytes. Consideremos el query Q: "Para cada empleado, recuperar el nombre del empleado y el del departamento para el cual trabaja". En el álgebra relacional esto puede establecerse como sigue:

Q: $\pi_{\text{NOMBRE, APELLIDO, DNOMBRE}}$(EMPLEADO $\lozenge_{\text{DNO=DNUMERO}}$ DEPTO)

El Resultado de este query incluirá 10,000 registros, suponiendo que cada empleado se relaciona con un departamento. Supongamos que cada registro del query tiene *40 bytes de longitud*. El query se submite a un sitio 3 distinto, al cual se le llama el **sitio resultado** porque el resultado del query se necesita allá. Ninguna de las relaciones EMPLEADO y DEPTO residen en el sitio 3. Existen tres estrategias para ejecutar este query distribuido:

1. Transferir ambas relaciones al sitio resultado, y realizar el join en el sitio 3. En este caso necesitamos transferir un total de 1,000,000 + 3500 = 1,003,500 bytes.

2. Transferir la relación EMPLEADO al sitio 2, ejecutar el join en el sitio 2, y enviar el resultado al sitio 3. El tamaño del resultado del query es de 40 * 10,000 = 400,000 bytes, entonces debemos transferir 400,000 + 1,000,000 = 1,400,000 bytes.

3. Transferir la relación DEPTO al sitio 1, ejecutar el join en el sitio 1, y enviar el resultado al sitio 3. En este caso tendríamos una transferencia de 400,000 + 3,500 = 403,500 bytes.

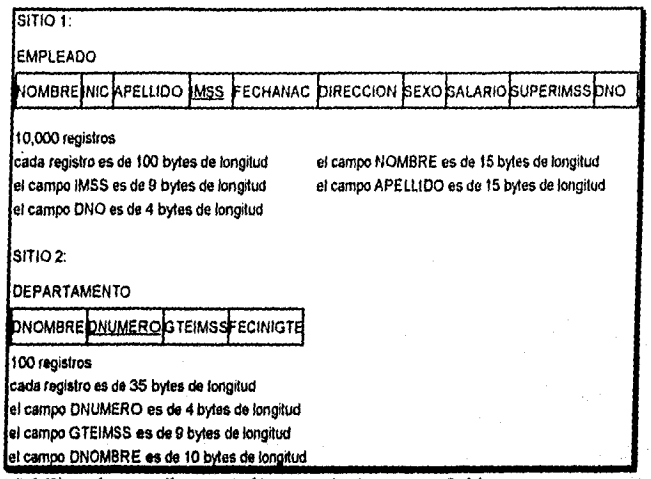


Figura 17.5 Ejemplo para ilustrar volúmenes de datos transferidos.

Si nuestro criterio de optimización es la minimización de datos a transferir, debemos elegir la estrategia 3. Ahora consideremos otro query Q': "Para cada departamento, recuperar el nombre del departamento y el del gerente". Esto puede establecerse como sigue en el álgebra relacional:

$$Q: \pi_{DNOMBRE,\ NOMBRE,APELLIDO}(DEPTO \lozenge_{GTEIMSS=IMSS} EMPLEADO)$$

Una vez más, supongamos que el query es submitido en el sitio 3. Las mismas tres estrategias para ejecutar al query Q aplican para el query Q', excepto que el resultado de Q' incluye sólo 100 registros, suponiendo que cada departamento tiene un gerente:

1. Transferir las relaciones EMPLEADO y DEPTO al sitio del resultado, y realizar el join en el sitio 3. En este caso necesitamos transferir un total de 1,000,000 + 3,500 = 1,003,500.

2. Transferir la relación EMPLEADO al sitio 2, ejecutar el join en el sitio 2, y enviar el resultado al sitio 3. El tamaño del query resultado es 40 * 100 = 4,000 bytes, así debemos transferir 4,000 + 1,000,000 = 1,004,000 bytes.

3. Transferir la relación DEPTO al sitio 1, ejecutar el join en el sitio 1, y enviar el resultado al sitio 3. En este caso tenemos que transferir 4000 + 3500 = 7500 bytes.

Una vez más, elegiríamos la estrategia 3 -en este caso por un remarcado margen sobre las estrategias 1 y 2. Las tres estrategias precedentes son las más obvias para el caso donde el sitio resultado (sitio 3) es diferente de todos los sitios que contienen archivos involucrados

en el query (sitios 1 y 2). Sin embargo, supongamos que el sitio resultado es el sitio 2; entonces tenemos dos estrategias sencillas:

4. Transferir la relación EMPLEADO al sitio 2, ejecutar el query, y presentar el resultado al usuario en el sitio 2. Aquí, necesitamos transferir el mismo número de bytes - 1,000,000- de ambos Q y Q'.

5. Transferir la relación DEPTO al sitio 1, ejecutar el query en el sitio 1, y enviar el resultado de regreso al sitio 2. En este caso debemos transferir 400,000 + 3,500 = 403,500 bytes para Q y 4000 + 3500 = 7500 bytes para Q'.

Una estrategia más compleja, la cual a veces trabaja mejor que estas estrategias sencillas, utiliza una operación llamada **semijoin**.

17.5.2 Procesamiento de Queries Distribuidos Usando Semijoin

La idea detrás del procesamiento de queries distribuido usando la operación semijoin es reducir el número de tuplas en una relación antes de transferirla a otro sitio. Intuitivamente, la idea es enviar la *columna de unión* de una relación R al sitio donde la otra relación S se localiza; esta columna entonces se une a S. Después de esto, los atributos join, junto con los atributos requeridos en el resultado, se proyectan y se regresan al sitio original y se unen con R. Por lo tanto, sólo la columna de unión de R se transfiere en una dirección, y un subconjunto de S sin tuplas extrañas se transfieren en la otra dirección. Si sólo participa una pequeña fracción de tuplas de S en el join, esta puede ser una solución poco eficiente para minimizar la transferencia de datos.

Para ilustrar esto, consideremos la siguiente estrategia para ejecutar Q o Q':

1. Proyectar los atributos join de DEPTO en el sitio 2, y transferirlos al sitio 1. Para Q, transferimos F = π_{DNUMERO}(DEPTO), cuyo tamaño es de 4 * 100 = 400 bytes, mientras que para Q', transferimos F = π_{GTEIMSS}(DEPTO), cuyo tamaño es de 9 * 100 = 900 bytes.

2. Unir el archivo transferido con la relación EMPLEADO en el sitio 1, y transferir los atributos requeridos del archivo resultante al sitio 2. Para Q, transferimos R = $\pi_{\text{<DNO, NOMBRE, APELLIDO>}}$ (F ◊ $_{\text{DNUMERO=DNO}}$ EMPLEADO), cuyo tamaño es de 34 * 10,000 = 340,000 bytes, mientras que para Q', transferimos R' = $\pi_{\text{<GTEIMSS, NOMBRE, APELLIDO>}}$ (F ◊ $_{\text{GTEIMSS=IMSS}}$ EMPLEADO), cuyo tamaño es 39 * 100 = 3900 bytes.

3. Ejecutar el query uniendo el archivo transferido R o R' con DEPTO, y presentar el resultado al usuarios del sitio 2.

Usando esta estrategia, transferimos 340,400 bytes para Q y 4800 bytes para Q'. Limitamos los atributos de EMPLEADO y las tuplas transmitidas al sitio 2 en el paso 2 a sólo aquéllos que se *unirán* con una tupla DEPTO en el paso 3. Para el query Q, esto

cambió para incluir todas las tuplas de EMPLEADO, logrando así un poco de mejora. Sin embargo, para Q' sólo fueron necesarias 100 de las 10,000 tuplas de EMPLEADO.

La operación semijoin fue desarrollada para formalizar esta estrategia. Una operación **semijoin** R ◊ $_{A=B}$ S, donde A y B son atributos compatibles de dominio de R y S, respectivamente, produce el mismo resultado que la expresión de álgebra relacional $\pi_{<R>}$ (R ◊ $_{A=B}$ S). En un medio ambiente distribuido donde R y S residen en diferentes sitios, el semijoin se implementa típicamente transfiriendo primero F = $\pi_{<B>}$ (S) al sitio donde R reside y luego uniendo F con R, llevando así a la estrategia aquí discutida.

Notemos que la operación semijoin no es conmutativa, esto es,

R ◊ S ≠ S ◊ R

17.5.3 Descomposición de Queries y Actualización

En un DDBMS *sin transparencia de distribución*, el usuario frasea un query directamente en términos de fragmentos específicos. Por ejemplo, consideremos otro query Q: "Recuperar los nombres y horas a la semana que cada empleado trabaja en algun proyecto controlado por el departamento 5", el cual se especificó en la distribución de la base de datos donde las relaciones en los sitios 2 y 3 se muestran en la Figura 17.3, y las del sitio 1 se muestran en la Figura 6.6, como en nuestro primer ejemplo. Un usuario quien submita tal query debe especificar si referencia a las relaciones PROJ5 y TRABAJA_EN5 en el sitio 2 (Figura 17.3) o a las relaciones PROYECTO y TRABAJA_EN en el sitio 1 (Figura 6.6). El usuario debe mantener también la consistencia de datos replicados cuando actualiza un DDBMS *sin transparencia de replicación.*

Por otro lado, un DDBMS que soporta *distribución completa, fragmentación, y transparencia de replicación* permite al usuario especificar un query o requesición de actualización en el esquema de la Figura 6.5 como si el DBMS estuviera centralizado. Para actualizaciones, el DDBMS es responsable de mantener la *consistencia entre los elementos replicados* usando uno de los algoritmos de control de concurrencia distribuida. Para queries, un **módulo de descomposición de queries** debe romper o **descomponer** al query en **subqueries** que puedan ejecutarse en sitios individuales. Además, para formar el query resultado debe generarse una estrategia para combinar los resultados de los subqueries. Cada vez que el DDBMS determina que un elemento referenciado en el query está replicado, se debe elegir **materializar** una réplica particular o que se referencie durante la ejecución.

Para determinar qué replicas incluyen a los elementos referenciados en un query, el DDBMS refiere a la información de fragmentación, replicación y distribución almacenada en el catálogo del DDBMS. Para fragmentación horizontal, se guarda una condición en cada fragmento, algunas veces llamada **guardia**. Esta es básicamente una condición de selección que especifica qué tuplas existen en el fragmento; se le llama *guardia* porque

sólo permite almacenar en ese fragmento a las *tuplas que satisfacen esta condición*. Para fragmentos mixtos, se almacenan en el catálogo la lista de atributos y la condición guardia.

En nuestro ejemplo anterior, las condiciones de guardia de los fragmentos del sitio 1 son VERDADEROS (todas las tuplas), y la lista de atributos son (todos los atributos). Para los fragmentos mostrados en la Figura 17.3, tenemos las condiciones de guarda y listas distribuidas mostradas en la Figura 17.6. Cuando el DDBMS descompone una requisición de actualización, puede determinar qué fragmentos deben actualizarse examinando sus condiciones de guarda. Por ejemplo, un usuario que requiere insertar una nueva tupla de EMPLEADO <'Alex', 'B', 'Coleman', '345671239', '22-ABR-64', '3306 Sandstone, Houston, TX', M, 33000, '987654321', 4> se descompondría por el DDBMS en dos requisiciones de actualización: la primera inserta a la tupla precedente en el fragmento EMPLEADO en el sitio 1, y la segunda inserta la tupla proyectada <'Alex', 'B', 'Coleman', '345671239', 33000, '987654321', 4) en el fragmento EMPD4 en el sitio 3.

Para la descomposición del query, el DDBMS puede determinar qué fragmentos pueden contener las tuplas requeridas comparando la condición del query con las condiciones de guarda. Por ejemplo, consideremos el query Q: "Recuperar los nombres y horas a la semana de cada empleado que trabaja en algun proyecto controlado por el departamento 5"; esto puede especificarse en SQL en el esquema de la Figura 6.5 como sigue:

```
Q:   SELECT   NOMBRE, APELLIDO, HORAS
     FROM     EMPLEADO, PROYECTOS, TRABAJA_EN
     WHERE    DNUM = 5 AND PNUMERO = PNO AND EIMSS = IMSS
```

Supongamos que el query se submite al sitio 2, el cual es donde será necesario el resultado. El DDBMS puede determinar que la condición de guarda en PROY5 y TRABAJA_EN5 sean todas las tuplas que satisfacen las condiciones (DNUM = 5 y PNUMERO = PNO) residen en el sitio 2. Por lo tanto, puede descomponer al query en los siguientes subqueries de álgebra relacional:

$T1 \leftarrow \pi_{IMSS}$ (PROYS5 ◊ PNUMERO=PNO TRABAJA_EN5)
$T2 \leftarrow \pi_{EIMSS, NOMBRE, APELLIDO}$ (T1 ◊ EIMSS=IMSS EMPLEADO)
$RESULT \leftarrow \pi_{NOMBRE,APELLIDO,HORAS}$ (T2 * TRABAJA_EN5)

Esta descomposición puede usarse para ejecutar el query usando una estrategia semijoin. El DDBMS conoce de las condiciones de guarda que PROYS5 contiene exactamente esas tuplas satisfaciendo (DNUM = 5) y que TRABAJA_EN5 contiene todas las tuplas a unirse con PROYS5; por lo tanto, el subquery T1 puede ejecutarse en el sitio 2, y la columna proyectada EIMSS puede enviarse al sitio 1. El subquery T2 puede entonces ejecutarse en el sitio 1, y el resultado puede enviarse de regreso al sitio 2, donde el resultado final se calcula y despliega al usuario. Una estrategia alternativa sería enviar al query Q al sitio 1, lo cual incluye todas las tuplas de la base de datos, donde sería ejecutado localmente y desde el cual el resutado sería enviado de vuelta al sitio 2. El optimizador de query estimaría los costos de ambas estrategias y elegiría al de menor costo estimado.

17.6 Panorama del Control de Concurrencia y Recuperación en Bases de Datos Distribuidas

Para propósitos de control de concurrencia y recuperación en un medio ambiente distribuido, surgen numerosos problemas. Dentro de estos problemas tenemos los siguientes:

- *Trabajar con* **copias múltiples** *de los elementos de datos*: El método de control de concurrencia es responsable de mantener consistencia entre los conceptos. El método de recuperación es responsable de hacer una copia consistente con otras copias si el sitio en el cual se almacena la copia falla y se recupera después.

- *Fallas en sitios individuales*: El DDBMS debe continuar operando con los sitos en funcionamiento, si es posible, cuando uno o más sitios individuales fallan. Cuando se recupera un sitio, antes de unir el sistema debe actualizarse su base de datos local con el resto de los sitios.

- *Falla en los enlaces de comunicación*: El sistema debe ser capáz de trabajar con la falla de uno o más enlaces de comunicaciones que conectan a los sitios. Un caso extremo de este problema es que ocurra el **particionamiento de la red**. Esto divide a los sitios en dos o más porciones, donde los sitios de cada parte pueden comunicarse sólo con el otro y no con los sitios de otras particiones.

- *Entrega distribuida*: Pueden surgir problemas con la entrega de una transacción que accesa bases de datos almacenadas en múltiples sitios si algunos fallan durante el proceso de entrega. Para tratar con este problema se usa a menudo el **protocolo de entrega de dos fases**.

- *Bloqueo distribuido*: El bloqueo puede ocurrir entre distintos sitios, así las técnicas para tratar con los bloqueos debe extenderse para tomar esto en cuenta.

El control de concurrencia distribuida y las técnicas de recuperación deben tratar con estos y otros problemas.

17.6.1 Control de Concurrencia Distribuida Sobre una Copia Distinguida de un Elemento de Datos

Para trabajar con elementos de datos replicados en una base de datos distribuida, un número de métodos de control de concurrencia ha sido propuesto extender las técnicas de control de concurrencia para bases de datos centralizadas. Discutiremos estas técnicas en el contexto de *bloqueo* centralizado. Extensiones similares aplican a otras técnicas de control de concurrencia. La idea es diseñar *una copia particular* de cada elemento de datos como una **copia distinguida**. Los bloqueos para este elemnto de datos se asocian *con la copia distinguida*, y todas las técnicas de bloqueo y desbloqueo se envían al sitio que contenga la copia.

```
(a)     EMPD5
        attribute list: NOMBRE, INIC, APELLIDO, IMSS, SALARIO, SUPERIMSS, DNO
        guard condition: DNO=5
        DEP5

        attribute list: *(all attributes DNOMBRE, DNUMERO, GTEIMSS, FECINIGTE)
        guard condition: DNUMERO = 5
        DEP5_LOCS

        attribute list: *(all attributes DNUMERO, LOC)
        guard condition: DNUMERO = 5
        PROJS5

        attribute list: *(all attributes PNOMBRE, PNUMERO, PLOC, DNUM)
        guard condition: DNUM=5
        TRABAJA_EN5

        attribute list: *(all attributes EIMSS, PNO, HORAS)
        guard condition: EIMSS IN (πIMSS(EMPD5))
        OR PNO IN (πPNumero(PROYS5))

(b)     EMPD4
        attribute list: NOMBRE, INIC, APELLIDO, IMSS, SALARIO, SUPERIMSS, DNO
        guard condition: DNO = 4
        DEP4

        attribute list: *(all attributes DNOMBRE, DNUMERO, GTEIMSS, FECINIGTE)
        guard condition: DNO = 4
        DEP4_LOCS

        attribute list: *(all attributes DNUMERO, LOC)
        guard condition:DNUMERO = 4
        PROJS4

        attribute list: *(all attributes PNOMBRE, PNUMERO, PLOC, DNUM)
        guard condition:DNUMERO = 4
        TRABAJA_EN4

        attribute list: *(all attributes EIMSS, PNO, HORAS)
        guard condition: EIMSS IN (πIMSS (EMPD4))
        OR PNO IN (πPNúmero(PROY84))
```

Figura 17.6 Condiciones de guarda y listas de atributos de fragmentos. (a) Guardas y listas de atributos para los fragmentos de relación en el sitio 2. (b) Guardas y listas de atributos para los fragmentos de relación en el sitio 3.

Diferentes métodos se basan en esta idea, pero difieren en el método de elegir las copias distinguidas. En la técnica de **sitio primario**, todas las copias distinguidas se guardan en el mismo sitio. Una modificación de este acercamiento es el sitio primario con un **sitio de respaldo**. Otro aproximamiento es el método de **copia primaria**, donde las copias distinguidas de los distintos elementos de datos pueden almacenarse en diferentes sitios. Un sitio que incluya una copia distinguida de un elemento de datos básicamente actúa como el **sitio coordinador** para el control de concurrencia en ese elemento.

Técnica de sitio primario. En este método un **sitio primario** está diseñado para ser el **sitio coordinador** *de todos los elementos de datos*. Por lo tanto, se guardan todos los bloqueos en el sitio, y todas las requisiciones de bloqueo y desbloqueo se envían a ese lugar. Así este método es una extensión del acercamiento de bloqueo centralizado. Por ejemplo, si todas las transacciones siguen el protocolo de bloqueo de dos fases, se garantiza la seriabilidad. La ventaja de este acercamiento es que es una simple extensión

del acercamiento centralizado y por lo tanto no es muy complejo. Sin embargo, tiene ciertas desventajas inherentes. Una desventaja es que todas las requisiciones de bloqueo se envían a un solo lugar, posiblemente sobrecargando ese sitio y ocasionando un cuello de botella. Otra desventaja es que la falla del sitio primario paraliza al sistema, debido a que toda la información bloqueada se guarda en ese sitio. Esto puede limitar la rentabilidad y disponibilidad del sistema.

A pesar de que todos los bloqueos se accesan en el sitio primario, los elementos mismos pueden accesarse en cualquier sitio en el cual residan. Por ejemplo, una vez que una transacción obtiene un bloqueo de lectura sobre un elemento de datos del sitio primario, puede accesar cualquier copia de ese elemento de datos. Sin embargo, una vez que una transacción obtiene un bloqueo de escritura y actualiza a un elemento de datos, el DDBMS es responsable de actualizar *todas las copias* del elemento de datos antes de liberar el bloqueo.

Sitio Primario con Sitio de Respaldo. Este acercamiento direcciona la segunda desventaja del método de sitio primario diseñando un segundo sitio para ser un **sitio de respaldo**. Toda la información de bloqueo se mantiene en los sitios primario y de respaldo. En caso de falla en el sitio primario, el sitio de respaldo puede sustituir al sitio primario, y puede elegirse un nuevo sitio de respaldo. Esto simplifica el proceso de recuperación de falla en el sitio primario, debido a que el sitio de respaldo toma su lugar y el procesamiento puede continuar después de que un nuevo sitio de respaldo se elige y el status de bloqueo de información se copia a ese sitio. Alenta el proceso de adquirir bloqueos, debido a que todas las requisiciones y liberación de bloqueos deben registrarse en *los sitios primario y de respaldo* antes de que se envíe una respuesta a la requisición de transacción. El problema de los sitios primario y de respaldo se sobrecaraga de requisiciones y alenta al sistema que permanece sin disminución.

Técnica de Copia Primaria. Este método intenta distribuir la carga de coordinación de bloqueos entre varios sitios teniendo distinguidas a las copias de diferentes elementos de datos *almacenados en diferentes sitios.* La falla de uno de los sitios afecta cualquier transacción que accese bloqueos sobre elementos de datos cuyas copias primarias residan en ese sitio, pero sin afectar a otras transacciones. Este método también puede usar sitios de respaldo para fortalecer la rentabilidad y disponibilidad.

Elección de un Nuevo Sitio Coordinador en Caso de Falla. Cada que un sitio coordinador falla en cualquiera de las técnicas precedentes, los sitios que siguen funcionando deben elegir a uno nuevo. En el caso del acercamiento del sitio primario *sin* sitio de respaldo, todas las transacciones de ejecución deben abortarse y reiniciarse, el proceso de recuperación es algo tedioso. Parte del proceso de recuperación involucra elegir un nuevo sitio primario y crear un administrador de procesos de bloqueo y un registro de toda la información bloqueada en ese sitio. Para métodos que usan sitios de respaldo, el procesamiento de transacciones se suspende mientras se designa al sitio de respaldo como el nuevo sitio primario y se elige un nuevo sitio de respaldo y se envían copias de toda la información bloqueada desde el nuevo sitio primario.

Si un sitio de respaldo X está a punto de convertirse en el nuevo sitio primario, X puede elegir el nuevo sitio de respaldo de entre los sitios funcionando del sistema. Sin embargo, si no existe respaldo, o si ambos sitios no funcionan, puede usarse un proceso llamado **elección** para elegir al nuevo sitio coordinador. En este proceso, cualquier sitio Y que intente comunicarse con el sitio coordinador repetidamente y falle puede asumir que el coordinador no funciona e iniciar el proceso de elección enviando un mensaje a todos los sitios funcionando proponiendo que Y se convierta en el nuevo coordinador. Tan pronto como Y recibe una mayoría de votos a favor, Y puede declarar que es el nuevo coordinador. El algoritmo de elección es algo complejo, pero esta es la idea principal detrás del método de elección. El algoritmo también resuelve cualquier intento por el cual dos o más sitios se convierten en coordinador al mismo tiempo.

17.6.2 Control de Concurrencia Distribuida Basada en Votación

Los métodos de control de concurrencia para elementos replicados discutidos anteriormente usan la idea de una copia distinguida que mantiene los bloqueos del elemento. En el **método de votación**, no existe copia distinguida; en lugar, se envía una requisición de bloqueo a todos los sitios que incluyen una copia del elemento de dato. Cada copia mantiene su propio bloqueo y puede conceder o denegar la requisición. Si una transacción que requiere un bloqueo es concesionada por una *mayoría* de las copias, retiene el bloqueo e informa *a todas las copias* que ha sido concedido el bloqueo. Si una transacción no recibe una mayoría de votos concede un bloqueo dentro de cierto *periodo de tiempo*, cancela su requisición e informa a todos los sitios de la cancelación.

El método de votación se considera un método de control de concurrencia verdaderamente distribuido, debido a que la responsabilidad de la decisión reside con todos los sitios involucrados. Estudios de simulación han mostrado que la votación tiene un tráfico de mensajes mayor entre sitios de lo que lo hacen los métodos de copia distinguida. Si el algoritmo toma en cuenta posibles fallas en el sitio durante el proceso de votación, se vuelve extremadamente complejo.

17.6.3 Recuperación Distribuida

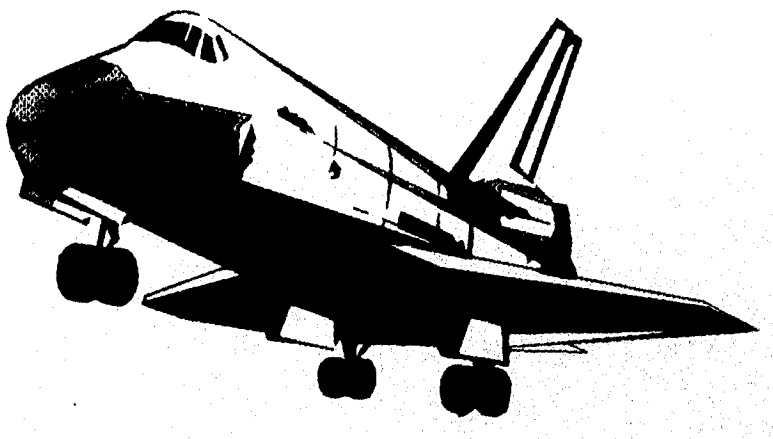
El proceso de recuperación en bases de datos distribuidas es poco involucrado. Sólo daremos una breve idea de algunos de sus problemas. En algunos casos es algo difícil determinar si un sitio no funciona sin intercambiar numerosos mensajes con otros sitios. Por ejemplo, supongamos que el sitio X envía un mensaje al sitio Y y espera un respuesta de Y pero no la recibe. Existen varias explicaciones posibles:

- El mensaje no fue entregado a Y debido a una falla de comunicaciones.
- El sitio Y no funciona y no pudo responder.
- El sitio Y funciona y envió una respuesta, pero la respuesta no fue entregada.

Sin información adicional o el envío de mensajes adicionales, es difícil determinar que es lo que sucede.

Otro problema con la recuperación distribuida es la entrega distribuida. Cuando una transacción actualiza datos en varios sitios, no puede distribuirla hasta asegurarse de que el efecto de la transacción en *cada* sitio no puede perderse. Esto significa que cada sitio debe haber registrado primero los efectos locales de las transacciones de manera permanente en el sitio local en disco. Para asegurar la corrección de la entraga distribuida, se usa con frecuencia el protocolo de entrega en dos fases.

CAPITULO 18



Tecnologías y Aplicaciones de Bases de Datos Emergentes

18.1 Progreso de la Tecnología de Bases de Datos

La Tabla 18.1 resume el progreso de la tecnología de bases de datos durante las tres últimas décadas. Las tres primeras columnas de la tabla corresponden a los tres periodos identificables. La última columna enlista los desarrollos del futuro. Discutiremos esta tabla renglón por renglón.

18.1.1 Modelos de Datos

Los modelos de red y jerárquicos emergieron en los 60's. Desde su introducción en 1970, el modelo relacional ha incrementado rápidamente su interés debido a sus propiedades, incluyendo sus bases formales, homogeneidad, conjunto de operaciones algebráicas bien definidas, y lenguajes basados en el cálculo. Los inconvenientes del modelo relacional en términos de su poder de semántica expresiva llevaron al interés en los modelos semánticos. Este ha crecido desde finales de los 70's particularmente con el advenimiento del modelo Entidad-Relación (ER), el Modelo de Jerarquía Semántica, y Modelo de Datos Semántico. En su modelo RM/T, Codd sugirió la necesidad de agregar más poder expresivo al modelo relacional agregándole ciertas abstracciones de los modelos precedentes. El interés en los modelos semánticos continuó durante los 80's. Y hoy, con horizontes de expansión de aplicaciones, la necesidad de la abstarcción está sintiéndose aún más intensamente.

Una de las mayores tendencias en los póximos años serán los modelos orientados a objetos (OO) y los sistemas administradores de bases de datos orientados a objetos (OODBMS). Los modelos OO tienen un poder expresivo similar al de los modelos de datos semánticos, yendo más allá que estos modelando explícitamente el comportamiento. Se aclaman ser más fáciles de implementar y usar para el desarrollo de aplicaciones debido a la modularidad interna, encapsulación y reutilización de código.

Logic está siendo propuesto como un modelo de datos básico para el modelo relacional y otros esquemas de representación. El cálculo relacional se basa en una rama de la lógica llamada cálculo de predicados de primer orden; por lo tanto, ya existe el uso de la lógica para caracterizar queries relacionales. Logic proporciona un formalismo que puede usarse en lenguajes query, modelos de integridad, evaluación de queries, tratamiento de valores nulos, trabajo con información incompleta, etc. Logic también conduce a un entendimiento formal de la deducción en las bases de datos.

El futuro parece observar una mayor proliferación de DBMSs orientados a objetos, mayor uso de la lógica para bases de datos deductivas, y un surgimiento de representación de conocimiento, lenguajes de programación, y modelos de datos.

Otra tendencia distinta en el desarrollo de DBMSs comerciales involucra la producción de sistemas unicampos con modelos de datos "híbridos". Un ejemplo es el sistema UniSQL el cual soporta un modelo de datos completamente orientado a objetos completamente compatible con el modelo relacional. Los vendedores de productos DBMS convencionales

ya han iniciado la implementación de una interface relacional basada en SQL para permitir el desarrollo concurrente de aplicaciones relacionales. Esto ha sucedido con DBMSs en red tales como el DMS1100 de UNISYS y el IDMS de Computer Associates, y se continuará con esa tendencia.

18.1.2 Hardware de la Base de Datos

En los últimos 30 años ha habido una revolución en el crecimiento de las capacidades de almacenamiento de las computadoras, microminiaturización de la circuitería, y velocidades de procesamiento mayores. El costo de los componentes ha caído dramaticamente. La cantidad de datos que pueden procesarse en cierta unidad de tiempo ha crecido. Sin embargo, se vuelven más factibles mayores bases de datos en equipo más pequeño. Hasta hace unos años, las bases de datos de tamaño medio sólo podían manejarse por mainframes y minis. Con el advenimiento de hardware poderoso, ahora es posible implementar mayores productos DBMS en estaciones de trabajo y computadoras personales. DBMSs relacionales tales como DB2/2, ORACLE e INFORMIX y DBMSs orientados a objetos como ObjectStore y O2 ya están disponibles para estos equipos. La funcionalidad de productos tales como Lotus 123, la serie dBase, ACCESS y Paradox están continuamente expandiéndose con hardware más poderoso que permite capacidades de ventaneo y gráficas así como altas velocidades.

Aunque los costos de almacenamiento han declinado y las capacidades de almacenamiento de los sistemas han crecido de manera considerable, el desempeño global de de los sistemas de bases de datos sigue siendo una preocupación principal. Además, los avances en software, los cuales incorporan sofisticados modelajes de datos, y arquietctura I/O, han sido muy graduales comparados con los procesadores y la arquitectura de I/O. Esto se debe a la no correspondencia entre las velocidades de acceso de los dispositivos de almacenamiento secundario y las unidades centrales de proceso, llevando a retrasos inevitables en el procesamiento de grandes volúmenes de datos almacenados en disco. Comparado con los avances en las velocidades de procesadores (atribuido a nuevas tecnologías lógicas así como a la disponibilidad del procesamiento en paralelo), el rango de incremento en velocidades de dispositivos de almacenamiento secundario (principalmente discos) ha sido lento.

Para tratar con estos problemas, se han hecho varias sugerencias para que hardware especial trabaje en las funciones de administración de datos. Estas alternativas, generalmente conocidas como **máquinas de bases de datos** o computadoras, incluyen procesadores back-end, dispositivos inteligentes (lógica por pista), sistemas multiprocesador, sistemas de memoria asociativa, y procesadores de propósito específico. La Máquina de Base de Datos Inteligente (IDM), introducida por Britton-Lee en 1982 es la primer máquina de bases de datos comercial. DBC/1012 de Teradata es algo reciente; es la única computadora de bases de datos multiprocesador disponible comercialmente que tiene una arquietctura reconfigurable muy flexible. Otra máquina de bases de datos llamada DBC/1012 de Teradata, ha sido introducida al mercado. Una implementación en paralelo del sistema manejador de bases de datos del DBMS relacional ORACLE existe en

el hardware de procesamiento en paralelo de KSR, así como en el procesador de Ncube. La investigación de arquiteturas alternas y la simulación de su desempeño son actividades de hoy en día. Típicamente, la mayoría de ellos han permanecido como diseños en papel o pequeños prototipos debido a la falta de recuros para construir hardware a gran escala o aún prototipos.

Se dará mucha atención en el futuro a máquinas de bases de datos y arquitecturas de procesamiento en paralelo debido a las demandas de transacción de procesamiento de alto volúmen y sobre el incremento de velocidades de los procesadores (especialmente en supercomputadoras). De la experiencia de los últimos 15 años, es difícil predecir si emerjerán soluciones comerciales viables y factibles.

Desde el punto de vista de nuevo hardware de almacenamiento, los dispositivos de almacenamiento óptico (los cuales son escritos concurrentemente una vez y de lectura para siempre) se están modificando en dispositivos de lectura-escritura y pronto serán económicamente accesibles. Tienen muy altas capacidades (del rango de 10 ↑ 12 bytes) y abren la posibilidad de almacenar vastas cantidades de información registrando información actualizada sin borrar la información anterior. Los discos ópticos de lectura-escritura ya han sido introducidos al mercado. Aún con el costo moderado, los dispositivos ópticos de una sola escritura disponibles hoy en día, los cuales tienen un rango de una página por segundo, que usan un disco para una carga normal pueden durar un año. Esto abre un número de nuevos prospectos excitantes para el almacenamiento y recuperación histórico de datos.

Tabla 18.1 Tendencias en la Tecnología de Bases de Datos

	60's a mediados de los 70's	70's a mediados de los 80's	80's a mediados de los 90's	Futuro
Modelo de Datos	Red Jerárquico	Relacional	Semántica Orientada a objetos Logic	Surgen modelos de datos con conocimiento Modelos Híbridos
Hardware de Bases de Datos	Mainframes	Mainframes Minis PCs	Pcs más veloces Estaciones de Trabajo Máquinas de Bases de Datos Back Ends	Configuración cliente-servidor Procesamiento en paralelo Memorias ópticas
Interface del Usuario	Ninguna	Lenguaje query Formas	Gráficas Menús Query-por-formas	Multimedia Lenguajes naturales Entrada de habla Texto a manos libres
Interface del Programa	Procedural	Lenguajes query internos	SQL estandarizado 4GL Programación lógica	Bases de datos integradas y lenguajes de programación
Presentación y Despliegue	Reportes	Generadores de reportes	Gráficas de negocios Salida de imágenes	Administradores de presentaciones generalizados Procesamiento de conocimiento y datos heterogéneos distribuidos con información multimedia
Procesamiento	Procesamiento de datos	Procesamiento de información y transacciones	Procesamiento de transacción Procesamiento de conocimiento	Administración de bases de datos en paralelo

18.1.3 Interfaces del Usuario

Cuando los DBMSs fueron introducidos a finales de los 60's los usuarios finales no tenían acceso directo a las bases de datos. Sus interacciones se limitaban a la comunicación verbal de sus requerimientos a los programadores. El llenando de formas fue uno de los primeros modos interactivos de entradas de datos para una gran población de usuarios finales, tales como empleados de escritorio o vendedores de seguros. La mayor parte de la interacción ocurre con programas batch en lenguajes procedurales.

En los 70's, los lenguajes queries se volvieron populares. Es remarcable que, para todos los propósitos prácticos, un producto grande como IMS no tiene un lenguaje query. El modelo de red también sobrevivió sin mucho soporte de lenguaje query excepto para comandos de usuario de alto nivel que clientes y usuarios pudieran desarrollar por sí mismos. La disponibilidad de estaciones de trabajo gráficas y el ventaneo ha incrementado tremendamente el potencial de crear interfaces no estáticas al DBMS, donde un usuario no se atormenta usando una sintaxis en específico. Las interfaces que crecen y se vuelven más sofisticadas hacen un uso extensivo de las capacidades de ventaneo, menús escalables, e íconos.

La mayoría de los DBMSs claman agregar facilidades llamadas **Query by forms**. Estas permiten a los usuarios invocar queries "parametrizados" para los cuales se proporcionan parámetros al runtime y el query predefinido se recompila o interpreta y ejecuta. Otras facilidades incluyen interfaces *basadas en íconos*, donde un usuario toca un ícono o imágenes en la pantalla para formular un query, y las interfaces basadas en mouse, donde un *mouse* o pantalla *sensible al tacto* permiten el movimiento del cursor mientras se especifica un query. Algunas interfaces accesan los datos actuales y proporcionan valores existentes en la base de datos en una ventana movible de tal forma que estos valores pueden usarse como constantes en queries.

Las interfaces de **lenguaje natural** han sido exploradas por algun tiempo. Para interpretar el lenguaje natural se usan tres aproximaciones básicas:

- *Extracción de instrucciones o correspondencia de patrones*: En los sistemas de instrucciones, el programa relaciona las palabras de lenguaje natural a campos específicos de una base de datos, donde un desarrollador de aplicaciones define las relaciones. Sin embargo, la correspondencia de patrones sin bases gramaticales es de uso limitado.

- *Parseo*: El parseo convierte una frase potencialmente ambigua a Inglés o a un formato interno que debe representar exactamente al query intentado por el usuario. Sobre este acercamiento son posibles dos variantes: una basada en una gramática del lenguaje natural, y otra basada en la semántica del lenguaje en términos de un léxico y una serie de reglas de producción. El sistema **LADDER** usaba una gramática semántica para accesar bases de datos de la Marina Estadounidense. El sistema **INTELLECT** de Harris, ahora un sistema comercial, se basa también en reglas gramaticales. El sistema

Rendezvous de Codd usaba queries de un léxico y controvertido lenguaje natural en cálculo relacional basado en una gramática de frases.

- *Mapeo de queries*: Un conocimiento base o "modelo del mundo" se usa con representaciones canónicas de sentencias de un cierto dominio de aplicación, junto con conocimiento de linguiística y de mapeo, para mapear queries de lenguaje natural a nivel conceptual a queries de bases de datos en un lenguaje específico.

Los sistemas de lenguaje natural de la actualidad sufren del hecho de que representan una pesada carga, requieren de la construcción de un léxico en específico, y permiten ambiguedades en la especificación de queries. Como resultado, el usuario puede no preocuparse cómo el sistema ha identificado un query y el sistema no tiene forma de predecir exactamente lo que el usuario puede desear. Las interfaces basadas en mouse e íconos son igualmente fáciles de operar en tiempo y son mucho más económicas.

La tecnología de *reconocimiento del habla y entendimiento* no es lo suficientemente sofisticada para permitir que la entrada por voz sea viable en el futuro cercano. Esa aproximación, sin embargo, tiene un fenomenal potencial par abrir puertas de bases de datos a personas de todas las edades y todos los niveles de sofisticación y pone al alcance de las masas la creación de la instrucción asistida por computadora.

18.1.4 Interfaces de Programa

Aún hoy, el volúmen de programación en el mundo comercial, el cual puede contabilizar el 70 u 80% del procesamiento por base de datos actual, se hace con lenguajes de programación convencional -la mayor parte en COBOL, PL/1, y C. Estos lenguajes, además de FORTRAN, Ensamblador y ALGOL, han sido la base de las aplicaciones. El lenguaje C++ ha entrado en escena recientemente y está gananando rápidamente territorio, especialmente en OODBMSs.

Los 70's vieron la aparición de lenguajes query que estaban dentro de los lenguajes de programación para el desarrollo de aplicaciones a gran escala.

A partir de esta tendencia se realizó una división en dos áreas. La Primera, **Lenguajes de Cuarta Generación** (4GLs, término empleado comercialmente sin una definición precisa) se volvieron porpulares en los 80's. Permitieron al usuario dar una especificación de alto nivel de una aplicación en lenguaje 4GL. El sistema (o herramienta) entonces *genera automáticamente* el código de la aplicación. Por ejemplo, INGRES, y la mayoría de los RDBMSs proporcionan una interface de Aplicación por Formas, donde el diseñador de la aplicación desarrolla usando formas desplegadas en la pantalla en lugar de escribir un programa para ello. Los 4GL sin embargo, no han sido uniformemente aceptados para la administración de datos de propósito general y para la generación de reportes. Su soporte para bases de datos es mínimo.

Otro de los grandes desarrollos de los 80's ha sido la estandarización de SQL, primero los estándares ANSI SQL y el SQL 89, y más recientemente la sintaxis SQL2. La mayoria de los vendedores de RDBMSs están intentando cumplir con estos estándares lo cual facilitará la interoperabilidad entre estos sistemas. En este momento, SQL3 es un esbozo combina muchas características de la orientación a objetos en SQL.

Hasta la fecha el uso de lenguajes de programación lógica para la administración de bases de datos deductivas no ha tenido mucho impacto. Una razón de ello es la falta de sistemas comerciales. Esperamos que la tendencia será hacia *sistemas de programación integrados*, donde la interface y la recursión de lenguajes de programación lógica se acoplen con la manipulación eficiente de hechos en el DBMS. Con la aparición de modelos de datos orientados a objetos y relacionales, o con una interface relacional desarrollada en la parte alta de un DBMS de red, los sistemas futuros continuarán presentando múltiples opciones de interface de programa dentro de un sólo sistema para el desarrollo de aplicaciones.

18.1.5 Presentación y Despliegue

Los dispositivos originales de salida en sistemas de cómputo eran impresoras en línea y perforadoras de tarjeta. La mayoría de la salida se presentaba como reportes impresos o valores calculados perforados en tarjetas y sujetos a procesamiento posterior. El lenguaje RPG (Lenguaje Generador de Reportes) fue el inicio de una tendencia hacia la actividad de generación de reportes especializados que podrían especificarse en términos de comandos de alto nivel.

Hoy en día, la generación de reportes es una característica estandar de la mayoría de los DBMSs. Ejemplos de ello son el generador de reportes de INGRES, CULPRIT de IDMS, y los procesadores lst y rpt de UNIFY. En los lenguajes especiales de definición de reportes los usuarios especifican el formateo, espaciado, paginación, niveles de totales, encabezados, justificación, etc.

Con acceso interactivo a bases de datos para una variedad de usuarios, la generación de reportes se volvió posible invocando al generador de reportes o especificándolos a través de formas. Con el advenimiento de las PCs, los usuarios esperan gran cantidad de información en la pantalla. La mayoría de los DBMSs proporcionan salida de **gráficas de negocios** en una variedad de formas, incluyendo puntos xy usando regresión lineal, gráficas de barras, y líneas. Estos tipos de despliegues requieren que la salida sea postprocesada por un administrador de despliegue.

Integrando la tecnología de imágenes al procesamiento de bases de datos y almacenando voz digitalizada como datos, algunos de los sistemas experimentales de hoy son capaces de producir salida de imágenes y voz. El almacenamiento de imágenes se está volviendo algo común. Dentro de los siguientes años el hardware especializado para el procesamiento del habla se puede volver comercialmente disponible.

18.1.6 Naturaleza del Procesamiento

La tecnología de las bases de datos fue originalmente introducida como respuesta al problema del procesamiento de archivos y sus desventajas asociadas. Las principales aplicaciones están dentro del procesamiento de información de negocios -manejo de inventarios, nómina, reportes de ventas, etc.

Durante finales de los 70's la tecnología empezó a aplicarse en una gran variedad de dominios de diferentes disciplinas. Se generaron aplicaciones de alto nivel que globalizaron datos en información para planeación táctica y estratégica. DBMSs seguros y rentables se volvieron un recurso de información centralizada con lo cual podrían interactuar un gran número de aplicaciones/usuarios geográficamente separados. Esto ha sido posible gracias a los avances en la **tecnología de comunicaciones de datos**, incluyendo banda ancha y redes de satélites de comunicaciones. Los usuarios más poderosos de la tecnología de DBMSs la usan para el **procesamiento de transacciones**.

18.1.7 Tendencias Actuales en la Tecnología

La tendencia de la tecnología dentro de los siguientes años será dominada por los siguientes temas:

- *Medios ambientes distribuidos heterogéneos*: Grandes bases de datos centralizadas serán reemplazadas por bases de datos distribuidas. Estas bases pueden implementarse con DBMSs jerárquicos o de red, el DBMS relacional actual, o los DBMS orientados a objetos que están ganando popularidad. La arquitectura "cliente-servidor" se está volviendo extremadamente popular y proporcionará a los usuarios varias opciones con énfasis en la funcionalidad del DBMS y la localización de datos.

- *Sistemas abiertos*: Varios esfuerzos de estandarización tales como SQL Access Group, ANSI/X3H7, Microsoft ODBS, IDAPI, y ODAPI de Borland, se están esforzando para mejorar la "apertura" de sistemas de bases de datos y aplicaciones. Los sistemas que cumplan con estos estándares serán capaces de comunicarse entre ellos fácilmente. SQL2 y SQL3 jugrán un papel importante en ello. La tendencia será diseñar aplicaciones globales que corran en sistemas cliente y desplieguen datos de una variedad de servidores con protocolos de acceso estandarizados.

- *Mayor funcionalidad*: La funcionalidad más pura de los DBMSs irá más allá de las funciones convencionales, define, store, access, query y report. Se incluirá la capacidad de deducción, el procesamiento de meta-data, y mejor especificación del comportamiento a través de métodos. Además, la capacidad "activa" de hacer las bases de datos más responsables, el manejo de datos multi media, mejor procesamiento de datos temporales y parciales, DBMSs especiales para científicos, van a dominar el desarrollo de la tecnología de bases de datos en los años próximos.

- *Administración de Bases de Datos Paralelas*: A escala comercial va a ofrecerse una nueva especie de bases de datos basadas en la arquitectura MIMD (instrucciones múltiples, múltiples trayectorias de datos). El trabajo actual involucra nuevos algoritmos para la estrategia de queries paralelos en la localización de datos y uso efectivo del sistema completo para minimizr el costo por transacción. Difícilmente cualquier trabajo hecho hasta la fecha ha establecido cómo debe paralelizarse un arreglo de bases de datos para beneficiarse del futuro de DBMSs en paralelo.

18.2 Aplicaciones de Bases de Datos

La aplicación de la tecnología de las bases de datos se expande continuamente. Debido a que los datos son el corazon en cualquier sistema, conforme la computación se vuelve cada vez más accesible, cada disciplina saldrá con su conjunto de aplicaciones propias. A continuación identificaremos las categorías principales de aplicaciones que se reconocen hoy en día como prospectos con buen potencial así como los grandes retos para la tecnología de las bases de datos. Se subrayarán las características especiales del modelado de cada aplicación.

18.2.1 Diseño de Ingeniería y Manufactura

La meta más difícil del diseño integrado a la computadora y la fabricación requiere del manejo efectivo de información de diseño y manufactura. Este tópico abarca sub-áreas denotadas por varios acrónimos: **CAD** (computación aplicada al diseño), **CAM** (computación aplicada a la manufactura), **CAE** (computación aplicada a la ingeniería), y **CIM** (computación integrada a la manufactura). Un acercamiento integrado para manejar la información de la manufactura requiere de la representación y manipulación compatible de información en diferentes fases del ciclo de vida del producto. Esto incluye aplicaciones de negocios tales como pronósticos, procesamiento de órdenes, planeación y control de producción, control de inventarios, y contabilidad de costos; el diseño e ingeniería del producto junto con la planeación de los requerimientos de materiales; aplicaciones relacionadas a la manufactura, incluyendo la planeación y control de los recursos; y aplicaciones de tecnología de grupos para clasificación de partes, robótica, y control del proceso de manufactura (incluyendo la programación numérica).

El *diseño de ingeniería* es un proceso exploratorio y reiterativo. La actividad del diseño de ingeniería de sistemas complejos tales como aeroplanos o automóviles se conduce por equipos de proyecto, y el diseño de un componente o subsistema evoluciona continuamente bajo un conjunto de guías de diseño, limitaciones de recursos, y contensiones de diseño. De manera intermitente, los diseños se checan cruzadamente contra otros diseños de evolución independiente, y finalmente, se almacenan las versiones o diseños "permanentes".

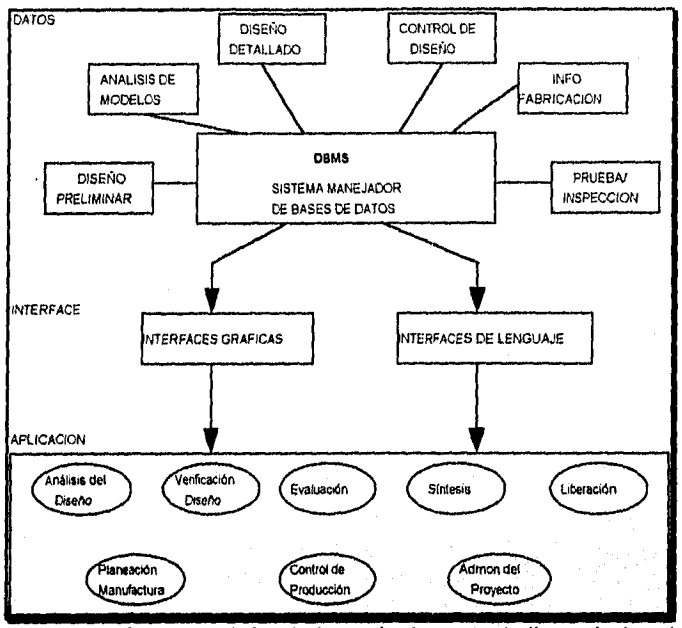


Figura 18.1 Uso de un manejador de bases de datos en el diseño de ingeniería y aplicaciones.

Rol del Manejador de Bases de Datos en CAD. Dos importantes áreas de investigación son el diseño de sistemas electrónicos VLSI (muy alta escala de integración) y el diseño de estructuras y sistemas mecánicos. Un reto básico en el diseño mecánico es el **modelado geométrico**, el cual se refiere a la representación física de partes mecánicas. La forma de las partes proporciona un hilo común a través del diseño, análisis, y ciclo de fabricación de las partes. Diferentes sistemas CAD emplean diferentes técnicas de modelado geométrico, incluyendo el modelado en alambre, modelaje de superficie, y el modelado sólido. Es traumatizante para el usuario transportar datos entre diferentes sistemas CAD o de CAD a CAM. Para facilitar dicho intercambio, el National Bureau de Estandares definió una interface neutral común o formato de datos llamado Especificación Inicial para el Intercambio de Gráficos (IGES). IGES define entidades estándares de diseño para dibujos 2D y objetos 3D así como diagramas eléctricos. Para sólidos tridimensionales, dos tipos de representaciones, llamadas representaciones de contorno y geometría sólida constructiva, son las más comúnes.

La ingeniería estructural tiene que ver con problemas que van del diseño de cimientos en construcciones hasta el diseño de complejas plataformas de lanzamiento espaciales. Ya sean construcciones, ensamblajes mecánicos, puentes, o naves espaciales, las estructuras están contenidas por geometrías y requieren de la selección de un conjunto óptimo de miembros que cumplan las contensiones de peso y diseño y proporcionen la rentabilidad

bajo diferentes cargas. Las bases de datos pueden jugar un rol de mejora en el almacenamiento de información centralmente para el uso con herramientas analíticas (tales como el análisis de elementos finitos), gráficas (típicamente gráficas tridimensionales en sistemas CAD), simulaciones, y diseño óptimo de algoritmos. En ingeniería química, las bases de datos se usan para el diseño y control de plantas. Existen dos razones principales para usar una base de datos centralizada para el diseño de datos:

- Parte del diseño puede sintetizarse, analizarse, coordinarse, y documentarse mientras que el equipo del proyecto individual trabaja en diferentes partes del diseño.

- Pueden verificarse y reforzarse automáticamente las contensiones relacionadas con estándares idénticos, diseños, especificaciones y otras propiedades físicas.

La ingeniería del diseño de datos es difícil de capturar y representar con modelos de datos y DBMSs convencionales por las siguientes razones:

- La ingeniería del diseño de datos contiene una colección no homogénea de objetos de diseño. Los modelos clásicos de datos funcionan con colecciones homogéneas.

- Los DBMSs clásicos son buenos para formatear datos (escalares), pequeñas cadenas, y registros de longitud fija. Los diseños digitalizados son grandes cadenas; tienen registros de longitud variable o información textual; con frecuencia contienen vectores y matrices de valores.

- En el diseño de operaciones de plantillas, colocación y ensamblaje son importantes las relaciones temporales y espaciales.

- El diseño de datos se caracteriza por un gran número de tipos, cada uno con un pequeño número de instancias. Las bases de datos convencionales tienen la situación opuesta.

- Los esquemas *evolucionan constantemente* en el diseño de bases de datos debido a que el diseño pasa a través de un largo periodo de evolución.

- Las transacciones en el diseño de bases de datos son de larga duración; un diseñador puede "checar" un diseño de un objeto y trabajar en él por varias semanas antes de restaurarlo (a su forma modificada) en la base de datos. De manera similar, las actualizaciones son investigaciones lejanas debido a las relaciones topológicas, funcionales, tolerancias, etc. Un reto es afectar un gran número de objetos de diseño.

- Es necesario guardar versiones anteriores y crear nuevas del mismo objeto. Debe mantenerse una bitácora de diseño para trazar la evolución del mismo y posiblemente regresar a través de él.

- En una base de datos de diseño, son funciones especializadas crear un diseño permanente, liberarlo para su producción, archivarlo, y otras.

- El diseño de datos no debe duplicarse a niveles inferiores, debido a que los elementos (tales como compuertas en circuitos eléctricos o nueces en sistemas mecánicos) se usan de manera muy repetitiva, se requiere del control de redundancia para omitir ciertos atributos comunes. Por ejemplo, cuando se usan cerrojos similares en dos lugares, las coordenadas posicionales se guardan pero otros atributos no se repiten. En tales casos, puede mantenerse una librería de diseño de objetos.

Debido a estas demandas, se están proponiendo nuevos acercamientos de modelado de datos para manejar diseños VLSI, así como diseños mecánicos. Los modelos OO son favorecidos debido a que poseen las siguientes características:

- *Modelo común*: El minimundo del diseñador puede mapearse en objetos de bases de datos mediante un mapeo uno a uno.

- *Interface uniforme*: Todos los objetos se tratan uniformemente accesándolos a través de métodos (u operaciones definidas por el usuario).

- *Soporte de objetos complejos*: Los modelos orientados a objetos para ingeniería de diseño deben permitir la creación de objetos arbitrariamente complejos involucrando jerarquías y patrones.

- *Ocultamiento de la información y soporte de abstracciones*: El mecanismo de abstracción puede proporcionar las características esenciales externas de objetos para su diseño mientras que ocultan la representación interna o detalles de implementación. La generalización y agregación son fácilmente soportados.

- *Versionamiento*: Un diseño de objeto puede sufrir una evolución a través de una serie de versiones; también puede tener diferentes representaciones. Es necesario tratar un *objeto genérico* y sus versiones de manera independiente. Al mismo tiempo, no debe existir un uso excesivo de la operación "crear versión" lo cual es opuesto a una actualización.

- *Modularidad, flexibilidad, extensibilidad, y hecho a la medida*: Las bases de datos orientadas a objetos soportan la evolución de esquemas de manera más fácil que los modelos convencionales; pueden agregarse fácilmente nuevos objetos u operaciones, y las anteriores modificarse o eliminarse.

Otro desarrollo importante es la estandarización del dato producto para propósitos de intercambio entre vendedores, diseñadores, y fabricantes. El estandar **PDES** está ahora delineando un seguimiento muy largo. Se aplica en diversas industrias tales como la construcción, componentes eléctricos, y arquitectura. Un lenguaje acompañante llamado **EXPRESS** está también ganando poularidad como un lenguaje descriptivo en la

especificación de partes, identificando varios tipos de contensiones en ellas, y proporcionando una manera uniforme de representar objetos de diseño para mayor análisis/síntesis. Los estándares PDES/STEP y EXPRESS promueven el compartimiento e integración de datos para el diseño de productos. Ambos desarrollos son los que parecen liderearán la nueva generación de sistemas y herramientas en el diseño de ingeniería y manufactura.

Debido a sus peculiaridades también existe la necesidad de un mayor trabajo en el modelaje del diseño de datos. Una dirección promisoria involucra la separación de la estructura y función en sistemas físicos complejos de tal forma que cada uno pueda representarse y analizarse independientemente. Este acercamiento facilita el modelaje de la dinámica de tales sistemas usando el acercamiento de bases de datos activas y la simulación.

18.5.2 Sistemas de Oficina y Sistemas de Soporte de Decisión

La automatización del trabajo de oficina ha sido una de las áreas de más rápido crecimiento de aplicación en los sistemas de información. La tecnología de bases de datos está teniendo un mayor impacto en el trabajo de oficina debido a que mucho de ello cae dentro de la categoría de *trabajo programable*, donde los eventos son predecibles y las respuestas conocidas. Las computadoras y, en particular, los sistemas de bases de datos pueden influenciar en gran parte a este tipo de trabajo. El otro extremo es el *trabajo creativo emergente*, de acuerdo a la taxonomía de Ciborra, donde los eventos son impredecibles y las respuestas desconocidas. El trabajo de oficina de alta gerencia, cambio de monedas internacionales, y similares caen dentro de la categoría y requiere la asistencia de sistemas de soporte de decisiones.

Sistemas de Información de Oficina Versus Aplicaciones de Bases de Datos Convencionales. Como las aplicaciones CAD, las aplicaciones OIS colocan una mayor demanda en un DBMS mientras centralizan la información en un medio ambiente de oficina. Esto se debe a las siguientes características de aplicaciones OIS:

- *Riqueza semántica*: Los datos de oficina tienden a ser semánticamente ricos y requieren soporte de mensajes no estructurados, cartas, texto, anotaciones, cadenas de direcciones, comunicación oral, etc. En una oficina existen *grupos de información estereotipada*, tales como cartas de negocios o reportes de progreso con formatos estandarizados.

- *Factor tiempo*: El factor tiempo y la contensión del tiempo deben modelarse para capturar los siguientes tipos de información: el total de tiempo de recorrido de un documento junto con una trayectoria; duración de actividades, calendarios, y programas; el tiempo de respuesta permisible para replicar un mensaje; generación de recordatorios automáticos; etc.

- *Falta de estructura*: Las actividades de oficina tienden a ser mucho menos estructuradas que otros sistemas de información. Las instrucciones para realizar el trabajo son incompletas e irregulares; son necesarias las contensiones de comunicación y diálogo.

- *Alta interconectividad*: Comparado con una manufactura que usa máquinas herramienta, una oficina representa a un grupo de elementos más complejo, cada uno efectuando una variedad de funciones e interconectada en múltiples direcciones a elementos de diferentes tipos. Por lo tanto, es difícil modelar exactamente el sobreflujo en una oficina, y las mejoras de automatización y productividad son difíciles de lograr a través de una metodología de diseño bien definida.

- *Contensiones y evolución de oficinas*: Debido a que una oficina es un grupo de humanos, es sujeto de constante evolución en la cual las autoridades y responsabilidades del trabajo de los individuos cambian. Por lo tanto es difícil modelar todas las contensiones definitivamente.

- *Interface interactiva*: Los OISs son altamente interactivos. Debe ser posible aún para el trabajador de más bajo nivel en la oficina comunicarse a un OIS. Los grupos de usuarios son diversos y abarcan el espectro completo de lo sencillo a lo sofisticado. Esto hace el reto del diseño de la interface.

- *Filtrado de información*: La mayoría de las oficinas tienen una jerarquía interna. Es necesario filtrar y globalizar la información en forma de pirámide dentro del sistema. Las transacciones de bajo nivel deben globalizarse antes de pasarse al siguiente nivel administrativo. El agregado de elementos de datos debe realizarse automáticamente a diferentes intervalos predefinidos y controlados por el usuario.

- *Prioridades, calendarización, recordatorios*: Todas estas son características importantes de diferentes elementos de trabajo; se genera una variedad de interrupciones y deben manejarse en la oficina constantemente.

Es evidente que la automatización de oficinas no es una tarea fácil. Una base de datos en el corazón de un OIS debe ser capáz de cumplir todos los requerimientos precedentes. Debido a estos factores, veremos "islas de mecanización" en oficinas de hoy en lugar de una oficina totalmente automatizada. Han sido propuestas muchas metodologías de modelado y diseño conceptual de OISs; ejemplo de ello son OFFIS, OAM, y MOBILE-Burotique. Recientes investigaciones en sistemas de información de oficinas se han concentrado en el modelado del sobreflujo de oficina. Definiendo los requerimientos de la base de datos, y trabajando con el procesamiento de queries específicos a la oficina. Un ejemplo es el proyecto TODOS, Herramientas para el Diseño de Sistemas de Oficina. La tecnología Multimedia espera jugar un rol mayor en sistemas de oficina.

Modelo Basado en Datos de un Sistema de Información de Oficina. Los modelos basados en datos agrupan la información en formas que asemejan a la papelería de una

oficina. El proyecto Oficina por Ejemplo de IBM desarrolló un lenguaje llamado Office By Example que es una extensión del lenguaje QBE. OBE modela una variedad de objetos usando el paradigma de las tablas relacionales: formas, reportes, documentos, direcciones, listas, mensajes, menús, etc. El sistema permite tipos de datos tales como imágenes, texto, y hora. Las funciones especificadas en estos objetos incluyen el procesamiento de palabras, queries, disparo de actividades automáticas (por ejemplo, enviar una carta cuando el pago se haya retrasado por más de n días, donde n es un parámetro que se obtiene de alguna tabla). Autorización, comunicación de mensajes, y manipulación de documentos son ejemplos de otras funciones manejadas a través de la misma interface. El punto principal de los modelos de oficina basados en datos es representar la oficina desde el punto de vista de objetos manipulados por trabajadores de oficina. El acercamiento orientado a objetos una vez más parece promisorio. Productos específicos para el procesamiento de documentos, correo electrónico, comunicación de audio/video, y similares están apareciendo para los usuarios de PC's con medio ambiente Windows. Un ejemplo de ello es el sistema NEWWAVE de Hewlett-Packard.

Sistemas de Soporte de Decisiones (DSSs). Los sistemas de oficina o de procesamiento de datos de negocios en general ofrecen el soporte de decisiones a trabajadores de alto nivel de diferentes maneras:

- *Análisis de datos de oficina*: Los datos deben presentarse al usuario en el nivel correcto de detalle, usando el despliegue adecuado después de realizar el filtrado correcto.

- *Control del estado del sistema*: Un OIS debe tener características que determinen su propio estado y evaluarlo. Cualquier discrepancia o condiciones excepcionales que requieren de revisión deben atraer la atención de los individuos correctos.

- *Soporte de herramientas de decisión analítica*: Además de tomar decisiones ad hoc, a los gerentes se les requiere para tomar decisiones de planeación y estrategias a largo plazo. Deben considerarse y analizarse las elecciones alternas. Un DDS debe tener conocimiento de las alternativas así como de los modelos necesarios para realizar la optimización o realizar elecciones "inteligentes".

- *Soporte de diseño organizacional y cambios en el sistema de oficina*: El DDS debe tener algunas características que permitan la reestructuración organizacional o modificación del flujo de información sin repercutir en las operaciones normales.

Los DDSs de hoy están siendo diseñados con un DBMS como su componente central. En general, un DSS permite al usuario realizar funciones de monitoreo y control, así como permitir el reforzamiento de decisiones políticas (contensiones) y acciones (procedimientos). Para permitir automatización parcial de toma de decisiones en oficinas, se puede regresar a una base de datos "activa", donde las acciones involucran la generación de mensajes, cartas, etc, o puede ser más envolvente en términos de reforzar

las políticas que abarquen el conjunto completo de funciones y bases de datos en la oficina.

18.2.3 Iniciativa del Genocidio Humano

Introducción y Metas del Proyecto. La Iniciativa del Genocidio Humano (HGI) es un buen ejemplo de una tendencia biológica que está dándole un nuevo aspecto: el de una ciencia analítica computacional. HGI es un esfuerzo internacional para la determinación de la localización de los 100,000 genes humanos que componen al genocidio humano completo. HGI es una monumental aventura científica que puede compararse con los esfuerzos del siglo XIV para dibujar el más completo y detallado mapa de la tierra, dadas las herramientas de esos días cuando se creia que la tierra era plana. Las implicaciones científicas, médicas, y económicas del HGI son profundas. HGI cimentaría la forma en que una rutina decodifica a los genes que hacen posible la vida así como proporcionar un mejor entendimiento de las enfermedades que afligen a la humanidad.

La meta más ambiciosa del proyecto genocidio es completar el mapa más detallado: una secuencia de referencias del genocidio humano completo. HGI es un proyecto a largo plazo, con una duración esperada entre 25 y 40 años. Involucra tres retos computacionales: análisis de secuencias, almacenamiento y recuperación de información, y predicción de la estructura proteínica. Estos elementos se descomponen como sigue:

- *Problema de análisis de secuencia*: Dado un conjunto de secuencias ADN, son necesarios los algoritmos correspondientes que puedan tratar elegantemente con la inserción, eliminación, sustitución y parchado en las series de secuencias de los elementos.

- *Problema de almacenamiento y recuperación*: El almacenamiento efectivo y de recuperación de datos, y el hacer la información fácilmente accesible a usuarios distribuidos mientras tratan efectivamente con errores, conflictos y actualizaciones es un gran reto -el ritmo al cual una nueva secuencia de información se generará para el cambio de siglo será de 1.6 billones de pares base al año.

- *Problema de predicción de la estructura del ADN y la proteína*: Dentro de las células, la mayoría del ADN se asocia con proteínas que, entre otras cosas, influencian el stress torsional en la doble hélice del ADN. Bajo stress torsional negativo, la hélice del ADN puede asumir estructuras alternas. Las secuencias específicas favorecen tales estructuras, y predecir las probabilidades de estructuras alternas para una secuencia dada es un gran reto. Otra dificultad es el problema del doblado de la proteína. ???Después de que una proteína es sintetizada por la célula, la cadena de proteínas de acuerdo a las leyes de la física en una forma especializada, basado en las propiedades particulares y orden de los aminoacidos. El problema de la proteina doblada se espera que rete a nuestras más poderosas supercomputadoras del futuro cercano.

En organismos multicelulares, el ADN se encuentra generalmente como dos cuerdas lineales enrolladas alrededor una de la otra en forma de una hélice doble. Una cadena de ADN es una cadena polimérica hecha de nucleótidos, cada uno de una base de nitrógeno, un azúcar desoxirribosa, y una molécula fosfatada. Al arreglo de nucleótidos junto con el espinazo del ADN se le llama secuencia del ADN. En las secuencias del ADN se usan cuatro nucleótidos: Adenosina (A), Citosina(C), Guanina(G) y Timina (T). En la naturaleza, los pares base se forman sólo entre As y Ts y entre Gs y Cs. El tamaño del genocidio está generalmente dado como su número total de pares base. Un **gene** es una región de un cromosoma cuya secuencia de ADN puede transcribirse para producir una molécula activa de ARN. Con este perfil, las metas del HGI relacionados con la computación son las siguientes:

- Establecer, mantener, y fortalecer las bases de datos que contengan información acerca de secuencias de ADN, localización de marcadores y genes, funcionamiento de los genes identificados, y otra información relacionada.

- Crear mapas de cromosomas humanos consistentes en marcadores de ADN que permitirían a los científicos localizar genes rápidamente.

- Crear reposos de materiales de investigación, incluyendo conjuntos ordenados de fragmentos de ADN que representen completamente al ADN en cromosomas humanos.

- Determinar la secuencia del ADN de una gran fracción del genocidio humano y de otros organismos.

Requerimientos de una Base de Datos Genocida. Es crucial para la comunidad científica ser capáz de accesar información sobre un tema de una variedad de bases de datos que pueden manejar diferentes aspectos del problema. Las bases de datos deben estandarizarse o trasladar fácilmente formatos e interconectarse. La base de datos genocida debe ser heterogénea e interoperable. Los principales problemas en lograr estos objetivos son los siguientes:

- Deben representarse diferentes vistas de una secuencia adecuadamente.

- Una secuencia de nucleótidos representada en una base de datos es una imágen imperfecta e inoperable de una molécula idealizada: imperfecta, porque los datos experimentales siempre contienen error; incompleta, porque siempre hay algo más que aprender de ella; e idealizada, porque ordinariamente se ignora la variación.

- Los conceptos son fluidos, y su significado cambia todo el tiempo. Actualmente, ninguna tecnología de bases de datos soporta un medio ambiente fluido de este tipo.

- El "proceso de construcción" de l modelo del universo produce inconsistencia, ambiguedad, y contradicción. Es importante que la base de datos genocida soporte la

evolución dinámica del esquema, mientras el proceso de construcción continúa en las bases de los nuevos datos experimentales e inferencias.

Debido a que los esquemas y conceptos cambian continuamente, el lenguaje query debe ser lo suficientemente poderoso para satisfacer las necesidades de los usuarios, cada uno de los cuales tiene un diferente grado de experiencia y conocimiento. La base de datos genocida debe realizar correpondencia de patrones para identificar secuencias relevantes, parches, etc, requiriendo de un medio poderoso de correpondencia de patrones.

La elección y el diseño del modelo de datos que refleja la biología del problema son por sí mismos los mayores retos. El modelo relacional es pobremente aconsejable para una base de datos genocida debido a que falla al capturar el orden en una secuencia. Existen modelos orientados a objetos que son útiles en la captura de objetos complejos y poder realizar el chequeo de consistencia incremental. Sin embargo, tienen una débil capacidad de hacer queries. El sistema LDL, con su capacidad deductiva, se está aplicando actualmente a este problema del genocidio E.coli.

Esfuerzos actuales. Las bases de datos y reposos existentes que juntan, mantienen, analizan, y distribuyen datos y materiales están ya luchando en mantener el crecimiento exponencial de la biología molecular. Algunas bases de datos se especializan en mapear y secuenciar información de un genocidio específico; por ejemplo, existen bases de datos exclusivamente devotas del ratón, para la bacteria E.coli, para drosophila, y genocidas nematodos. A continuación se proporcionan algunas bases de datos y reposos existentes en los Estados Unidos:

- Herencia Mendeliana en el Hombre (OMIMI). Este es un atlas computarizado de los rasgos humanos que se sabe se heredan -esto es, los genes.
- Librería Mapeada del Gene Humano (HGML). Esta librería consta de cinco bases de datos enlazadas -cada una con información del mapa, literatura relevante, mapas RFLP, pruebas ADN, y contactos (investigadores con información sobre datos y materiales).
- Banco de Genes. Es la mayor base de datos Estadounidense de información de secuencias del ácido nucléico de humanos y otros organismos.
- Fuente de Investigación Proteínica (PDB). Esta librería conjunta información sobre la estructura de los ácidos nucléicos, mensajeros ARN, aminoácidos, proteínas, y carbohidratos que han sido derivados de estudios cristalográficos.

18.3 Siguiente Generación de Bases de Datos y Sistemas Manejadores de Bases de Datos

En esta sección destacaremos brevemente los mayores tipos de bases de datos y la tecnología de base de datos que está asociada actualmente bajo desarrollo. Estas incluyen

a las siguientes: bases de datos activas, bases de datos multimedia, bases de datos científicas y estadísticas, y bases de datos espaciales y temporales. Seguido de una sección sobre la administración de bases de datos unificadas y extensibles, donde se subrayarán dos sistemas extensibles -EXODUS y GENESIS- y un producto comercial llamado UniSQL, el cual unifica los acercamientos orientados a objetos y relacional.

18.3.1 Bases de Datos Activas

Tradicionalmente, los DBMSs son pasivos; ejecutan queries o transacciones sólo cuando se les requiere explícitamente por un usuario o un programa de aplicación. Pero muchas aplicaciones, como el proceso y el control, redes de generación/distribución de energía, control de flujo de oficinas automatizadas, intercambio de programas, manejo de batallas, y monitoreo de pacientes en un hospital, los cuales requieren respuestas conforme pasa el tiempo a situaciones críticas, no son bien manejadas por estos DBMSs "pasivos". Para estas aplicaciones de ***contensión en el tiempo***, deben monitorearse condiciones definidas en el estado de los datos, invocarse acciones específicas, posiblemente sujetas a algunas contensiones de tiempo. Un escenario posible en la manufactura involucra el monitoreo de un evento -extensiones que salen de una máquina de ensamblaje M1 durante un intervalo - evaluación de una condición- 10% de las extensiones son defectuosas durante el intervalo de tiempo -y tomar una o más acciones- traer a la máquina M2 a la línea y notificar al personal apropiado. Todo esto puede involucrar acceso a bases de datos compartidas que están siendo constantemente actualizadas por varios usuarios y deben mantenerse en un estado consistente.

Acercamientos. Tradicionalmente, se han tomado dos acercamientos que cumplen los requerimientos de aplicaciones de contensión en el tiempo. El primero es escribir una aplicación especial que haga un poleo (queries periódicos) de la base de datos para determinar la situación que ha ocurrido en el monitoreo. El segunda es aumentar cada programa que actualice a la base de datos para checar la situación que está siendo monitoreada. El primero es difícil de implementar debido a que la frecuencia óptima de poleo es difícil de determinar; el segunda compromete la modularidad y reusabilidad de código. Las bases de datos activas soportan el monitoreo de condiciones (incluyendo activaciones y alertas) a un nivel de abstracción que tiene tres características: su semántica está bien definida; satisface los requerimientos de modelado y eficiencia de aplicaciones de bases de datos no tradicionales; y está integrada a un DBMS sin nigún parche.

Un DBMS activo continuamente monitorea el estado de la base de datos (incluyendo el reloj del sistema) y reacciona espontáneamente cuando ocurren eventos predefinidos. Funcionalmente, un sistema manejador de base de datos activa monitorea *condiciones* disparadas por *eventos* representando eventos de la base de datos (por ejemplo, actualizaciones) o eventos que no son de la base (por ejemplo, falla en el hardware detectada por un programa de diagnóstico); y si la condición a evaluar es verdadera, se ejecuta la *acción*. Un DBMS activo proporciona modularidad y respuesta en el tiempo. El acercamiento integrado a un DBMS activo que se ilustra en la Figura 18.3 puede visualizarse como el siguiente paso lógico de los sistemas de bases de datos deductivos.

En un DBMS activo, la noción de evento es generalizada y explícita. Las reglas incluyen eventos, condiciones y acciones. La Figura 18.4 muestra una base de datos automática para el reordenamiento automático de elementos en una aplicación de control de inventarios.

Temas en Bases de Datos Activas. Seis temas importantes separan a los DBMSs activos de los pasivos:

- **Eficiencia:** El conjunto de todos los eventos -condiciones- reglas de condición es como formar un gran conjunto de queries predefinidos que necesitan manejarse y evaluarse eficientemente cuando ocurren eventos especificados. La evaluación de reglas complejas bajo contensiones de tiempo es un reto cuando el conjunto de reglas que representa un medio ambiente altamente dinámico se vuelve muy grande. Esto abre nuevas posibilidades de optimización usando resultados de optimización de queries múltiples.

- **Módulos de ejecución de reglas:** Las reglas pueden saltarse y ejecutarse bajo diferentes modos: inmediato, diferido, o separado. Bajo el modo de ejecución inmediato, el procesamiento de los pasos restantes de la transacción original (esto es, la transacción de activación que ocasionó que el evento ocurriera) se suspende hasta que la regla rota haya sido completamente procesada. El tiempo de respuesta y concurrencia puede mejorar si la condición de evaluación o acción de ejecución se aparta de la transacción original (esto es, corre en una transacción separada). Si a una regla se le permite disparar a otra regla como parte de su ejecución, el modelo de transacción se vuelve anidado, llevando a una mayor complejidad.

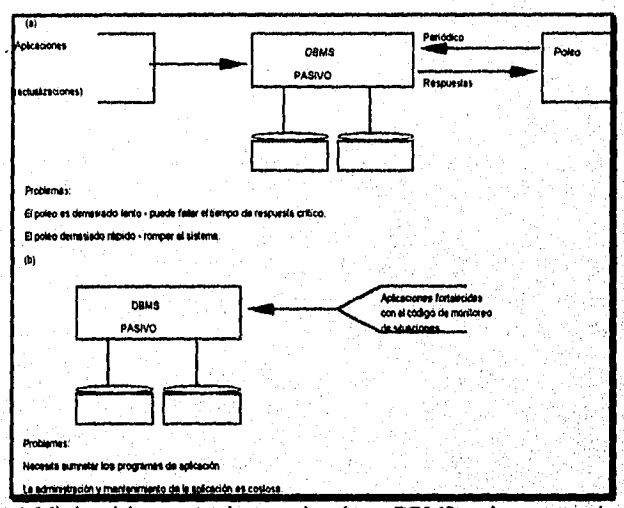


Figura 18.2 Mímica del comprtamiento activo de un DBMS pasivo convencional: (a) el acercamiento por poleo; (b) el acercamiento de aplicación fortalecida.

- **Extensiones del modelo de datos:** Los DBMSs activos requieren de fortalecimiento al modelo de datos en al menos dos formas: especificación de eventos, y especificación de condiciones y acciones. Además de los eventos que corresponden a operaciones de bases de datos (tales como insertar, borrar y modificar), eventos periódicos y/o temporales (por ejemplo, que el balance en todas las cuentas bancarias debe checarse todos los dias a las 5 PM), y soportar los eventos generados por usuarios o aplicaciones (por ejemplo, la falla de una señal de una rutina de diagnóstico en un componente del hardware). El lenguaje puede estar diseñado para soportar una especificación de eventos complejos basados en un álgebra de eventos.

- **Manejo de reglas:** Primero, la capacidad de manipular reglas (agregar, eliminar, modificar) como si fueran cualquier otro objeto en el sistema es esencial. Segundo, con frecuencia son necesarios mecanismos para habilitar y deshabilitar reglas individuales o juegos de reglas. Por ejemplo, el juego de reglas activadas mientras un avión está despegando necesita desactivarse en cuanto se eleva, y necesita activarse un conjunto de reglas diferentes para el contexto actual. Finalmente, el habilitamiento y deshabilitamiento selectivo de reglas también es importante.

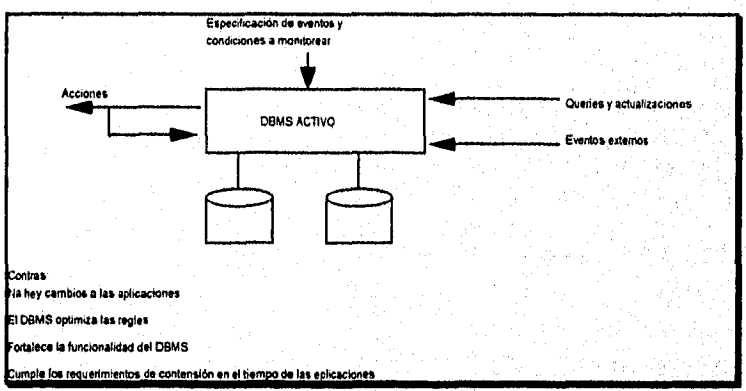


Figura 18.3 Acercamiento integrado a bases de datos activas.

- **Soporte de funciones del DBMS:** Las funciones del DBMS pueden soportarse por el DBMS activo. Las contensiones de manejo (incluyendo integridad y seguridad), mantenimiento de datos derivados (tales como vistas), e interfaces basadas en reglas son algunos ejemplos.

- **Interacción con partes del DBMS:** A diferencia de un optimizador de queries convencional, en un DBMS activo, las reglas no pueden optimizarse por separado. La optimización de reglas requiere de interacción con varios componentes del DBMS (tales como el manejador de transacciones, el manejador de objetos, y el calenadario).

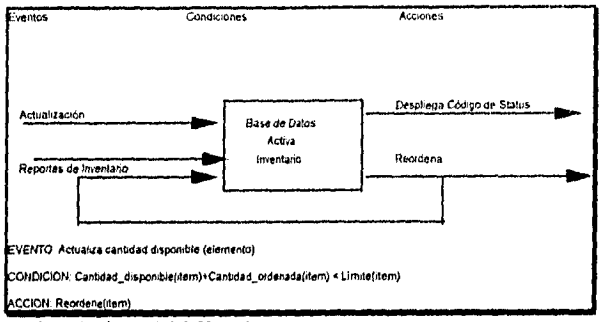


Figura 18.4 Ejemplo de un DBMS activo.

Tecnología de Punta de las Bases de Datos Activas. A continuación se da un breve panorama de la investigación y esfuerzos comerciales en al área de bases de datos activas.

HiPAC, que significa sistema de base de datos Activo de Alto Desempeño, fue una investigación enfocada al manejo de datos con contensión en el tiempo. El modelo del conocimiento de HiPAC incluyó principios de eventos, condiciones, y acciones, y formuló la noción de "reglas como objetos de primera clase". En el modelo de ejecución, se investigó la interacción entre ejecución de regla y ejecución de transacción convencional; fueron desarrollados modelos de acoplamiento y algoritmos para soportar reglas que tenían varios modos de acoplamiento. Otros varios puntos fueron direccionados como parte del proyecto HiPAC: una arquitectura de un DBMS activo; combinando procesamiento en tiempo real con bases de datos; optimización de queries múltiples; y evaluación de modelado y desempeño de los distintos modos de acoplamiento. PROBE fue extendido para incluir detección de eventos y procesamiento de condiciones consistentes de queries espaciales.

En la Universidad de Karlsruhe fue diseñado un Mecanismo Evento/Disparo (ETM) para reforzar las contensiones complejas de consistencia en el diseño de bases de datos. El esfuerzo se direccionó al problema general del reforzamiento de contensiones y fue implementado en la parte superior de un sistema de bases de datos DAMASCUS. Se mantienen tablas de eventos, acción y disparo para el acceso eficiente y determinación de disparos cuando un evento en específico ocurre. La capacidad activa es un agregado al sistema base, y consecuentemente los modos en los cuales un disparo puede ejecutarse con respecto a una transacción son limitados. En POSTGRES, comercialmente designado como Miro, las operaciones de la base de datos y otros eventos predefinidos (tales como fechas) disparan la regla de evaluación. Las reglas se expresan en QUEL usando instrucciones (tales como *always* y *demand*), y el sistema usa tipos especiales de bloqueo para los elementos mencionados en la condición de las reglas de implementación. La implementación de disparos (también llamados *procesamiento a nivel de tupla*) está dentro del ejecutor y usa tres tipos de bloqueos. Sólo soporta al modo de acoplamiento inmediato.

El proyecto Starbust de IBM se ha enfocado al problema de reglas orientadas a conjuntos disparadas por transiciones de estado en la base de datos en el contexto de bases de datos relacionales. Pueden asociarse eventos correspondientes a operaciones de bases de datos (inserción, eliminación, actualización) con una condición y una acción usando sintáxis SQL. La interacción entre ejecución de la transacción y de la regla se direcciona en detalle.

Sybase soporta disparos sencillos, donde las condiciones de disparo involucran a una sola relación, y puede visualizarse como un DBMS activo en principio. Sybase reconoce eventos que corresponden sólo a operaciones de la base de datos -llamar, insertar, borrar y actualizar. Los disparos no retornan datos al usuario y no pueden especificarse sobre vistas u objetos temporales. Un disparo no puede disparar a otro, y siempre se ejecutan de modo inmediato.

InterBase, por otro lado, no impone la mayoría de las restricciones observadas en Sybase. En InterBase, una especificación de disparo consiste de un evento, una secuencia de números asociados con el disparo, y la acción del disparo. Un evento puede ser alguna de las operaciones de la base de datos. En una relación pueden definirse cualquier número de disparos. El usuario también puede especificar si la acción del disparo se ejecuta antes o después de realizar la actividad correspondiente al evento. Se soporta la ejecución de reglas múltiples usando la información de prioridad (un entero) para resolver conflictos. Se proporciona el acceso a tuplas nuevas y anteriores a través de las instrucciones *old* y *new*.

ORACLE versión 7, INGRES, INFORMIX, proporcionan algún grado de soporte de disparo. En términos de productos DBMS comerciales, las facilidades basadas en reglas van a ser ofrecidas con mayor frecuencia en los años venideros.

18.3.2 Bases de Datos Multimedia

Las bases de datos multimedia almacenan información que se origina de diferentes medios, incluyendo datos numéricos, texto, imágenes bit map, imágenes gráficas, audio y video. También usan dispositivos de despliegue apropiado en la presentación de información. En los años venideros pueden esperarse aplicaciones a gran escala de bases de datos multimedia en las siguientes disciplinas:

- *Manejo de documentos y registros*: Un gran número de industrias y negocios guardan registros muy detallados y una variedad de documentos. Los datos pueden incluir diseño de ingeniería y de manufactura, registros médicos de pacientes, material publicitario, y registros de reclamos de seguros, dependiendo de la organización.

- *Diseminación del conocimiento*: El modo multimedia es muy efectivo en la diseminación del conocimiento. Como tal, esperamos ver libros electrónicos y reposos de información sobre muchos temas del dominio público.

- *Educación y entrenamiento*: A través de recursos multimedia pueden diseñarse materiales educativos para diferentes audiencias, desde jardines de niños hasta operadores de equipo profesional.

- *Marketing, publicidad y viajes*: No existen límites para usar información multimedia, en estas aplicaciones.

- *Monitoreo y control en tiempo real*: Acoplada con la tecnología de bases de datos activas, la presentación multimedia de información puede ser un medio muy efectivo de manipular y controlar operaciones de manufactura, plantas de energía nuclear, pacientes en unidades de cuidados intensivos, sistemas de transporte, etc.

Puntos Sobresalientes en las Bases de Datos Multimedia. Las aplicaciones multimedia tienen que ver con miles de imágenes, documentos, segmentos de audio y video, y texto libre dependen críticamente del modelado apropiado de la estructura y contenido de los datos. Los sistemas de información multimedia son muy complejos y abarcan un gran espectro de temas, incluyendo los siguientes:

- *Modelado*: Esta área tiene que ver con la controversia de aplicar bases de datos versus técnicas de recuperación de información al problema. Existen problemas que tienen que ver con objetos complejos conformados de tipos de datos que van desde los numéricos, texto, gráfico (imágenes generadas por computadora), imágenes gráficas animadas, flujo de audio, y secuencias de video. Los documentos constituyen un área especializada y merecen consideración especial.

- *Diseño*: El diseño conceptual, lógico y físico de bases de datos multimedia no ha sido investigado aún. Puede basarse en la metodología general ya descrita, pero estos puntos son más complejos a cada nivel.

- *Almacenamiento*: El almacenamiento de datos multimedia en dispositivos parecidos a disco presentan problemas de representación, compresión, mapeo a jerarquías de dispositivos, archivado, y buffereo durante las operaciones de I/O. Adheridos a estándares como JPEG o MPEG existe una forma de tratar este problema. En DBMSs, un BLOB permite que bit maps sin tipo se almacenen y recuperen. Se requerirá que software estandarizado trabaje con sincronización, compresión/descompresión, etcétera, lo cual sigue siendo dominio de la investigación.

- *Recuperación*: La forma en que la base de datos reupera información se basa en lenguajes query y estructuras de índice internamente guardadas. La recuperación de información radica fundamentalmente en instrucciones o términos índice predefinidos. Para imágenes, audio y video, esto abre nuevos puntos a tratar.

- *Desempeño*: Para aplicaciones multimedia que involucran simplemente documentos y texto, las contensiones de desempeño son subjetivamente determinadas por la población del usuario. Para aplicaciones que involucran la reproducción o

sincronización de video, dominan las limitantes físicas. Por ejemplo, el video debe entregarse a una frecuencia de 60 cuadros por segundo. Las técnicas de optimización de query pueden calcular el tiempo de respuesta esperado antes de evaluar el query. El uso de procesamiento paralelo puede aliviar algunos de los problemas, pero en la actualidad tales esfuerzos son sólo experimentales.

Los puntos anteriores han dado lugar a una variedad de problemas de investigación. Sólo hablaremos de algunos de ellos como los más representativos.

Bases de Datos Versus Perspectivas de Recuperación de Información. En modelos y sistemas de bases de datos, modelar el contenido de los datos no ha sido una novedad debido a que los datos tienen una estructura rígida y el significado de instancia puede inferirse del esquema. Por el otro lado, la recuperación de información (IR) está principlamente ocupada con el concepto de modelado de documentos de texto (instrucciones, índices, redes semánticas, etc) para el cual la estructura está generalmente abandonada. Modelando el contenido, el sistema puede determinar si el sistema es relevante a un query examinando los descriptores del documento. Consideremos, por ejemplo, un reporte de reclamos de una compañía de seguros como un objeto multimedia: contiene imágenes del accidente, formas de seguros estructuradas, registros de audio de las partes involucradas en el accidente, el documento escrito del representante de la aseguradora, y otra información. Qué modelos emplear en la representación de la información multimedia permanece como un punto abierto.

Requerimientos de Modelado y Recuperación de Multimedia/Hipermedia. Para capturar el poder completo de multimedia el sistema debe tener un constructor de modelado que permita al usuario especificar enlaces entre dos nodos arbitrarios culesquiera. Los enlaces **hipermedia** pueden tomar un número de formas diferentes, incluyendo las siguientes:

- Pueden especificarse enlaces con o sin información asociada. Un enlace puede tener grandes descripciones asociadas con él.
- Un enlace puede empezar desde un punto específico dentro de un nodo, o puede empezar desde el nodo completo.
- Un enlace puede ser direccional o no direccional.

La capacidad del enlace del modelo de datos debe contar con todas estas variantes. Cuando una recuperación basada en contenido de datos multimedia es necesaria, el mecanismo del query debe tener acceso a los enlaces y a la información asociada al enlace. El sistema debe proporcionar facilidades de definir vistas sobre enlaces, así como enlaces públicos y privados. Puede obtenerse información contextual valiosa de la información estructural. Un enlace de hipermedia automáticamente generado no revela nada nuevo acerca de estos dos nodos, debido a que el enlace fue creado automáticamente por algún medio. Por el otro lado, los enlaces hipermedia generados manualmente y la información

del enlace puede usarse para ganar mayor conocimiento acerca de los nodos que están siendo conectados. Facilidades de crear y utilizar tales enlaces, lenguajes query navegacionales para utilizar los enlaces, y otros son características importantes de cualquier sistema que permite el uso efectivo de información multimedia.

Indexado de Imágenes. Existen dos problemas con el indexado de imágenes. Uno utiliza técnicas de procesamiento de imágenes para identificar los objetos automáticamente. El otro usa técnicas de indexado manual para asignar términos y frases índice. Uno de los problemas importantes en el uso de técnicas de procesamiento de imágenes para hacer indexado se refiere a la escalabilidad. El estado tecnológico actual permite sólo algunos patrones de imágenes. La complejidad se incrementa con el número de características reconocibles. Otro problema importante se relaciona con la complejidad del query. Las reglas y mecanismos de inferencia, pueden usarse para derivar hechos de más alto nivel a partir de características sencillas de las imágenes. Esto permite queries de más alto nivel como "encontrar el diseño del departamento que permita el máximo de iluminación en la recámara" , en una aplicación arquitectónica.

La recuperación de información para el indexado de imágenes se basa en uno de los siguientes tres esquemas de indexado:

- *Sistemas clasificatorios* clasifican las imágenes jerárquicamente dentro de ciertas categorías predeterminadas. En este acercamiento, el indexador y el usuario deben tener un buen conocimiento de las categorías disponibles. Los detalles más finos de una imágen compleja y las relaciones entre objetos en una imágen no pueden capturarse.

- *Sistemas basados en instrucciones* usan un vocabulario descontrolado (alternativamente, uno controlado) como en el indexado de documentos textuales. Pueden capturarse hechos sencillos representados en la imágen (como "una región helada") y hechos derivados como resultado de interpretación a más alto nivel por los humanos (como hielo permanente, una avalancha reciente, un iceberg).

- *En el esquema entidad-atributo-relación*, se identifican todos los objetos en la imágen y las relaciones entre objetos y los atributos de los objetos.

En el caso de documentos de texto, un indexador humano puede elegir las instrucciones de la alberca de palabras disponibles en el documento a indexar. Esto no es posible en el caso de datos de imágenes y video.

Problemas Abiertos en la Recuperación de Texto. A pesar de que la recuperación de texto ha existido en aplicaciones de negocios y librerías de sistemas por varios años, los diseñadores de sistemas de recuperación de texto siguen enfrentando los siguientes problemas:

- *Indexado de frases*: Los descriptores de frase pueden llevarse a cabo mejoras sustanciales si se asignan a documentos y se usan en queries, ya que son buenos indicadores del contenido del documento y la información necesaria.

- *Uso de tesauros*: Una razón de la llamada pobreza de los sistemas actuales es que el vocabulario del usuario difiere del usado para indexar documentos. Una solución es usar un tesauro para expander los queries del usuario con términos relacionados. El problema se vuelve entonces en encontrar un tesauro del dominio en cuestión.

- *Resolver la ambiguedad*: Una de las razones de la baja precisión en sistemas de información de recuperación de texto es que las palabras tienen varios significados. Una forma de resolver esta ambiguedad es usar un diccionario en línea mientras que otro compara los contextos bajo los cuales ocurren las dos palabras.

Los sistemas de información multimedia prometen traer un matrimonio de las disciplinas de recuperación de información y manejo de bases de datos, las cuales hasta ahora, han tendido a divergir.

18.3.3 Bases de Datos Científicas y Estadísticas

El uso de computadoras para el manejo de información científica y estadística no es nuevo. Sin embargo, la tecnología de bases de datos llegó tarde a la investigación de estadísticos y científicos. En Marzo de 1990, la Fundación Nacional de la Ciencia patrocinó un taller, celebrado en la Universidad de Virginia, en el cual los representantes de la tierra, vida y ciencias espaciales se reunieron con los científicos de la computación para discutir el problema que enfrenta la comunidad científica en el área del manejo de bases de datos. Estos problemas estarán creciendo exponencialmente durante la siguiente década.

La información científica es relevante debido a su naturaleza relativamente estadística y retención indefinida. El medio ambiente de transacción-procesamiento de bases de datos convencionales está ausente de las científicas; tienen un bajo nivel de actualización, y los datos anteriores raramente se descartan. Anteriormente nos referimos al gran volúmen de datos científicos en la iniciativa del genocidio humano: que involucra a 3 billones de bases nucleótidos. El planetario Magallán generará cerca de 1 trillon de bytes de datos en un periodo de cinco años.

Los sistemas de bases de datos científicas comprenden "tres dimensiones": nivel de interpretación, análisis proyectado, y fuentes de información. En la siguiente discusión, usaremos *juego de datos* para significar los datos relacionados a un solo experimento o misión; *base de datos* se usa para cualquier agregado que se globalice o compile de varios experimentos.

Dimensión de Interpretación. Estas dimensiones se refieren al continuo reflejar de la naturaleza cruda (contra la procesada) de la información. Afecta lo que uno espera y cómo

se usarán el juego de datos. Basado en esta dimensión, los datos pueden dividirse en las siguientes categorías:

- *Datos crudos/sensor*: Consiste en valores crudos obtenidos directamente del dispositivo de medición.
- *Datos calibrados*: Consisten en valores fisicamente crudos, corregidos con operadores de calibración.
- *Datos validados*: Consisten en datos calibrados que han sido filtrados a través de procedimientos de calidad; es el tipo de dato más comúnmente usado con fines científicos.
- *Datos derivados*: Consisten de datos frecuentemente agregados, tales como datos punteados o promediados, para el cual el detalle de las mediciones base no se ha perdido.
- *Datos interpretados*: Consisten en datos derivados que se relacionan a otros juegos de datos o a la literatura del campo.

Típicamente, la información acerca del procesamiento debe retenerse y propagarse junto con un juego de datos. Puede ser necesario que se interrelacione a la información cruda y su resumen.

Análisis Científico. La mayoría de los datos científicos están sujetos a algún tipo de análisis. La información de la tierra está sujeta a análisis estadístico y de series en el tiempo. La información del genocidio se analiza por patrones sobre almacenamiento de caracteres. Los datos espaciales en grandes arreglos se transforman (por ejemplo, usando la transformada de Fourier) para análisis espectrales. El modelo relacional es inaconsejable debido a su incapacidad de representar secuencias y a la falta de representación de características de datos usados durante el análisis. Algunas veces, los datos son "corregidos" en lugar de "actualizados", pero no existen medios de lograr esto en un DBMS relacional.

Fuentes de Datos. Esta es una característica fundamental discriminatoria de las bases de datos científicas. En un medio ambiente de bases de datos monousuario, la información cruda se recolecta y posteriormente se procesa por técnicas de filtrado o calibración; la información resultante se usa en luz de la misión científica. La complejidad sintáctica y semántica de la información interpretada es mucho más alta que la de los datos originales. Las bases de datos multifuentes agregan una nueva dimensión de complejidad, con múltiples metodologías de colección, instrumentos para el registro, y diferentes protocolos que afectan la codificación y exactitud. Estas bases de datos pueden usarse y manejarse independientemente por diversos sistemas de cómputo y DBMSs. Sin embargo, la meta es crear un archivo de datos en común tal como el GENBANK de la iniciativa del genocidio

humano, para usarse por los equipos originales de investigadores que generan y manejan los datos crudos. Esta hereda los problemas de sistemas distribuidos y multi-bases.

Los DBMSs tradicionales han probado inadecuado el tratar con el conjunto anterior de requerimientos principalmente por las siguientes razones:

- Carecen de bases para tratar con redundancias e inexactitudes de datos durante experimentación científica y su registro. La progresión de datos junto con la interpretación de dimensión no puede manejarse transparentemente.
- Carece del análisis de datos usando funciones estadísticas multivariables y agregaciones.
- Los DBMSs tienen la desventaja del procesamiento de transacciones, lo cual es innecesario para la mayoría de las aplicaciones científicas.
- La distinción inherente en información estadística entre un valor y lo que significa (tal como un número (valor) representando el número de propietarios en Aragón que han entrado en el rango de $75,000-100,000 al año y tienen al menos un hijo en escuela privada) no se soporta en los DBMSs tradicionales. Abstracciones tales como el producto-cruz propuesto en el modelo de datos OSAM.

Necesitan direccionarse puntos importantes en el área de definición, manipulación, y estándares tecnológicos para cumplir con los requerimientos de bases de datos científicas y estadísticas. En el área de definición de datos, surgen los siguientes puntos:

- Necesitan explorarse nuevos modelos de datos. Deben soportar tipos especiales de datos tales como series en el tiempo y secuencias de ADN.
- El manejo de meta-data es crucial, con una provisión necesaria para trabajar con ciudades, diferentes tipos de documentos, y detalles experimentales.
- La evolución de modelos de datos debe soportarse con un apropiado sendero de auditorio.

Son necesarios estándares a través de las disciplinas científicas para registrar resultados experimentales y compartir las citas.

En el área de manipulación de datos, los siguientes puntos son relevantes:

- No es adecuado realizar queries y generación de reportes convencionales. Las funciones estadísticas y analíticas deben volverse parte integral del sistema. También es visible la tendencia de agregar una interface de lenguaje query relacional tal como SQL a paquetes estadísticos existentes tales como SAS.

- Deben proporcionarse varias operaciones de sumatoria y agregación, junto con un medio para describir la sumatoria estadística.

- Deben existir opciones de exportar e importar para transferir datos hacia y fitera de la base de datos científica y hacia utilerías estadísticas.

- Debido a que un usuario estadístico o científico es inicialmente un usuario, deben proporcionarse un rango de interfaces del usuario y facilidades de navegación. La provisión debe permitir a los usuarios "aprender" el aspecto del manejo de datos del mismo sistema a través de una graduación de estas interfaces.

En el área de estándares tecnológicos, son prominentes los siguientes puntos:

- La codificación física de datos es importante, con técnicas de compresión e indexado necesarias para el almacenamiento y recuperación eficiente. Donde se involucran factores humanos la encripción de datos es esencialmente por privacidad.

- Para experimentos a largo plazo (tales como pruebas espaciales o registro de pacientes con cierta enfermedad), los esquemas pueden tener que sufrir cambios como técnicas experimentales evolutivas.

- Son necesarios estándares que incluyen formatos de datos y análisis de herramientas asociadas dentro y a través de las disciplinas. Un ejemplo es el estandar FITS usado por la comunidad astrofísica.

- Se está volviendo factible la transmisión de juegos de datos completos sobre redes de alta velocidad. Son necesarios catálogos extensivos con nombres, alias, códigos, procedimientos de derivación de datos, y uso de información. Es necesario el archivado de juegos de datos.

De la discusión precedente es evidente que la comunidad de usuarios de bases de datos científicas tienen un conjunto de necesidades muy especiales.

18.3.4 Manejo de Bases de Datos Espaciales y Temporales

En esta sección cubriremos dos tipos de datos importantes que han empezado a recibir la atención de los investigadores de modelado de bases de datos recientemente. Mientras discutamos aplicaciones CAD/CAM, aludiremos al modelo geométrico. Ya sea en el contexto de CAD/CAM, del diseño en arquitectura o ingeniería civil, o de cartografía y geología, es importante modelar la dimensión espacio. Los modelos de datos de la actualidad carecen de esta área. La semántica espacial puede capturarse por tres representaciones comunes:

- *Representación sólida*: El espacio está dividido en trozos de varios tamaños. Las características espaciales de una entidad se representan entonces por el conjunto de estos trozos asociados con esa entidad.

- *Representación frontera (o modelos de estructura de alambre)*: Las características espaciales se representan por segmentos de líneas o fronteras.

- *Representación abstracta*: Las relaciones con semánticas espaciales, tales como ARRIBA, CERCA, JUNTO A, Y DETRAS, se usan para asociar entidades.

El proyecto PROBE en la corporación de cómputo de América proporcionó soporte para la información espacial en el modelo de datos funcional DAPLEX. Mientras es claro que muchas aplicaciones se beneficiarían de tales facilidades, diferentes dominios de aplicaciones tienen requerimientos variables. Por ejemplo, en aplicaciones VLSI, el espacio es bidimensional y discreto; se almacenan objetos básicos como puntos, segmentos de línea, rectángulos, y polígonos. En el modelado sólido la mayoría de las aplicaciones de manufactura, el espacio es tridimensional y continuo, y los objetos básicos se definen paramétricamente como curvas y superficies. Las operaciones relevantes de cada área de aplicación también difieren.

Si asumimos que un espacio 2D o 3D se representa usando segmentos de línea y polígonos, podrían hacerse los siguientes queries representativos:

- Queries de segmentos de líneas:
  - Todos los segmentos que intersectan a un punto o conjunto de puntos.
  - Todos los segmentos que tienen un conjunto de puntos finales dado.
  - Todos los segmentos que intersectan a un segmento de línea dado.

- Queries de proximidad:
  - El segmento de línea más cercano a un punto dado.
  - Todos los segmentos dentro de una distancia desde un punto dado (también conocido como rango o query ventana).

- Queries que involucran atributos de segmentos de línea:
  - Dado un punto, el mínimo de polígonos cuyos segmentos de línea son todos de un tipo en específico.
  - Dado un punto, todos los polígonos que inciden en él.

Han sido propuestas diferentes formas de almacenar objetos en el espacio, descomponiendo a este. Algunos ejemplos de ello son los siguientes:

- *Mínimo de rectángulos colindantes*: Los objetos se agrupan en jerarquías, los cuales en cambio se organizan en estructuras similares a los B-trees. Ejemplos de ellos son los R-tree y R*tree.

- *Celdas disjuntas*: Los objetos se descomponen en sub-objetos. Cada sub-objeto es una celda diferente.

- *Sangrado uniforme*: El espacio se divide creando una estructura sangrada uniforme. Los objetos aparecen en una celda especifica de esta estructura.

- *Arboles quad*: Es una estructura de resolución variable jerárquica; existen muchas variantes de ello.

La representación de objetos en el espacio y el procesamiento de queries sobre esos objetos son áreas especificas de investigación que están siendo pesadamente perseguidos por los desarrolladores de Sistemas de Información Geográfica (GISs).

Manejo de Datos Temporales. La información temporal es un caso de información espacial unidimensional. Los aspectos temporales construidos en las bases de datos deben incluir tres tipos de soporte en el tiempo; puntos en el tiempo, intervalos de tiempo, y relaciones abstractas que involucran al tiempo (antes, después, durante, simultáneamente, concurrentemente, y otras). El aspecto **histórico** de las bases de datos es muy importante para aplicaciones tales como el manejo de proyectos, historias clínicas de pacientes en los hospitales, históricos de mantenimiento de equipo, y control administrativo y operacional en sistemas de oficina.

Una limitante de las bases de datos actuales es que la información se vuelve efectiva en el mismo momento que se registra en la base de datos. No existen provisiones para hacer una distinción entre el *tiempo de registro* (o *tiempo de la transacción*) cuando se ingresa cierto dato y el *tiempo lógico* (o *tiempo de validación*) periodo durante el cual son válidos los valores de datos específicos. En efecto, algunas veces debe registrarse un tiempo de transacción para un evento. Muchas actualizaciones de bases de datos en aplicaciones reales son *retroactivas* (se vuelven efectivas en algún punto en el tiempo previo) o *proactivas* (se vuelven efectivas en algun punto del tiempo futuro). Otra limitante es que los sistemas actuales no mantienen cambios históricos. Cada actualización destruye los hechos anteriores. Así la base de datos sólo representa el estado actual de algún dominio en lugar de una historia del dominio. Para tratar con los requerimientos precedentes, son necesarios modelos de datos que incorporen tiempo, separen a la información variante en el tiempo de la que no varía y las represente independientemente. Un modelo propuesto es el Modelo Relacional Temporal (TRM). En este modelo, los atributos se dividen en variantes y no variantes en el tiempo, del mismo modo las relaciones. En las relaciones variantes en el tiempo, se agregan dos atributos en el tiempo, representando el inicio y el fin del tiempo lógico sobre el cual la tupla es válida. A este acercamiento se le llama *tupla variable en el tiempo*. La Figura 18.5 muestra una relación variante en el tiempo EMPLEADO en TRM.

Para este modelo, se propone un procedimiento de normalización en el tiempo para evitar anomalías temporales; el lenguaje SQL se extendió a SQL Temporal (TSQL) con una variedad de opciones, incluyendo la cláusula WHEN, ordenamiento temporal, periodos de

tiempo, funciones de agregar, agrupamiento, y una ventana de movimiento en el tiempo. Otra escuela propone *atributos variables en el tiempo*, lo cual agrega una nueva arquitectura a cada momento de una actualización. Esto resulta en relaciones anormalizadas con álgebra y lenguaje query mucho más complicado; además, las relaciones que involucran llaves no variables en el tiempo *no pueden* representarse desde este punto de vista.

NoEmp	Salario	Posición	Ti	Tf
33	20K	Mecanógrafa	12	24
33	25K	Secretaria	25	35
45	27K	Ingeniero Jr	28	37
45	30K	Ingeniero Sr	38	42

Notas: Ti: Hora de inicio
Tf: Hora final

Figura 18.5 Una relación variante en el tiempo en el Modelo Relacional Temporal.

Una base reciente sobre investigación de bases de datos temporal fue la publicación de una colección completa de trabajos que representan la última tecnología en este campo. La organización de esta colección indica temas en los cuales han aparecido trabajos interesantes en el manejo de bases de datos temporal en el pasado cercano:

- *Extensiones al modelo relacional*: Este trabajo ha enfatizado las diferentes formas de agregar información variable en el tiempo a un modelo relacional estático que contiene relaciones "instantáneas". Pueden agregarse diferentes tipos de tiempo usando los dos puntos de vista primarios: triple variación en el tiempo, la cual se ilustró anteriormente; y atributos variables en el tiempo, lo cual da lugar a relaciones anormalizadas (no INF).

- *Otros modelos de datos*: Este trabajo incluye extensiones al modelo ER y al modelo de datos funcional. También se ha propuesto una hoja de cálculo histórica y una base de datos deductiva.

- *Factores de implementación*: Dependiendo del modelo de datos y el lenguaje empleado el procesamiento y optimización de queries toman una nueva forma. Ha sido investigado el indexado de información temporal de varias formas y la actualización así como el control de concurrencia y recuperación del *tiempo de transacción*.

Los problemas abiertos incluyen razonamiento con información temporal, procesamiento de transacciones sobre tiempos válidos y bases de datos de transacción en el tiempo, mezclando el procesamiento temporal con bases de datos activas y deductivas, e integrando información temporal sobre medios heterogéneos.

Con el advenimiento de discos ópticos que pueden almacenar 10 ↑ 13 bytes de datos, se vuelve viable agregar actualizaciones a datos históricos. Además es necesario trabajar en los aspectos de implementación y factibilidad comercial de estos modelos. La implementación de estos modelos también coloca demandas sobre sistemas abiertos para el manejo de tiempo, registro, y sincronización de cuadros de tiempo multiusuario.

18.3.5 Manejo de Bases de Datos Extensibles y Unificados

Existe la necesidad de construir nuevos DBMSs que cumplan los retos de las nuevas aplicaciones. Los DBMSs son complejas piezas de software con algoritmos complicados y relación entre sus componentes. Debido a que toma mucho tiempo el desarrollo de DBMSs, se ha propuesto un nuevo acercameinto modular que construya DBMSs fuera de las "partes del DBMS" o construyendo bloques.

Tal **sistema manejador de base de datos extensible** ensamblaría módulos pre-escritos (algoritmos) en nuevos DBMSs. Este acercamiento tiene varias ventajas obvias:

- El desarrollo de nuevos DBMSs es rápido y económico.
- Pueden incorporarse rápidamente las mejoras tecnológicas o algoritmicas a los módulos reusables, y como resultado, el sistema puede permanecer actualizado hasta la fecha.
- Pueden evaluarse las nuevas técnicas y algoritmos propuestos sin hacer modificaciones significativas al sistema.

Este acercamiento ha sido seguido por el proyecto GENESIS en la Universidad de Texas y por EXODUS en la Universidad de Wisconsin. GENESIS toma el acercamiento de definir los componentes de un DBMS y las interfaces entre ellos de tal manera que un nuevo DBMS pueda configurarse en minutos desde una interface manejada por menús, seleccionando las opciones apropiadas. Los módulos compatibles se diseñan para accesar métodos, optimización de queries, control de concurrencia, recuperación, y otras funciones pre-implementando un juego de algoritmos alternos. Las interfaces comúnes les permiten colocarse juntos como construyendo bloques.

EXODUS está construido de manera diferente. Proporciona ciertas facilidades kernel, incluyendo un manejador de almacenamiento versatil y uno de tipos. El manejador de tipos permite la definición de jerarquías de clase con múltiple herencia. El almacenamiento de objetos no tiene tipo, secuencias de bytes de longitud variable no interpretados de tamaño arbitrario. En la parte alta del manejador de almacenamiento se proporcionan manejadores de buffers, control de concurrencia, y mecanismos de recuperación. De una librería de métodos de acceso pueden seleccionarse estructuras índice independientes de tipo, incluyendo B+trees, archivos punteados, y seccionamiento lineal. El implementador de la base de datos (DBI) proporciona el lenguaje E. Es una extensión de C agregando la noción de objetos persistentes a C, liberando al implementador de preocuparse acerca de la estructura interna de esos objetos. La capa de "métodos operador" en la arquitectura contiene una mezcla de DBI y EXODUS para operar sobre los tipos de objetos almacenados. El procesamiento de queries se divide en optimización y evaluación del query. El DBI proporciona la dscripción de operaciones, métodos para implementar los operadores, y fórmulas de costo para esos métodos. El optimizador basado en reglas transforma estas descripciones a código fuente C, lo cual constituye un optimizador para

el sistema objetivo. El manejador de almacenamiento EXODUS ha sido usado por varios vendedores para implementar sus propios DBMSs.

La configuración del acercamiento precedente, lo cual involucra generar un DBMS a la medida, contrasta con otro acercamiento propuesto para trabajar con nuevas aplicaciones: construyendo DBMSs de funcionalidad extensible proporcionando un amplio juego de características. Este **acercamiento de funcionalidad completa** está ejemplificado por los proyectos PROBE y STARBURST.

Una tercer tendencia en esta categoría es un manejador de bases de datos unificado. Un ejemplo de sistema dentro de esta categoría es UniSQL, el cual se dice combina "la sofisticación, poder, y facilidad de uso en las herramientas populares de desarrollo de aplicaciones" mientras que desarrolla la promesa de orientación a objetos y base de datos multimedia". El producto UniSQL se organiza como sigue:

- *UniSQL/X*: Este componente proporciona la plataforma cliente-servidor. Permite que aplicaciones relacionales tradicionales coexistan con la nueva base de datos orientada a objetos. El SQL/S es un lenguaje query orientado a objetos, SQL ANSI.

- *UniSQL/M*: Este componente es un manejador de bases de datos heterogéneo que permite acceso a bases de datos desde otros DBMSs relacionales y pre-relacionales. Soporta un lenguaje query orientado a objetos, SQL/M, que permite la definición y procesamiento de un esquema multi-base.

- *Herramientas UniSQL/4GE*: Object Master es una herramienta para generar aplicaciones dinámicamente a partir de la creación y manejo de objetos dentro de ambientes SQL/X y SQL/M. Otras herramientas, llamadas Visual Editor y Media Master, permiten la edición y visualización de esquemas y la generación de reportes sofisticados.

La siguiente generación de DBMSs es parecida al patrón de UniSQL. Combinarán características del modelo relacional y otros modelos de datos y propondrán la combinación de los medios activa, multimedia, científica, epacial y temporal discutidos hasta ahora.

18.4 Interfaces con Otras Tecnologías y Futura Investigación

La tecnología de las bases de datos no se desarrolla aisladamente. Está muy unida a otras disciplinas, algunas de las cuales ya han sido descritas. Por ejemplo, el área de bases de datos distribuidas y multi-bases está muy ligada a las telecomunicaciones y redes. Las bases de datos multimedia están muy ligadas a la recuperación de información. En esta sección, comentaremos algunas de estas interfaces y algunas áreas que aparecerán en el futuro cercano.

18.4.1 Interface con Ingeniería de Software

La meta principal de la ingeniería de software es desarrollar formas que faciliten el desarrollo de software, más efectivo, y eficiente. Un producto final de la ingeniería de software es el software de aplicación. Con los DBMSs volviéndose tan comunes, el desarrollo de software de aplicación debe considerarse en el contexto de los DBMSs. En las siguientes áreas debe de ocurrir la aparición de los conceptos de ambas disciplinas:

- *Diseño de bases de datos*: Una gran actividad de diseño tal como el desarrollo de grandes sistemas de software es como el diseño de proyectos en otras áreas, como la construcción de rascacielos, estaciones de energía, o una central manufacturera. El diseño de un gran sistema puede descomponerse jerárquicamente en diseños de componentes más y más pequeños. Cada componente evoluciona, cambian sus especificaciones de diseño, y aparecen nuevas alternativas. Una forma apropiada de consolidar y registrar esa información es crear bases de datos para ello.

- *Generación de aplicaciones desde la especificación a alto nivel*: Esta área es de gran interés en el desarrollo de aplicaciones comerciales. En el mecado existen varios generadores de aplicaciones comerciales, como parte de herramientas CASE (Computer Aided Software Engineering). El principal incentivo es que tales generadores reduzcan el esfuerzo y costo del desarrollo de la aplicación. Debido a que esta meta se comparte por los DBMSs, en el futuro veremos generadores de aplicaciones muy acoplados con DBMSs como base.

- *Herramientas de software*: Es un área importante de futuro desarrollo. Las herramientas de software tienen dos propósitos: reducir el tedio de ciertas tareas humanas, y mejorar el desempeño de ciertas máquinas. Junto con los DBMSs son necesarios varios tipos de herramientas. Las herramientas del primer tipo incluyen especificación de requerimientos y análisis, de diseño conceptual, herramientas para la integración de vistas, para el mapeo de esquemas entre modelos, para el diseño físico y optimización, y de particionamiento y localización. Las del segundo tipo incluyen herramientas de monitoreo de desempeño, reorganización, ajuste, y reestructuración. En los años anteriores ha habido tremenda actividad en el desarrollo de herramientas. Sin embargo, sólo unas cuantas proporcionan buenas interfaces y pueden trabajar con otras.

- *Prototipos*: Este es un tema popular entre los entusiastas de la ingeniería de software, pero ha recibido poca atención en el área de bases de datos. El prototipo de bases de datos y aplicaciones puede recorrer un gran camino validando el diseño de esquemas, refinando sus estructuras, y evaluando las frecuencias relativas de queries (transacciones). Estamos a punto de ver un desarrollo substancial dentro de esta área. La creciente disponibilidad de personal de cómputo y estaciones de trabajo hacen aún más significativo el rol del prototipo.

18.4.2 Interface con Tecnología de Inteligencia Artificial

Existe un crecimiento en el interés de cambiar de bases de datos a **bases del cococimiento** (KBs). El conocimiento es información a un nivel de abstracción más alto. Por ejemplo, "Mr Jones tiene 45 años" puede considerarse como un hecho en la base de datos. "Mr Jones tiene edad media" no es un hecho preciso; requiere de un mayor conocimiento. Similarmente, mientras que "Mr Jones tuvo un accidente en enero 18, 1988 en Nueva York, por manejar a exceso de velocidad" es un hecho, "Mr Jones es un conductor temerario" es un conocimiento. "Todos los hombres de edad media son temerarios" es otro elemento de conocimiento de nivel más alto, a pesar de que puede representar una inferencia pobre que generalmente no es correcta.

Es muy difícil cuantificar el conocimiento. El conocimiento se genera típicamente por expertos en algun dominio del conocimiento. El conocimiento se usa para definir, controlar, e interpretar los datos en la base de datos.

El conocimiento viene en diferentes formas:

- *Conocimiento estructural*: Este es conocimiento acerca de dependencias y contensiones entre los datos (por ejemplo, "la inserción en el plan de beneficencia x está sujeto a pre-registro en el plan de beneficencia y").

- *Conocimiento procedural general*: Este es conocimiento que puede describirse sólo por un procedimiento o método (por ejemplo, un procedimiento para determinar el "límite de crédito" de un cliente).

- *Conocimiento específico de la aplicación*: Este conocimiento se determina por reglas y regulaciones aplicables en un dominio específico (por ejemplo, el cálculo de la línea aérea más económica entre dos ciudades).

- *Conocimiento de dirección de empresas*: Esta es una forma más alta del conocimiento que permite que una empresa tome decisiones. Por ejemplo, en una compañía, este conocimiento incluye el costo de relocalizar y entrenar empleados así como algunas medidas del beneficio en forma de moral y lealtad manteniendo empleados laborando por más de n años.

El **conocimiento intensional** se define como el conocimiento que yace y precede el contenido de hechos de la base de datos. Tal conocimiento puede especificarse completamente *antes* de establecer la base de datos. Los sistemas de bases de datos existen para manejar los datos y meta-data, lo cual en conjunto constituye el **conocimiento extensional** o conocimiento dentro de los hechos e instancias. Los sistemas de conocimiento no sólo usan el conocimiento extensional sino también los conocimientos intensionales, posiblemente en forma de reglas en una base de reglas. El **conocimiento derivado** es una forma en la cual se mezclan el conocimiento extensional y el intensional. Una base de datos puede almacenar relaciones llamadas padre, madre y persona. De estas

tres relaciones básicas, sería posible definir una variedad de relaciones familiares, desde sobrinos a tíos y abuelos, lo cual es la tarea de la base de reglas.

La mayoría de los sistemas KB almacenan conocimiento intensional en forma de reglas. Otras representaciones incluyen aserciones lógicas, redes semánticas, y tramas. Nos referiremos a la interface y estrategia de control de un KB como *estrategia de conocimiento.*

Un área promisoria de investigación es *la mina del conocimiento,* la cual involucra extraer información previamente desconocida pero útil de una base de datos. Areas de potencial desarrollo incluyen el razonamiento basado en casos, aplicado como un razonamiento de ayuda a las bases de datos; el diseño de esquemas automatizado por medio de agrupamiento conceptual; y explicación basada en el aprendizaje, aplicada para mantener la consistencia de una base de datos, entre otros. Pueden usarse muchas estrategias de aprendizaje como nuevos paradigmas para el procesamiento de queries en sistemas de bases de datos.

18.4.3 Interfaces de Usuario para Bases de Datos

Después de las corrientes actuales en interfaces de usuarios, quedan algunas áreas por explorar, particularmente para el beneficio de usuarios de bases de datos:

- *Lenguajes personalizados*: Usuarios casuales y ocasionales necesitan de lenguajes fáciles de usar con sus problemas de dominio. Una cuenta ejecutiva en una firma de investigación tiene que ver con un juego de diferentes términos y requerimientos de procesamiento que hace un supervisor de producción en una compañía. Si ambos utilizan el mismo sistema, cada uno debe ser una interface personalizada. Actualmente, estos no son imposibles de proporcionar, pero involucran un pesado esfuerzo de programación. Los sistemas del futuro deben facilitar generar un conjunto de construcciones de lenguaje personalizables para cada conjunto de usuarios.

- *Paradigmas alternos para accesar bases de datos*: El sistema llamado Sistema Manejador de Datos Espaciales (SDMS) de Computer corporation de America es un buen ejemplo de salir del acercamiento de datos de registros y archivos para el manejo de datos. SDMS simula un medio ambiente en el cual los datos se organizan en un espacio tridimensional tal como una oficina. Basa la búsqueda de datos en los principios de que las personas son buenas para localizar información por el conocimiento de dónde se colocan y algunos atributos de apariencia del medio o contenedor en el cual se registra. Tal sistema permite una "exploración espacial" natural a través de la base de datos, motiva al despleiegue de datos, y usa íconos como diccionarios de información. Este sistema es capáz de almacenar y desplegar ilustraciones e imágenes de videodiscos y otros medios. La visualización de datos científicos y aún archivos de documentos usando técnicas inteligentes están recibiendo mucha atención. En el futuro podremos ver más desarrollos, junto con estas lineas se

incorporarían otras formas de información, incluyendo cines, animaciones, y (yendo más allá) olores y sabores!

- *Interface natural del lenguaje en varios idiomas*: Con medios de comunicación más rápidos, se está volviendo necesario hacer la misma base de datos bajo los mismos DBMSs a Americanos, Italianos, Japoneses y otros al mismo tiempo. Las interfaces de lenguaje natural de hoy son ineficientes e inadecuadas. Futuras interfaces demandarán el soporte multilenguaje. Existen problemas relacionados con el despliegue de información en diferentes scripts, permitiendo la edición de texto que involucra muchos lenguajes (por ejemplo, Español y Arabigo, donde el último se lee de derecha a izquierda), almacenando alias en diferentes lenguajes, etc.

18.4.4 Máquinas y Arquitectura de Bases de Datos

La investigación de las máquinas de las bases de datos se ha enfocado principalmente en la recuperación y almacenamiento eficiente de grandes cantidades de datos. Una futura dirección en la investigación de las máquinas de bases de datos sería diseñar una computadora que integre los componentes para la solución del problema con las funciones de manejo de datos. Otra es desarrollar hardware para el manejo de funciones de bases de datos, tales como el control de seguridad e integridad, control de concurrencia, y manejo de transacciones, que se implementan actualmente por software y constituyen una mejor fuente de desempeño. Para sistemas basados en el conocimiento también es necesaria la investigación de arquitecturas paralelas.

18.4.5 Interface con la Tecnología de Lenguajes de Programación

La investigación sobre lenguajes query -diseño y modificación de lenguajes query, optimización y compilación- es una relación en camino en relación a áreas difíciles de CAD, sistemas de oficina, y manejo de bases de datos estadísticas, espaciales, y temporales. Se requiere de más trabajo en proporcionar los modelos equivalentes y lenguajes en la formalización de sus semánticas.

Está emergiendo un nuevo campo de investigación llamado Lenguajes de Programación de Bases de Datos (DBPLs). Estos lenguajes minimizarán la "impedancia de no correspondencia" entre los lenguajes de programación tradicionales y lenguajes query de bases de datos. Sin embargo, ninguna de las propuestas hechas hasta ahora se han demostrado a una escala práctica.

Concluiremos este capítulo con la Figura 18.6, la cual muestra las aplicaciones futuras en las distintas áreas, en las cuales se aplicará el modelaje y procesamiento de la información de bases de datos, la tecnología de bases de datos debe trabajar de la mano con tecnologías de ingeniería de software, inteligencia artificial, lenguajes de programación, interfaces de usuario, y sistemas distribuidos. Los usuarios se unirán para la solución de problemas en actividades con la base de datos como foco y con estas tecnologías que soportan varias facetas del proceso de solución del problema.

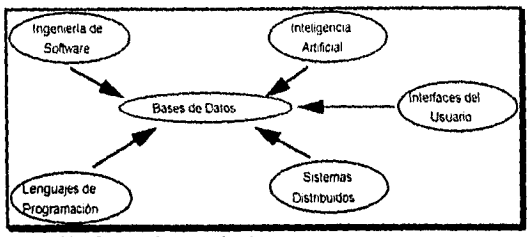
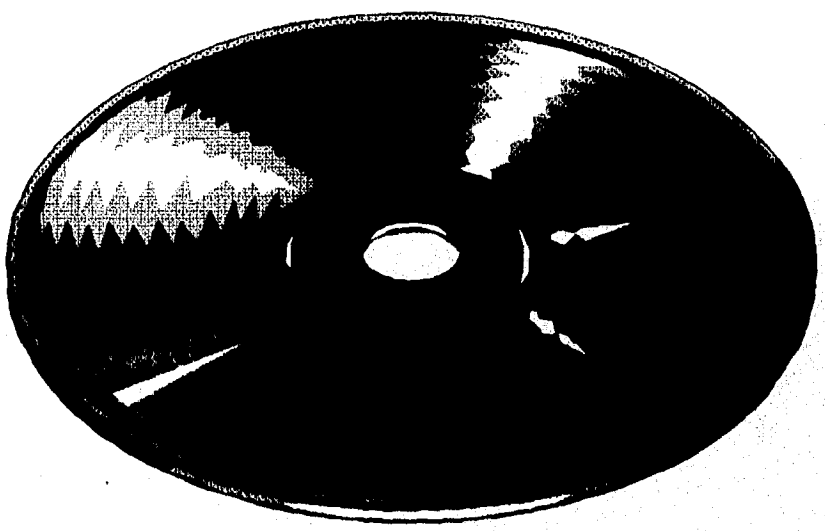


Figura 18.6 Integración futura de tecnologías.

CAPITULO 19



DBMS Paradox for Windows
y el Sistema
"CD Virtual Multimedia"

19.1 Paradox for Windows

Paradox for Windows es un sistema manejador de bases de datos relacional. Una *base de datos* es simplemente una colección de información o datos, como un tarjetero, archivero, o directorio telefónico lleno con nombres y direcciones. Cada vez que se accesa a una de estas bases de datos en papel, ya sea agregando, recuperando cambiando u ordenando información de manera significativa, se está *manejando* la base de datos.

Un *sistema manejador de bases de datos* (DBMS) como Paradox for Windows permite manejar a una base de datos almacenada en discos de computadora. La ventaja de usar una computadora en lugar del papel es quizá obvia: Tareas que pueden tomar varios minutos, horas o días en efectuarse en papel usualmente sólo toma algunos minutos para llevarse a cabo en Paradox.

19.2 Qué puede hacerse con Paradox?

Paradox for Windows es una aplicación extremadamente flexible que da virtualmente opciones ilimitadas para almacenar y manejar información. El tipo de información que puede almacenarse está únicamente limitado por las necesidades y la imaginación. A continuación se presentan algunos de los usos comunes de sistemas manejadores de bases de datos:

- Manejo de listas de correos y directorios telefónicos.
- Manejo de archivos de clientes, ventas y membresías
- Manejo de libros y tareas contables tales como, catálogos generales, cuentas por pagar y cuentas por cobrar.
- Manejo de ordenes y control de inventarios.
- Manejo de librerías personales o profesionales de fotos, descripciones, y precios de los productos de una compañía.
- Manejo e impresión de catálogos de fotos, descripciones, y precios de los productos de una compañía.
- Almacenamiento y actualización de información de los empleados, incluyendo salario y sonidos creados con aplicaciones fuera de Paradox.
- Análisis y graficación de tendencias de ventas y de compras a través del tiempo.

Sin importar el tipo de información que se quiera manejar con Paradox y cómo manejarla, existe un requerimiento que debe cumplirse: La información debe organizarse en una o más *tablas*.

Qué es una Tabla?

Una tabla es un simple cuerpo de información que se organiza en renglones y columnas. En terminologia de bases de datos, nos referiremos a cada columna de información en la tabla como un *campo* y a cada renglón de información como un *registro*. Por ejemplo, la Figura 1.1 muestra una lista de los nombres y números telefónicos organizados en una tabla, ilustrando los términos de *campo* y *registro*.

Por supuesto, la información que se piensa almacenar podría aún no estar arreglada en tablas, así que será necesario imaginar cómo podría reorganizarse la información en un formato tabular.

Por ejemplo, supongamos que tenemos un tarjtero, con cada tarjeta en el tarjetero conteniendo el nombre y dirección de un individuo, como en la Figura 1.2. Cada terjeta tiene cuatro líneas de información en ella: (1) nombre, (2) dirección, (3) ciudad, estado, código postal, y (4) número telefónico.

Para poner esta información en una tabla, imaginemos que cada terjeta representa un registro (renglón) de información. Cada elemento discreto de información en la tarjeta (nombre, dirección, ciudad, estado, código postal) representa un campo (columna).

Familiarizándose con Paradox

Terminología Básica

Base de Datos

Una *base de datos* es una colección organizada de información relacionada (o datos) almacenados para facilitar y hacer más eficiente su uso. En Paradox, una base de datos es una colección de una o más tablas utilizadas para el registro de información.

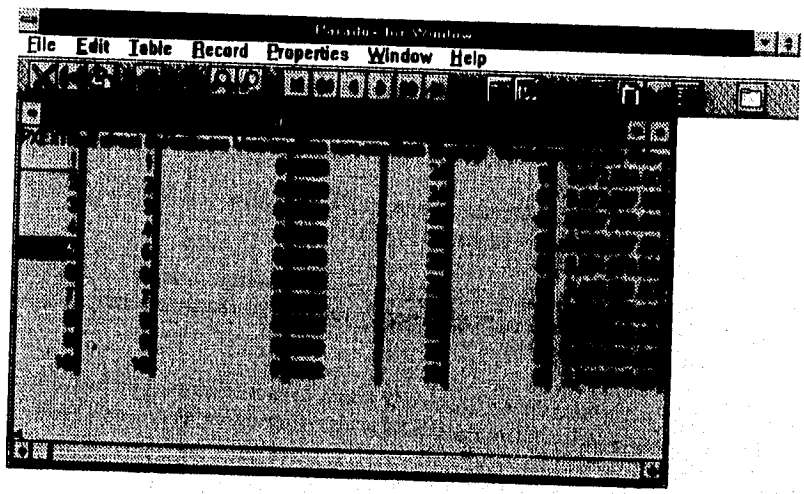
Tablas

Una *tabla* es el lugar donde se almacenan los datos. Las tablas se organizan en renglones horizontales (registros) y columnas verticales (campos). Esto facilita la examinación o modificación de los datos. La Figura muestra el ejemplo de la Tabla *Pxextr*.

Registros

Un renglon horizontal (o *registro*) en una tabla contiene información acerca de una persona, lugar o cosa individual, dependiendo del contenido de la tabla. En la tabla *Pxextr*, cada registro describe una extracción efectuada en el sistema de Computador A Bordo. Indica qué numero de extracción le corresponde, información acerca del hardware del computador, fecha de extracción, etc. De esta forma, cada renglón en la tabla *Pxextr* se

compone de varias categorías de información acerca de una sola cosa en específico -una extracción.



Campos

Una columna vertical en una tabla es un *campo*. Cada campo contiene una categoría de información acerca de la persona, lugar o cosa descrita en el registro. En la tabla *Pxextr*, Extract # es un campo, Hardware Version otro, y así sucesivamente.

Número de Registro

El *número de registro* es un contador interno que usa Paradox para distinguir a cada registro. Paradox maneja los números de registro de manera automática, de tal forma que no se pueden cambiar de manera directa.

Tipos de Campos

El *tipo de campo* le indica a Paradox que clase de información puede retener un campo en específico y qué acciones pueden efectuarse con el dato de ese campo. Algunos tipos de campos comúnes incluyen: alfanuméricos (o caracter), numérico, fecha, monetario, y memo. Las tablas de Paradox también soportan *campos blob* (grandes objetos binarios),

los cuales retienen información especializada, tales como memos formateados, imágenes gráficas, y enlaces OLE.

Archivos

Paradox guarda los trabajos efectuados en archivos.

Documentos de Diseño

Los documentos de diseño permiten presentar los datos de formas nuevas o diferentes. Paradox soporta dos tipos de documentos de diseño:

- Las *formas* permiten editar y desplegar datos de diferentes maneras, incluyendo combinar datos de más de una tabla. Algunas formas también se crean para que hagan el trabajo por nosotros.

- Los *reportes* imprimen los datos de las tablas. Como las formas, permiten definir relaciones entre tablas. A diferencia de las formas, no permiten la edición de las tablas.

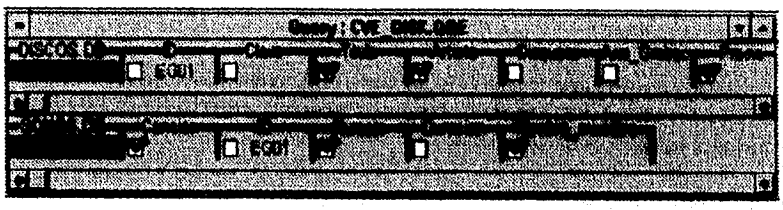
En Paradox, cada tipo de documento de diseño comparte algunas características del otro. Cada tipo de documento tiene fortalezas y debilidades, asi debe seleccionarse el tipo apropiado a las necesidades.

Queries

Un query es una pregunta que se le formula a Paradox acerca de información en las tablas. Puede ser una pregunta sencilla acerca de la información en una tabla o una pregunta compleja acerca de la información en varias tablas. Por ejemplo, se puede preguntar

> Qué clientes han necho compras?
> Cuál es el monto total de las compras efectuadas por cada cliente?
> Qué compras no han sido pagadas?

Los queries nos permiten localizar información, efectuar cálculos, y totalizar los valores en las tablas. Al método de query de Paradox se le llama QBE, o Query By Example (Query Por ejemplo). Para efectuar un QBE, se le asigna a Paradox un ejemplo del resultado que deseamos. Paradox determina la mejor forma de conseguirlo. Los queries son flexibles, interactivos e iterativos. Si un query no obtiene los resultados deseados, fácilmente se puede redefinirlo y llevarlo a cabo otra vez. La Figura muestra el ejemplo del Query *CVE_DISK*.



19.3 ObjectPAL

ObjectPAL es un lenguaje de programación visual que puede emplearse para automatizar el trabajo y construir aplicaciones sofisticadas en Paradox for Windows. El lenguaje proporciona cientos de herramientas para automatizar *todo* lo que puede hacerse en Paradox for Windows.

Una explicación global de ObjectPAL requiriría de un libro completo. Así en lugar de presentar un catálogo exhaustivo de las características de ObjectPAL y las reglas de sintáxis, nos enfocaremos a destacar los puntos importantes del lenguaje.

Entendiendo la Terminología de ObjectPAL

Qué es un objeto?

Todo lo que se puede crear en Paradox for Windows es un objeto. Por ejemplo, botones, campos, gráficas, tablas, queries, formas, y reportes son todos ellos objetos.

Cada objeto Paradox se compone de *propiedades* (tales como el color, la posición, el font y el estilo) y *métodos* (el código que define cómo se comporta el objeto). Las propiedades específicas y los métodos disponibles dependen del tipo de objeto. Por ejemplo, las propiedades y métodos disponibles para los botones son diferentes de aquéllos disponibles para las tablas.

Qué es un Evento?

Un *evento* es una acción que afecta a un objeto. Todo lo que se hace en Paradox for Windows puede generar un evento. Los tipos comúnes de eventos incluyen:

- La presión de una tecla o el click del mouse
- Abrir o cerrar una forma, reporte o tabla.
- Cambiar el valor de algun campo.
- Agregar un registro a una tabla.
- Hacer click en un botón.
- Elegir una opción de algún menú.

Qué es un Método?

Un *método* es un snippet de ObjectPAL que define cómo responde un *objeto* a un *evento*. Los métodos se activan (o *disparan*) cuando ocurren eventos específicos. Por ejemplo, el siguiente método maximiza una ventana en el Desktop cuando el usuario llega a la forma:

```
method arrive(var eventInfo MoveEvent)
        maximize()
endmethod
```

Posteriormente explicaremos al componente de cada método. Por ahora,sólo queremos dar una idea de los sencillo que se ve un método.

Los métodos se colocan típicamente a los objetos en formas, pero también pueden almacenarse en archivos independientes llamados *scripts* y en colecciones de código ObjectPAL llamadas *librerías*.

Qué es una Aplicación?

Una *aplicación* es un sistema que automatiza las tareas de administración de la base de datos. Con frecuencia las aplicaciones efectúan funciones de negocios, tales como el procesamiento de cuentas, el llenado de órdenes, y chequeo de compras en el stock. La mayoría de las aplicaciones en ObjectPAL se construyen a partir de formas personalizadas que contienen botones a los cuales se unen métodos ObjectPAL.

Los Objetos son Entes "Inteligentes"

Cuando aprendimos a diseñar formas, descubrimos que los objetos "conocen" acerca de los objetos dentro de ellos. Aún más, probablemente descubrimos que los mismos tipos de objetos siempre tienen las mismas propiedades y métodos disponibles para ellos -aún tienden a parecer y comportarse muy similarmente. Así, una forma "conoce" acerca de las tablas y campos dentro de su modelo de tatos, y todos los botones en una forma tienen la misma apariencia general y dan básicamente la misma respuesta cuando se les hace un click. Por estas razones, decimos que los objetos de Paradox son "inteligentes".

Cómo difiere ObjectPAL de los Lenguajes de Programación Tradicionales

Si alguna vez se ha desarrollado alguna aplicación en algun lenguaje de programación tradicional tal como BASIC, FORTRAN, PASCAL o C, sabemos que el proceso puede ser muy consumidor de tiempo y frustrante, debido a que se debe escribir y depurar el código del programa para *cada* acción realizada por la aplicación.

Cuando usamos Paradox for windows, una gran parte de la programación se lleva a cabo *automáticamente* (y correctamentel) cada vez que se colocan objetos sobre una forma y se le dan propiedades a esos objetos. Por qué? Porque el comportamiento estandar de cada

objeto en la forma se define automáticamente. ObjectPAL es sólo necesario si se desea automatizar una serie de pasos, cambiar el comportamiento interno de un objeto, o ejercer mayor control sobre lo que los usuarios ven y hacen. Por ejemplo, se podría usar ObjectPAL para pedir el nombre de una tabla o forma a abrir; para automatizar los diferentes pasos requeridos para actualizar inventarios y archivos históricos; o calcular automáticamente un Precio de Venta.

Para acelerar el proceso de desarrollo y reducir la cantidad de programación debemos de utilizar muchas de los métodos incluidos en ObjectPAL, así como el sistema de ayuda (Help) que contiene grandes cantidades de código que se puede pegar y revisar cuando sea necesario.

Antes de empezar el desarrollo de alguna aplicación con ObjectPAL, se deben entender las técnicas básicas de programación, y sentirse cómodo utilizando todas las opciones de Paradox. A pesar de que ObjectPAL es tremendamente poderoso para automatizar y simplificar las tareas, las características estandard de Paradox for Windows y algunas formas y reportes personalizados pueden proporcionarnos todo lo que necesitamos.

Ligando un Método a un Objeto

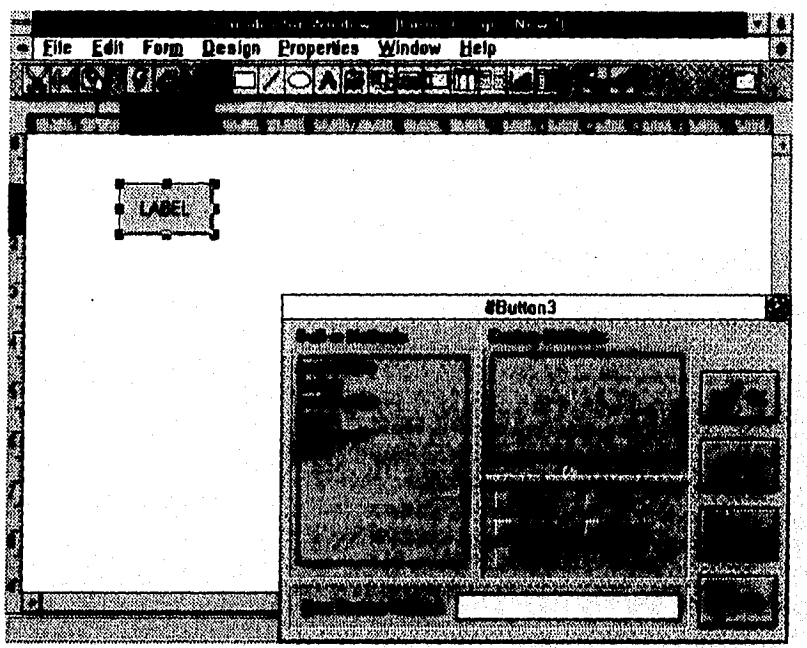
Típicamente, se unirán métodos de ObjectPAL a un botón, una página de una forma, o a la forma misma. Para iniciar el ligado de un método, inspeccionamos (click derecho) al objeto en la ventana de Diseño de la Forma y elegimos Métodos, o seleccionamos el objeto y presionamos Ctrl+Espacio. La Figura muestra la caja de diálogo **Métodos**, la cual aparece cuando se inspecciona un botón y se elige Métodos. En la Figura, hemos seleccionado el método *pushButton*. Este método se dispara cuando el usuario presiona (hace click) en el botón en el cual el método está ligado.

A continuación se explican las áreas más importantes de la caja de diálogo Métodos.

- La barra de título despliega el nombre del objeto al cual se le está colocando el método.
- El lado izquierdo de la caja de diálogo presenta una lista de métodos internos proporcionados por ObjectPAL. El nombre del método refleja el tipo de evento que lo dispara.
- El botón OK acepta sus selecciones y abre una o más ventanas del Editor de ObjectPAL.
- El botón Delete permite borrar algún método previamente ligado.
- El botón Cancel cancela las selecciones y nos retorna a la ventana de Diseño de Forma.

- El botón Help despliega información de ayuda acreca de la caja de diálogo Métodos y proporciona una entrada al sistema de ayuda en linea de ObjectPAL.

- El botón rueda nos permite registrar a la caja de diálogo Métodos en pantalla conforme se seleccionan objetos en la ventana de Diseño de Forma o Arbol de Objetos. Cada vez que se selecciona un objeto, la caja de diálogo de Métodos cambiará para reflejar los métodos disponibles a ese objeto.

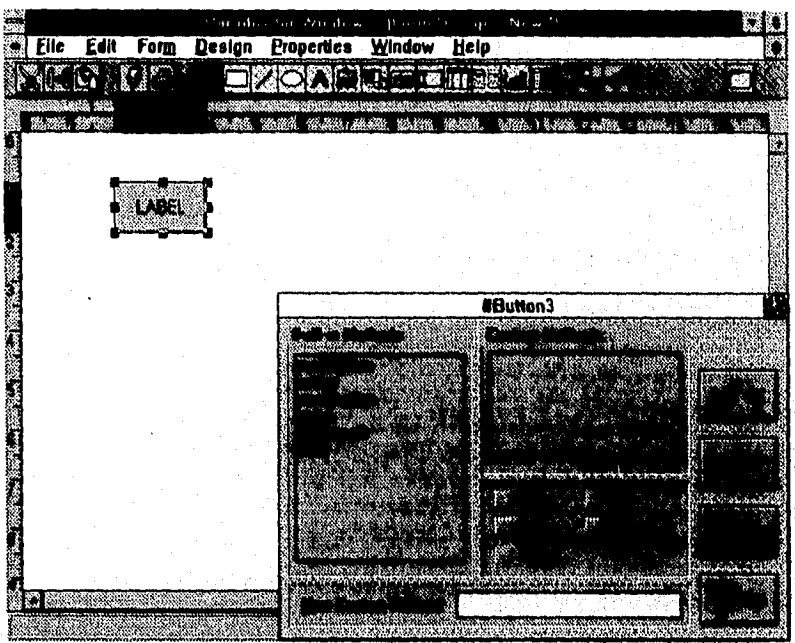


Los métodos internos desplegados en la caja de diálogos dependen del objeto inspeccionado y la configuración seleccionada en la caja de diálogo Desktop-Properties. La configuración por default es Desktop-Properties-Beginner, lo cual proporciona un "conjunto inicial" de métodos. Si se selecciona Desktop-Properties-Advanced, estarán disponibles mayores métodos. Cuando se aprende ObjectPAL, es mejor seguir la configuración default de Beginner para evitar confusión. La Tabla describe brevemente los métodos internos a "nivel principiante".

Para seleccionar un método de la lista de Métodos internos, haga click sobre él y escoja OK, o simplemente doble click al método. Para seleccionar varios métodos adyacentes, presione la tecla Shift mientras hace click sobre los nombres de los métodos, después seleccionar OK. Para seleccionar varios métodos no adyacentes, presionar Ctrl mientras que se hace click a los nombres de métodos, y seleccionar OK.

- El botón Help despliega información de ayuda acreca de la caja de diálogo Métodos y proporciona una entrada al sistema de ayuda en línea de ObjectPAL.

- El botón rueda nos permite registrar a la caja de diálogo Métodos en pantalla conforme se seleccionan objetos en la ventana de Diseño de Forma o Arbol de Objetos. Cada vez que se selecciona un objeto, la caja de diálogo de Métodos cambiará para reflejar los métodos disponibles a ese objeto.



Los métodos internos desplegados en la caja de diálogos dependen del objeto inspeccionado y la configuración seleccionada en la caja de diálogo Desktop-Properties. La configuración por default es Desktop-Properties-Beginner, lo cual proporciona un "conjunto inicial" de métodos. Si se selecciona Desktop-Properties-Advanced, estarán disponibles mayores métodos. Cuando se aprende ObjectPAL, es mejor seguir la configuración default de Beginner para evitar confusión. La Tabla describe brevemente los métodos internos a "nivel principiante".

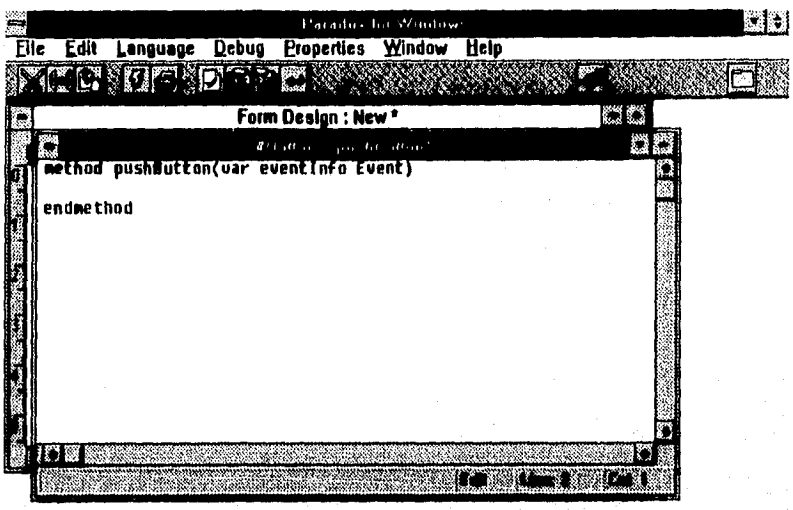
Para seleccionar un método de la lista de Métodos internos, haga click sobre él y escoja OK, o simplemente doble click al método. Para seleccionar varios métodos adyacentes, presione la tecla Shift mientras hace click sobre los nombres de los métodos, después seleccionar OK. Para seleccionar varios métodos no adyacentes, presionar Ctrl mientras que se hace click a los nombres de métodos, y seleccionar OK.

METODO	CUANDO SE LE LLAMA
action	Llamado cuando tiene lugar alguna acción (por ejemplo, el cursor se mueve a otro campo en la tabla) o cuando algun método desea efectuar alguna acción (por ejemplo, cambiar el modo de Edit)
arrive	Llamado después de moverse a (llegar a) un objeto tal como una forma, página, campo, tabla, u objeto multi-registro.
canDepart	Llamado cuando se intenta salir de un campo o registro.
changeValue	Llamado cuando está a punto de almacenarse un nuevo valor. (Disponible únicamente para campos).
error	Llamado cuando ocurre algun error. Los objetos pasan errores a sus contenedores (excepto a la forma misma).
menuAction	Llamado cuando el usuario elige un elemento de un menú o hace click sobre el SpeedBar que ejecuta alguna acción del menú.
mouseClick	Llamado cuando el boton izquierdo del mouse se presiona sobre el objeto.
pushButton	Llamado cuando un usuario hace click sobre un botón. (Disponible únicamente para botones y campos definidos como cajas de listas).

Para cada método seleccionado se abrirá una ventana de Editor de ObjectPAL. La Figura muestra la ventana del Editor de ObjectPAL que aparece después de seleccionar el método *pushButton* de la caja de diálogo Métodos. Paradox genera automáticamente las instrucciones ***method*** y ***endmethod*** para definir el nombre del método y sus parámetros y finalizar el método seleccionado.

La instrucción ***method*** especifica el nombre del método (por ejemplo, ***pushbutton***), y definir la información que está siendo pasada al método entre paréntesis. Cada método interno tiene la siguiente información pasada a él: la palabra *var* (la cual introduce un parámetro), el parámetro *eventInfo* (el cual proporciona información acerca del evento que disparó el método), y el tipo de dato del parámetro (por ejemplo, *Event*, *MoveEvent*). El tipo de dato dependerá del método seleccionado y determina cuáles métodos se pueden usar con *eventInfo* para obtener y/o almacenar información acerca del evento.

La barra de status indica la posición actual del punto de inserción (en el ejemplo se encuantra en la columna 1 línea 2). El código del programa debe ingresarse después de la instrucción ***method*** y antes de la instrucción ***endmethod*** (esto es, entre ambas instrucciones).



En la siguiente lista se detallan los pasos generales para escribir y probar un método.

1. Ingresar el código ObjectPAL para el método en la ventana del Editor ObjectPAL.
2. Checar la sintaxis del método y hacer las correcciones necesarias.
3. Guardar el método.
4. Probar el método.
5. Cambiar o borrar el método en caso de ser necesario.

Librerías ObjectPAL

Una librería es una colección de métodos y procedimientos personalizados. La slibrerías son útiles para almacenar y mantener rutinas empleadas con frecuencia, y para compartir métodos y variables personalizadas entre distintas formas.

Cuando se selecciona File | New | Library, Paradox abre la ventana de librerías. Cuando se inspecciona la ventana de Librería, se abre la caja de diálogo de Métodos. En muchos sentidos, trabajar con una librería es como trabajar con una forma. Por ejemplo, una

librería tiene métodos propios. Se puede agregar código a la librería tal como se hace con las formas, usando la caja de diálogo de Métodos y el Editor de ObjectPAL. Como con las formas, se pueden abrir ventanas del Editor para declarar métodos, procedimientos, variables, constantes, tipos de datos y rutinas externas personalizados.

Sin embargo, también existen algunas diferencias importantes:

- Al momento de la corrida, una libreria no se despliega en una ventana.
- Una librería no puede contener objetos de diseño; sólo puede contener código.
- En una Librería, las instrucciones que utilizan Self no se refieren a la Librería -en su lugar, se refieren al objeto que llamó al método.

Las reglas de alcance son diferentes a los de las librerías.

Uses

El bloque uses, declarado en la ventana Uses de algún objeto, hace que las rutinas almacenadas en librerías externas estén disponibles a los métodos de ObjectPal. Las rutinas deben caer en alguna de las siguientes descripciones:

- Rutinas escritas en ObjectPal y almacenadas en una librería ObjectPAL.
- Rutinas escritas en ObjectPAL y pegadas a una forma. La sintáxis para llamar a los métodos pegados a las formas es la misma que para llamar a una librería.

Las rutinas escritas en lenguaje ensamblador, C, C++ o Pascal y almacenadas en una librería ObjectPAL o Librería de Enlace Dinámico (DLL). Un DLL es una librería de código ejecutable o datos que se pueden enlazar a las aplicaciones al momento de la corrida. El uso de DLLs puede agregar cracterísticas y funciones sin modificar la aplicación compilada en ObjectPAL.

```
Uses libraryName
        routineName (parameter List) return type
endUses
```

libraryName: especifica el archivo DLL, Paradox asume una extensión .DLL o .EXE. Windows busca al archivo en el siguiente orden:

1. Directorio actual
2. Directorio de Windows
3. Directorio system de Windows

4. Directorios listados en el PATH

5. Lista de directorios mapeados en la red

Un bloque Uses puede contener una o más Nombres rutinas, y cada uno puede tener su lista de parámetros. Una lista de parámetros especifica uno o más argumentos y tipos de datos. Si la rutina retorna un valor, el Type return especifica el tipo de dato o valor retornado.

Se declara un bloque Uses en un objeto, y dentro de esa ventana se declara un Uses por cada librería o DLL que se quiera usar. Una vez declaradas, las rutinas estarán disponibles a todos los métodos relacionados con ese objeto, y a todos los objetos contenidos en éste.

En un bloque Uses, se declaran tipos de datos usando las siguientes palabras reservadas:

Tipo de Dato	ObjectPAL	C
Entero de 16 bits	CWORD	int
Entero de 32 bits	CLONG	long
Punto flotante de 64 bits	CDOUBLE	double
Punto flotante de 80 bits	CLONGDOUBLE	long double
Apuntador	CPTR	char far *
Datos gráficos o binarios	CHANDLE	THandle (Windows)

Estas instrucciones sólo son válidas dentro de un bloque Uses. No se usan en ninguna otra parte.

Para usar una rutina dentro de un método, se declaran variables a utilizar como argumentos, después se llama a la rutina. Por ejemplo,

```
; esto va dentro de la ventana Uses de algún objeto
uses myStuff ; lee rutinas de MYSTUFF.DLL
  doSomething(thisNum CLONG, thatNum CLONG) CDOUBLE ; declara una rutina
endUses

; esto modifica el método mouseUp de algún objeto
method mouseUp(var eventInfo MouseEvent)
var
  thisNum, thatNum LongInt ; declara variables a pasar a la rutina
  myResult Number
endVar

thisNum = 3,155,111
thatNum = 5,535,345
myResult = doSomething(thisNum, thatNum) ; llama a la rutina, retorna un resultado
```

En el ejemplo anterior, se declaran los argumentos dentro del bloque uses...endUses utilizando CLONG y CDOUBLE, y las variables del método fueron declaradas utilizando LongInt y Number. Esto se debe a que los tipos de datos de ObjectPAL son más complejos (y poderosos) que los correspondientes tipos en C y Pascal:

CWORD corresponde a SmallInt.
CLONG corresponde a LongInt.
CDOUBLE y CLONGDOUBLE corresponde a Number.
CPTR corresponde a String.
CHANDLE corresponde a Binario y Gráfico.

Desarrollo de Aplicaciones

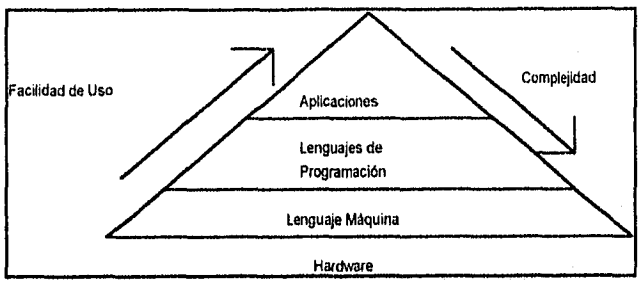
Vivimos en una era en la cual la información es el combustible de la economía. El éxito de un negocio puede radicar en cuanta información posea, qué tan rentable es, y qué tan rápido se puede accesar la información apropiada.

Desde que fue creada la primer computadora completamente electrónica (ENIAC) en 1946, han ayudado al manejo de información. Los compañías crearon departamentos de cómputo, a menudo llamados MIS (Servicios de Manejo de Información), para mantener las necesidades de hardware y el software de la compañía. La programación tradicional requería que un programador del MIS se reuniera con un usuario experto para que en conjunto diseñaran el software necesario para completar una tarea. Después de definir las necesidades de los usuarios, el equipo de programación pasaba varios meses creando el software para llevar a cabo esas tareas. Con mucha frecuencia durante esos meses de programación, cambiaban las necesidades de los usuarios, ocasionando retrasos en el desarrollo de la aplicación final. El mantenimiento de estos sistemas usualmente seguía el mismo patrón de comportamiento.

Hoy, avances en las herramientas de hardware y software permiten a los usuarios finales manejar información directamente. Con herramientas como Paradox for Windows y ObjectPal, usuarios expertos pueden instruir al hardware sobre cómo efetuar una tarea tal como duplicaciones en contabilidad o cálculo de impuestos, por lo tanto, el tiempo transcurre en la automatización de la tarea. El mantenimiento tiene el mismo beneficio en tiempo.

Qué es una aplicación?

Software es el término general utilizado para describir todos los niveles de instrucciones al hardware. Como se ilustra en la Figura, los niveles del software pueden visualizarse como una pirámide. En la base, lo más cercano al hardware, está el lenguaje máquina. A la mitad se encuentran lenguajes como ObjectPal que convierte las instrucciones a código máquina. En la punta, lo más cercano al usuario, están las aplicaciones. En la Figura se puede observar la Pirámide del software.



Una aplicación es un conjunto de herramientas de interface que conducen al usuario a través de los pasos necesarios para completar una tarea. La inetrafce del usuario es la interacción entre el usuario y la aplicación. Cuando se trabaja interactivamente con Paradox para crear una atbla, por ejemplo, Paradox es la aplicación. Sin embargo, se puede usar ObjectPal para crear aplicaciones propias que automaticen la tarea de crear una tabla definiendo nuestra propia interface del usuario. En lugar de que el usuario seleccione File|New|Table y después llene la caja de diálogo estandar para la creación de tablas, se le puede proporcionar al usuario un botón que despliegue nuestra forma o caja de diálogo.

Como mínimo, una aplicación en ObjectPal consta de una forma o un script.

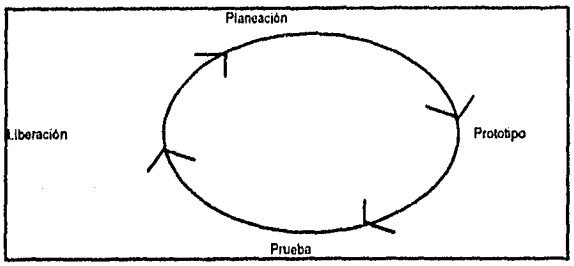
Una aplicación de base de datos en ObjectPal consta de los siguientes objetos que trabajan en conjunto como una sola unidad que ayuda a los usuarios efectiva y eficientemente en el manejo de información.

- Tablas
- Queries
- Formas
- Reportes
- Scripts
- Librerías

Estos objetos se comunican entre sí y se presentan al usuario como una sola ente por emdio de las instrucciones que se escriben en ObjectPal. Con estas herramientas, se crea un medio ambiente donde los usuarios pueden entrar, visualizar, mantener y reportar sus datos.

Ciclo de Desarrollo

La Figura ilustra el Ciclo de Desarrollo de Aplicaciones: planezción, prototipo, prueba y liberación. Cada fase puede componerse de varios pasos, dependiendo de la complejidad de la aplicación.



Planeación

Cuando se neceesita automatizar una tarea sencilla, tal como encontrar un registro basado en entradas del usuario, la solución es tan sencilla que probablemente no se pasará mucho tiempo planeando, diseñando o haciendo prototipos de la solución. Sin embargo, cuando se necesitan automatizar muchas tareas a través de distintas tablas, formas, y reportes, se debe de tomar un tiempo para planear. No puede explicarse lo importante que es la planeación par el éxito del proyecto. El tiempo invertido en la planeación de la aplicación ahorrará tiempo y frustraciones en el camino.

Al primer paso de la fase de planeación se le llama recopilación de información. Antes de empezar a crear tablas, formas, reportes o instrucciones en ObjectPal, se debe tener una clara idea de la información necesaria, qué tareas se van a automatizar, y qué tipo de interface al usuario deseamos crear. Consideremos las siguientes áreas:

- Tareas

  Definir qué tareas deseamos que nuestra aplicación realice.

  Listar los pasos requeridos para llevar a cabo cada tarea.
  Considerar las tareas a automatizar que son representativas, consumidoras de tiempo, o difíciles de realizar.

  Analizar la lista de tareas. Agrupar tareas similares. Buscar pasos similares entre las tareas.

- Gente

  Hablar con la gente que utilizará la aplicación.

  Hacer que los usuarios revisen las listas de tareas y los pasos y anoten cualquier discrepancia o concepto mal interpretado.

Determinar la experiencia de los usuarios en la aplicación y literatura del misnto. Los usuarios novatos pueden necesitar pantallas de ayuda que los orienten en el manejo del sistema.

Clarificar la jerga. Si los usuarios llaman a la tarea "localizar", la aplicación debe de hacerlo así. Nuestra meta es crear herramientas de automatización amigables y fáciles de usar que permitan hacer un trabajo de manera más eficiente.

- Datos

  Determinar qué datos manejará la plicación.

  Si aún no han sido creadas las tablas, determinar sus estructuras. Seguir las reglas de oro de las bases de datos: eliminar información redundante, usar varias tablas pequeñas en lugar de una sola grande, y elegir inteligentes tipos de campos.

  Considerar factores como la integridad de datos, acceso a los mismos y seguridad.

  Definir los medios de entrada de datos (electrónicos o por el usuario) y salida de datos (vistas, impresiones o electrónicos).

  Planear la aplicación identificando las tareas, personas, y datos involucrados nos ayuda a entender la aplicación antes de iniciar su desarrollo.

El resultado de la fase de recopilación es una sentencia de diseño, la cual incluye lo siguiente:

- Metas de la aplicación.
- Personal involucrado.
- Agenda Tiempo/Costo
- Aprobación
- Información de muestra empleada para la prueba
- Especificaciones de tareas y datos
- Especificaciones de documentación del sistema

Una sentencia de diseño proporciona un mapa para el desarrollo de aplicaciones. También asegura que todos los involucrados entiendan las especificaciones del diseño final, fechas y responsabilidades.

Prototipos

Un prototipo es un modelo funcional de la aplicación final. Un prototipo da a los usuarios y desarrolladores la oportunidad de ver los resultados del diseño antes de invertirle demasiado tiempo. En un lenguaje convencional, el proceso de crear un prototipo podría tomar meses; muy poco del prototipo se usa en la aplicación final. Sin embargo, en un lenguaje de desarrollo orientado a objetos tal como ObjectPal, los componentes del prototipo van directamente a la aplicación final. En tal lenguaje, se pueden hacer prototipos de cada objeto, uno a la vez, y visualizar los resultados del trabajo de inmediato.

Estructuras de las Tablas

El primer paso en la fase del prototipo es crear las tablas de la aplicación. Siempre se debe finalizar la estructura de la tabla antes de crear formas, reportes e instrucciones en ObjectPal. Si las estructuras de las tablas cambian posteriormente, se tiene que volver a revisar el trabajo. Se busca e identifica a través de la lista de tareas a aquéllas que puedan llevarse a cabo a través de Paradox interactivo.

Diseño de Formas

Con el diseño en mano y las tablas creadas, se está listo para diseñar la interface del usuario. Se toman ventajas de de las opciones de programación visual. En un lenguaje de eprogramación visual tal como ObjectPal , no se limita a menús en cascada como el único método de conducir al usuario a través de un proceso para completar una tarea. En cambio, se pueden proporcionar botones, campos, objetos multi-registros, y SpeedBars personalizadas. Uniendo instrucciones en ObjectPal a métodos manejados por eventos, se puede conducir a los usuarios a través de una tarea en el momento adecuado, en lugar de sólo darles una larga lista de opciones por donde desplazarse.

Se examina cada tarea y se decide qué objetos son más aconsejables para la tarea. Una opción de menú Editar|Agregar Vendedores, por ejemplo, podría ser mejor como un botón. La tarea de imprimir una orden de compra podría ser necesaria una caja de diálogo que pregunte a los usuarios si la impresora está lista y darles la oportunidad de cambiar la configuración en lugar de que aparezca como una opción en otro menú. Las herramientas de programación visual reducen el aglutinamiento de menús, lo cual facilita el uso del sistema.

Una vez determinados los objetos que componen la interface del usuario, se necesitan colocar esos objetos en la forma. No importa qué tan eficientes sean las instrucciones de ObjectPal, los usuarios lo notarán si la pantalla les da un dolor de cabeza. Se seleccionan fonts y colores fáciles de ller, especialmente en las formas de entradas de datos. Se debe dejar suficiente espacio entre los objetos. Si es posible se organiza todo por tareas. Se debe tener cuidado con las jerarquías si nos movemos entre campos.

Antes de automatizar tareas en ObjectPal se debe de probar la forma con datos. Este proceso eliminará problemas interactivos en la configuración de la tabla antes de codificar en ObjectPal.

Instrucciones en ObjectPal

Se inicia con las tareas básicas. Se selecciona una tarea que varias formas necesiten realizar: por ejemplo, insertar un nuevo registro. Crear un objeto (tal como un botón) con instrucciones en ObjectPal para llevar a cabo la función. A pesar de que puede tomar más tiempo el generalizar la rutina, se tendrá siempre ese objeto para todas las formas y aplicaciones, y a la larga nos ahorrará tiempo. Se prueba y depura el objeto generalizado en una de las formas, y después se copia a aotra. Se prueba y depura el objeto en esa forma para asegurarse de que es generalizado.

Una vez que se han automatizado las tareas generalizadas, se considera qué tipo de menú nos gustaría proporcionar a los usuarios. Usando ObjectPal, se pueden crear menús personalizados que se parezcan (y reemplacen) a los menús de Paradox. Se pueden crear los siguientes:

- Menús de barras horizontales.
- Menús verticales en cascada.
- Menús en cascada a partir de un menú de barras o un menú en cascada.

Las tareas restantes deben automatizarse en cada forma basados en un evento que el sistema o el usuario generen.

Después de determinar el método al cual se necesita ponerle instrucciones en ObjectPal, se deben considerar las reglas de alcance.

Armando Todo

Las librerías y scripts de ObjectPal nos permiten unir a las distintas formas, reportes, queries, y tablas que componen la aplicación. Se pueden colocar en la librería rutinas comúnmente usadas, tales como insertar un registro, y permitir que todas las formas accesen a ese método. También pueden colocarse menús personalizados en librerías para crear para crear un menú principal que conduzca al usuario a través de las distintas formas, reportes, queries y tablas.

Prueba y Depuración

La prueba y depuración deben ser procesos conjuntos. Conforme se diseñan prototipos, se deben probar y depurarlos. Cuando se agregan o modifican características, se debe de probar el objeto modificado y después probar el objeto en el contexto.

Si es posible, hacer lo necesario para que los usuarios prueben las formas, tablas, reportes y queries con datos reales (pero no sobre las tablas verdaderas). Además de probar las instruciones, este proceso retroalimentará a la interface del usuario y los pasos lógicos para completar la tarea.

ObjectPal permite automatizar con facilidad. Se pueden hacer prototipos, probar y depurar un objeto, algunos objetos, o un sistema completo. En ObjectPal, el ciclo de desarrollo no es estático. También, debido a su naturaleza modular, se puede responder rápido y fácilmente para cambiar el diseño.

Liberación

Una vez que la aplicación se prueba y depura, se está listo para liberarla a los usuarios. La fase de liberación consiste en la distribución de documentación, y mantenimiento de la aplicación. Antes de considerar la duplicación de discos, se deben considerar los siguientes puntos:

- Guardar la versión fuente o compilada
- Utilizar Paradox completo o el Runtime
- Puntos de red
- Puntos de documentación
- Puntos de Windows

Versión Fuente o Compilada

Se debe considerar si se desea que los usuarios sean capaces de modificar el diseño de formas o reportes y tener acceso a las instrucciones en ObjectPal. Una de las razones por las que las personas crean aplicaciones es la de proteger los datos de los usuarios novatos.

Nombres de Archivos y Extensiones

Objeto	Extensión Compilada	Extensión Desarrollada
Forma	FSL	FDL
Reporte	RSL	RDL
Librería	LSL	LDL
Script	SSL	SDL

Paradox Completo o el Runtime

Los usuarios pueden correr la aplicación en la versión completa de Paradox, llamada Paradox Workbench, o la versión Run-Time de Paradox for Windows.

19.4 Herramientas Multimedia (Sound Blaster 16)

En el presente Capítulo se describirán a grandes rasgos las herramientas Multimedia empleadas para el desarrollo de esta tésis. El componente principal fue el Kit Creative Multimedia de Sound Blaster (Tarjeta de Sonido, unidad de CD-ROM y software).

- Tarjeta Sound Blaster 16: es una tarjeta de alta calidad de 16 bits, que produce sonido y música de calidad CD a través de la computadora.

- Drive de CD-ROM: Doble velocidad, ritmo de transferencia de 300 KB/s mejora la visualización de Video for Windows y presentaciones multimedia Quicktime, compatible con multi-sesión foto CD; reproduce CDs musicales.

- Aplicaciones y utilerías para DOS y Windows.

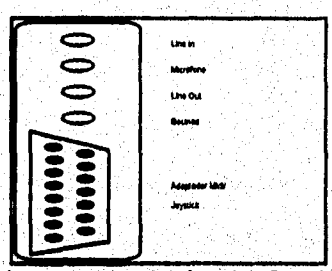
Requerimientos del Sistema

- IBM PC o compatible 386 o superior.
- 4 MB de RAM.
- 4 MB de espacio en disco duro.
- MS-DOS 3.1 o posterior.
- Tarjeta SVGA (640 X 480, 256 colores mínimo)
- Drive de 3.5 alta densidad.
- Mouse
- Windows 3.1 o posterior.
- Un slot de 16 bits de expansión disponible.
- Un abahía disponible para el CD-ROM.

19.4.1 Instalación del Kit Creative Multimedia

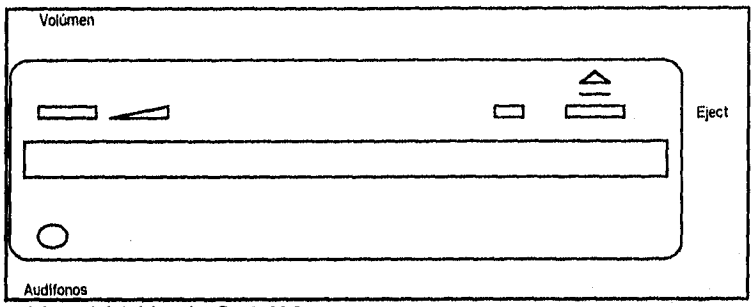
La instalación de este kit requiere hacer lo siguiente:

- Instalar la Tarjeta Sound Blaster 16 en la computadora.



Localización de los controles en un extremo de la tarjeta Sound Blaster 16.

- Instalar el drive del CD-ROM, el cual quedará denominado como el drive D:\>.



Panel frontal del drive del CD-ROM.

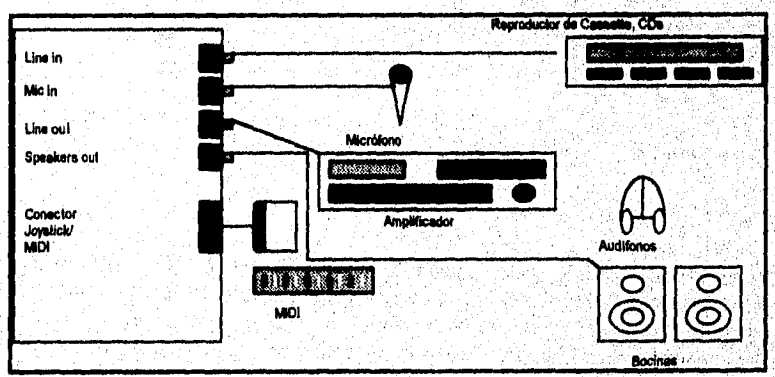
- Conectar bocinas o audífonos, así como otros dispositivos a la Tarjeta Sound Blaster 16.

- Correr el programa de instalación (instalar el software Creative en el disco duro e inicializar la Tarjeta y el drive del CD-ROM).

- Configurar los drivers y aplicaciones de Windows 3.1.

19.4.2 Conexión de Dispositivos a la Tarjeta Sound Blaster 16

Se pueden conectar una amplia variedad de dispositivos -tales como reproductores de cassette, micrófonos, bocinas, amplificadores MIDI (Interface Digital de Instrumentos Musicales) y joysticks, como se muestra enb la Figura.

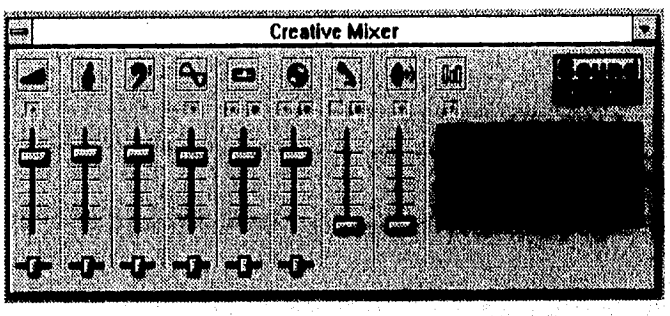


Conexión de Dispositivos a Sound Blaster 16.

Aplicaciones Windows

Las aplicaciones Sound Blaster producidas por Creative Labs trabajan con Sound Blaster automáticamente.

19.4.3 Creative Mixer



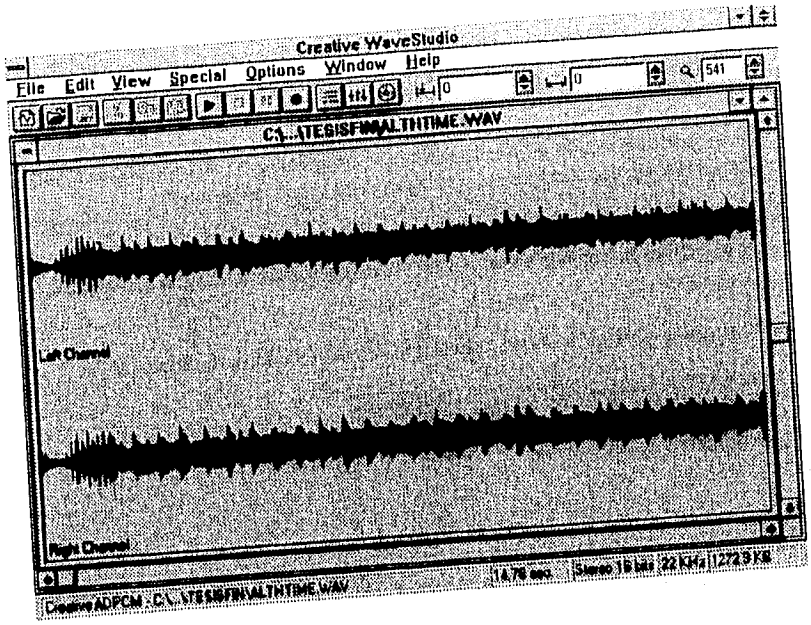
Creative Mixer es una mezcladora basada en Windows que permite combinar y manipular sonido de diferentes fuentes de audio. Con Creative Mixer es posible combinar el volúmen, tono y ganancia de una fuente de audio mientras se corren otras aplicaciones Windows. También se pueden seleccionar y mezclar diferentes fuentes de audio durante la reproducción y la grabación.

19.4.4 EnsembleCD

Se puede usar EnsembleCD para efectuar operaciones tales como la reproducción de un CD, recopilar una lista de selecciones, y asignar nombres a las pistas de los CDs.

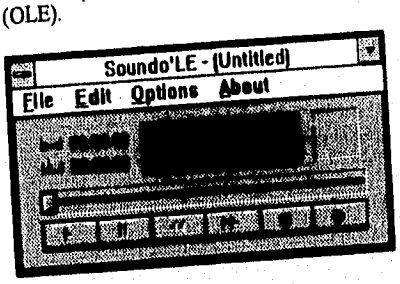
19.4.5 Creative WaveStudio

WaveStudio nos permite grabar, reproducir y editar *formas de ondas*, representación gráfica de un sonido grabado. Se puede cortar o pegar un segmento de la forma de onda y pegarla en una posición diferente de la misma, -o cortar y pegar un segmento de una forma de onda y pegarla en otra. Se puede invertir la onda, agregarle eco y desvanecerla, cambiar su volúmen, e inclusive agregar efectos de "rap".



19.4.6 Creative Soundo'LE

Creative Soundo'LE graba y reproduce archivos comprimidos y descomprimidos. Soundo'LE permite insertar archivos de sonido en otras aplicaciones mediante el enlace (OLE).



19.4.7 Monologue for Windows

Monologue for Windows nos permite agregar el habla a virtualmente cualquier aplicación Windows:

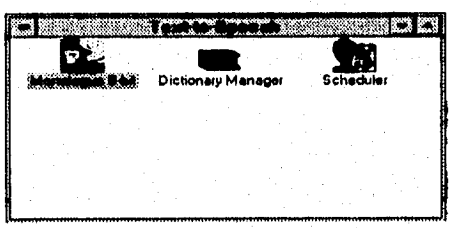
- Cualquier aplicación que pueda expresar datos como texto pueden tener a ese dato hablado.

- Los datos transferidos al clipboard pueden ser hablados, o pueden utilizarse las interfaces DDE y DDL para personalizar los datos hablados desde las aplicaciones Windows.

- La administración de recursos asegura que Monologue for Windows pueda trabajar aún con los programas más consumidores de memoria de Windows.

- Puede asistir a nuestros oídos proporcionando un grado extra de confianza en la entrada/recuperación de datos -los datos se leen exactamente como son ingresados o recuperados.

- Un manejo a través de software elimina la necesidad de hardware adicional.



19.4.8 Programación con Monologue for Windows

Monologue for Windows proporciona las interfaces DDE y DLL. Usando estas interfaces se puede agregar fácilmente el poder del habla a nuestras aplicaciones Windows existentes. Las aplicaciones deben proporcionar un mecanismo ya sea para enviar mensajes DDE o llamadas DLL.

Se debe seleccionar la herramienta que cumpla con los requerimientos y capacidades del medio ambiente de programación. Generalmente, la interface DLL proporciona la opción más rápida para enlazar al habla de Monologue for Windows.

Habla a Través de un DLL

La interface DLL es la preferida para agregar el habla a programas y a cualquier aplicación personal desarrollada bajo ambiente Windows que permitan hacer llamadas a librerías externas:

- La interface es mucho más directa y por lo tanto más rentable. La interface DLL es una llamada directa al Speech Engine de Monologue for Windows. La interface DLL es directa al Speech Engine; nonecesita iniciarse Monologue for Windows para accesar al Speech Engine, reduciendo así los requerimientos de memoria.

- Las macros o programas que utilizan la interface DLL tienen la capacidad de controlar los valores de volúmen y velocidad. Debido a que esto se logra sin la intervensión de Monologue for Windows, los cambios se llevan a cabo localmente e independientes de los parámetros de Monologue for Windows.

Cabe señalar que la opción utilizada para el desarrollo de esta aplicación fue la del "Habla a Través de un DLL" por las ventajas ya mencionadas.

19.5 Básicos del Sonido

El sonido se representa gráficamente mediante *ondas sinoidales*. Una *onda sinoidal* despliega la *frecuencia* y la *magnitud* del sonido. La *frecuencia* es el número de ciclos vibratorios por segundo y determina la velocidad o el timbre de un sonido; mientras más alto sea el número de ciclos, más alta será la velocidad. La *amplitud*, representada por la altura de la onda sinoidal, determina el volúmen o ruido de un sonido; mientras más alta sea la altura de las ondas sinoidales, más ruidoso será el sonido.

Una forma de onda WaveStudio representa a un sonido *análogo* como una onda sinoidal continua, una línea. Una onda sinoidal *digitalizada* se traslada a una serie de patrones de puntos que se asemejan a la forma y tamaño de una linea de ondas sinoidales. El *muestreo* es el proceso de convertir una onda de forma analógica a una onda de forma digital. La *frecuencia de muestreo* es el número de veces que el convertidor, tal como Sound Blaster 16, busca en el sonido para crear una imágen de forma de onda digitalizada. Por ejemplo, grabar sonido a 4 KHz crea 4000 imágenes digitalizadas o muestras de la onda de sonido durante un segundo.

WaveStudio permite grabar a frecuencias de muestreo desde 11 KHz hasta 44.1 KHz. La frecuencia de muestreo determina la calidad de la grabación. Mientras más alta sea la frecuencia de muestreo, mejor será el sonido. Por ejemplo, el audio de CD es de alta calidad y tiene una frecuencia de muestreo de 44 KHz a 16 bits.

19.5.1 Archivos Wave Comprimidos

Para trabajar efectivamente con archivos comprimidos, es necesario conocer cómo se abren y se editan tales archivos.

19.5.2 Apertura de Archivos Comprimidos

WaveStudio soporta actualmente cinco formatos de compresión: Microsoft ADPCM, CCITT A-Law, CCITT-μLaw, IMA/DVI ADPCM y Creative ADPCM.

Cuando se abre un archivo comprimido, WaveStudio lo descomprime antes de desplegarlo. El archivo debe residir en un drive con el suficiente espacio para alojar a la información descomprimida.

La cantidad de espacio requerido depende del formato de compresión adoptado. Para CCITT A-Law y CCITT µLaw, el radio de compresión es de 1:2 (comprimido:descomprimido). Para ADPCM, el radio es de 1:4. Por ejemplo, si se está trabajando con un archivo ADPCM de 1 MB, se necesitará contar con 4 MB de espacio libre en disco.

19.5.3 Edición de Archivos Comprimidos

Cuando se abre un archivo comprimido para edición, WaveStudio automáticamente comprime el archivo cuando se salva. Esto puede ser consumidor de tiempo para grandes archivos. Primero se guarda al archivo como PCM descomprimido. Una vez que se ha finalizado la edición, se guarda el archivo con el formato de compresión deseado.

19.6 Sistema CD Virtual

El motivo que me llevó al desarrollo de este sistema surgió a partir del día en que adquirí un CD de un grupo del cual sólo conocía su nombre, pero no los de sus canciones; por lo tanto ignoraba si el compacto que iba a comprar sería de mi total agrado, desgraciadamente ninguna de las canciones que aparecían en él me gustó, teniendo la idea de diseñar algún sistema que por medio de herramientas multimedia comerciales pudiera dar a los compradores información en general tal como nombres de artistas, de discos, de canciones, etc. Con frecuencia sucede que no sabemos el nombre de las canciones pero conocemos el nombre del grupo, o viceversa, teniendo que acudir al personal de la tienda para tratar de llegar al disco en el cual nos interesamos. Este sistema tiene como objetivo el que el usuario pueda hacer cuantas consultas desee a través de una PC con equipo multimedia de manera computarizada, obteniendo muestras que ayuden a darle una idea del CD que está por adquirir.

19.6.1 Estructuras de las Tablas Principales del Sistema CD Virtual

Cada registro de la estructura describe un campo del que se requiere información en la tabla. Dentro de cada registro de la tabla Struct de Paradox for Windows, los campos serán los siguientes:

- **Field Name:** Contiene el nombre del campo que está siendo descrito.
- **Type:** Contiene el tipo del campo: A (Alfanumérico), N (Numérico), $ (Monetario), D (Fecha), S (Short), M (Memo), F (Memo con Formato), B (Binario), G (Gráfico), u O (OLE).
- **Size:** Contiene el tamaño del campo.
- **Key:** Contiene un asterisco (*) para los campos llave; los campos no llave están en blanco.

- **_Invariant Field ID:** Contiene un identificador numérico para el campo. Este ID permanece sin cambio aún si se cambia el orden original de la tabla.
- **_Required Value:** Contiene un asterisco (*) si el campo es requerido; los campos no requeridos permanecen en blanco.
- **_Min Value:** Contiene el valor mínimo permisible para este campo, si es que fue definido.
- **_Max Value:** Contiene el valor máximo permisible para este campo, si es que fue definido.
- **_Default Value:** Contiene el valor default para este campo, si es que fue definido.
- **_Picture Value:** Contiene el formato para este campo, si es que fue definido.
- **_Table Lookup:** Contiene el nombre de la tabla de búsqueda para este campo, si fue definida.
- **_Table Lookup Type:** Contiene un número describiendo el tipo de tabla de búsqueda empleada. Si no se emplea tabla de búsqueda el número será un cero (0), y diferente de cero si es que se emplea alguna tabla de búsqueda.

Clave	A	1	"	1	"	1	9		#		0
Categoria	A	8		2	"						0

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
ID	A	12	"	1	"						0
Clave	A	1		2						GENEROS.DB	3
Titulo	A	30		3	"						0
Artista	A	20		4	"						0
Disquera	A	20		5	"						0
Ano_Grabacion	A	4		6	"	1970	1996		"4(#)		0
Precio	$			7	"	20,00	500,00				0
Existencia	S			8	"	1	20				0
Evaluacion	A	1		9					{M,R,B,E}		0

02/25/96

Estructura de la Tabla Songs

Field Name	Type	Width	Dec	Index	Lookup Table	Lookup Key	Lookup Field	Min Value	Max Value	Default Value	Table Lookup	Table Lookup Type
Cancion	A	30	*	1								
ID	A	12	*	2							DISCOS.DB	3
Numero	A	2		3				01	50		##	0
Duracion	A	5		4					59:59		##:##	0
Nombre_muestr	A	8		5								

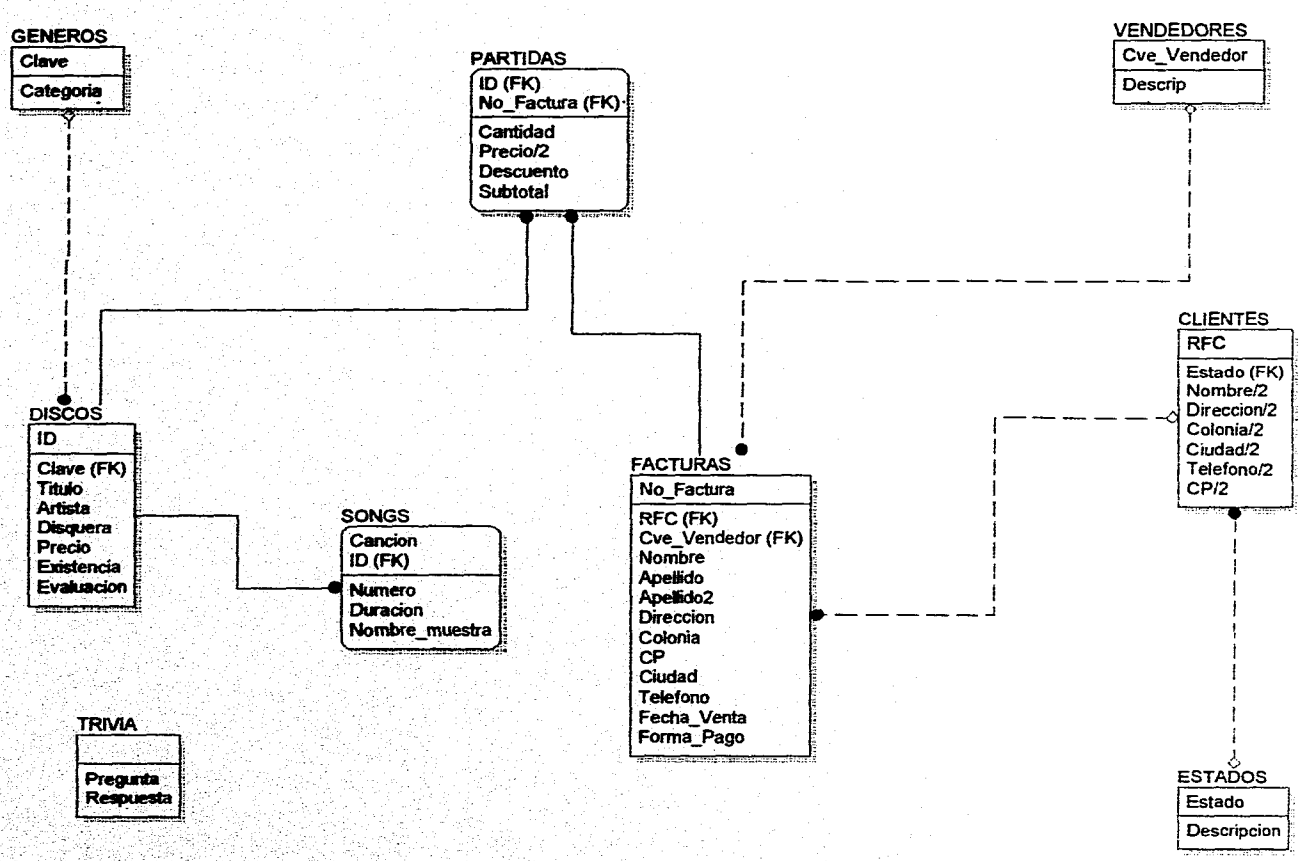
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
RFC	A	14	"	1					"4{&}"6{#		0
Nombre	A	15		2							
Apellido	A	15		3							
Apellido2	A	15		4							
Direccion	A	30		5							
Colonia	A	20		6							
CP	A	5		7					"5{#}		0
Ciudad	A	15		8							
Estado	A	5		9						ESTADOS.DB	3
Telefono	A	8		10					"3{#}-"4{		0

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
No_Factura	N		*	1							
RFC	A	14		2					"4{&}"6(#	CLIENTES.DB	4
Nombre	A	15		3							
Apellido	A	15		4							
Apellido2	A	15		5							
Direccion	A	30		6							
Colonia	A	20		7							
CP	A	5		8							
Ciudad	A	15		9							
Estado	A	5		10						ESTADOS.DB	3

Telefono	A	8		11							
Cve_Vendedor	A	3		12					*3(&)	VENDEDOR.DB	3
Fecha_Venta	D			13							
Forma_Pago	A	8		14							

No_Factura	N		*	1							
ID	A	12	*	2						DISCOS.DB	4
Cantidad	S			3							
Precio	$			4							
Descuento	N			5		0,00	40,00	0,00			0
Subtotal	$			6							

Diagrama ER del Sistema CD - Virtual



Virtual CD Land

No. Factura:	RFC:	Fecha_Venta:	Vendedor:	Forma de Pago:	Tarjeta
2	GUR9891003-S20	2/05/96	GGR		

Nombre:
Ma Mayela
Gutiérrez
Reséndiz

Domicilio:
Camino Prosperidad A #23

Colonia:
Campestre Aragón

C.P.:
07530

Ciudad:
México

Estado:
DF

Teléfono:
757-1617

	Precio	Descuento	Cantidad	Subtotal
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Subtotal 160.00
IVA 24.00
Total 184.00

CONCLUSIONES

CONCLUSIONES

La evolución de las herramientas de software a partir de la metodología orientada a objetos, ha cambiado de forma radical desde el desarrollo de un sistema de información automatizado, hasta la visión que tiene el usuario final del mismo sistema. Los lenguajes de programación y sus ambientes de desarrollo son cada vez más poderosos y sencillos de utilizar, permitiendo construir una aplicación en muy poco tiempo y a bajo costo, en comparasión a hace unos cuantos años, con interfaces de ventanas, iconos y programación visual.

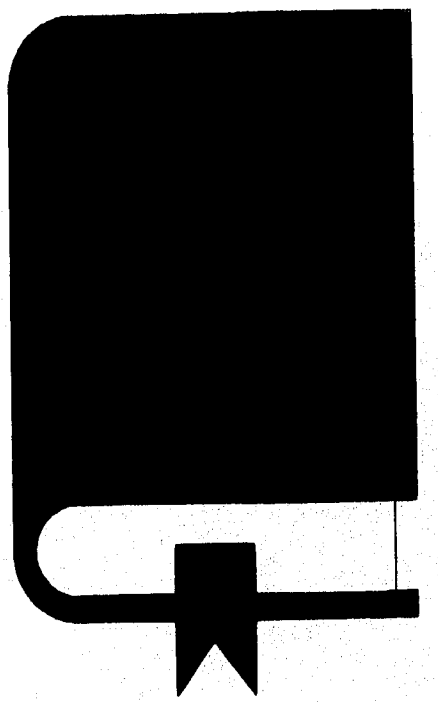
En esta evolución las bases de datos han jugado un papel predominante, han dado nuevas facilidades, producto de investigaciones que han partido de dos diferentes filosofías: bases de datos relacionales evolutivas y bases de datos orientadas a objetos con una visión revolucionaria.

Los OODBMS auanque no han tenido el auge tal que compitan con el número de instalaciones RDBMS, ofrecen un gran número de ventajas que facilitan el trabajo de modelado y organización de la información que en algunos casos se complica potencialmente al llevarse en una base de datos relacional.

Se requieren de herramientas de bases de datos que faciliten la representación de entidades complejas, heterogéneas, que cambian dinámicamente. Por otro lado, que ayuden y fomenten la reutilización del diseño y construcción de las aplicaciones. Los OODBMS son buenos candidatos a evaluar para lograr estos objetivos.

Sin embargo, existe una gran cantidad de aplicaciones que requieren de información heterogénea y sencilla, que se adecúa fácilmente a una representación de relaciones en tablas planas con valores atómicos. En esos casos los RDBMS siguen siendo la solución óptima.

Por otra parte, las herramientas multimedia disponibles hoy a gran escala proporcionan un valor agregado a todos los desarrollos de bases de datos que empleen esta poderosa opción, abriendo una gama de aplicaciones ilimitadas.



GLOSARIO

Glosario

Abstracción de Datos:	característica que permite la independencia de programas de datos y de operación.
ADIP (Direcciones en Prehuecos):	en un MD grabable las direcciones se formatean a intervalos fijos de tiempo, el cual ha sido estampado con prehuecos que formados con un patrón ligeramente variable creado por una señal de onda senoidal bajo velocidad lineal constante.
Administrador de Bases de Datos (DBA):	es el responsable de autorizar acceso a la base, para coordinar y monitorear su uso, y de adquirir recursos de hardware y software cuando sean necesarios. El DBA es responsable de problemas tales como cuarteaduras de seguridad o tiempos de respuesta pobres. En las grandes organizaciones, el DBA es asistido por un staff que le ayuda a realizar estas funciones.
Administrador de información almacenada:	controla el acceso del DBMS a la información almacenada, ya sea parte de la base de datos o del catálogo.
Analista de sistemas:	determina los requerimientos de los usuarios finales, especialmente de los usuarios ingenuos, y desarrolla especificaciones para transacciones enlatadas que cumplan estos requerimientos.
Area de trabajo del usuario (UWA):	es un conjunto de variables de programa, declaradas en el lenguaje huésped, para comunicar el contenido de registros individuales entre el DBMS y el programa huésped.
Area externa:	área externa a la del programa en donde no se registran señales musicales.
Area interna:	área en el disco antes del programa, iniciando a un diámetro interno de 0.29 mm, en la cual se guarda la tabla de contenidos (TOC).

Arquitectura Cliente-Servidor:	se usa junto con un DBMS si la aplicación corre físicamente en una máquina, llamada el cliente, y el almacenamiento y acceso se maneja por otra máquina llamada el Servidor.
ATRAC (Codificación Acústica de Transformación Adaptiva):	sistema de compresión digital de datos desarrollado para el MD en el cual pueden reproducirse señales de audio con cerca de 1/5 parte de los datos normalmente requeridos para la reproducción de alta fidelidad.
Atributo:	propiedades particulares que describen a una entidad. En el modelo ER ocurren diferentes tipos de atributos: sencillos contra compuestos; un sólo valor contra multivaluados; y almacenados contra derivados.
Atributo débil:	se le llama así a los tipos de entidad que no tienen ninguna llave dentro de sí mismos.
Atributo llave:	un argumento importante en las entidades de un tipo.
Atributo primo:	a un atributo del esquema de relación R se le llama atributo primo de R si es miembro de cualquier llave de R.
Base de datos:	es una colección de información relacionada.
Base de datos estadística:	se usa para proporcionar información estadística o resumida de valores basados en diferentes criterios. Por ejemplo, una base de datos de población puede proporcionar información acerca de grupos por edades, niveles de ingresos, niveles de educación, etc.
Bitácora:	registra todas las transacciones que afectan a los valores de elementos de la base de datos. Esta información puede ser necesaria para permitir la recuperación de fallas en la transacción. La bitácora se guarda en disco, de tal manera que no se afecta por ningún tipo de falla más que por una falla catastrófica. Además, la bitácora se respalda periódicamente en cinta para mantenerse en caso de tales fallas catastróficas.
Bloqueo:	variable asociada con un elemento de datos en la base y describe el status de ese elemento con respecto a las posibles operaciones que pueden aplicarse a él.

Buffer:	área reservada en almacenamiento principal que guarda un bloque de datos.
Calendario de transacciones:	cuando las transacciones se ejecutan concurrentemente de manera intercalada, el orden de ejecución de las operaciones de las distintas transacciones forma lo que se conoce como calendarios.
Capa Magneto-Optica:	capa delgada dentro de un MD en el cual pueden registrarse señales de "1" y "0" mediante la inversión del campo magnético.
Catálogo:	contiene información tal como la estructura de cada archivo, el tipo y formato de almacenamiento de cada elemento, y varios argumentos en la información.
CD-MO (CD Magneto-Optico):	versión magneto-óptica grabable del CD.
CD-ROM (CD de Sólo Lectura):	CD grabado con programas y datos para computadora.
CIRC (Código Entrelazado Reed-Solomon):	método de corrección de errores que combina el código Reed-Solomon, el cual es un código de corrección de errores con una alta capacidad de corrección de errores aleatorios, con un método entrelazado para convertir errores cíclicos en aleatorios.
Clase:	objetos con la misma estructura y comportamiento.
Cluster:	unidad mínima de grabación en un MD. 1 cluster = 36 sectores.
CLV (Velocidad Lineal Constante):	un lector debe moverse a lo largo de una pista en un disco a velocidad lineal constante (CLV) para leer la misma cantidad de datos en una unidad de tiempo. La relación CLV se mantiene variando la velocidad del disco de acuerdo a la localización del lector. El disco gira más lento en la parte externa del disco, y más rápido en la parte interna.
Codificador:	dispositivo utilizado para procesar (codificar) señales a un formato específico.

Compilador DDL:	procesa definiciones de esquemas, especificaciones en el DDL, y almacena la descripción de los esquemas (meta-data) en el catálogo del DBMS.
Conjunto multimiembro:	se usan en instancias donde el número de registros de un conjunto puede ser de más de un tipo de registro.
Conjunto propiedad del sistema:	es un conjunto sin tipo de registro propietario, en su lugar, el sistema es su propietario. Podemos imaginar este sistema como un tipo de registro propietario "virtual" con un sólo registro de ocurrencia.
Conjunto recursivo:	conjunto en el cual el mismo tipo de registro juega el papel de miembro y propietario.
Construcción de una base de datos:	proceso de almacenar los datos en algun medio controlado por el DBMS.
Constructor del esquema:	se le llama así a un diagrama desplegado gráficamente.
Argumento de integridad:	el tipo más sencillo de argumento de integridad involucra especificar un tipo de dato para cada elemento de información.
Argumento de participación:	especifica si la existencia de una entidad depende en si está relacionada a otra entidad a través del tipo de relación. Existen dos tipos de argumentos de participación: total y parcial.
Conversión A/D:	las señales se convierten en señales digitales dividiéndolas en intervalos constantes de tiempo y convirtiendolas a amplitudes de varios puntos. Estas se representan en código binario.
Conversión D/A:	conversión de señales digitales "1" y "0" en señales analógicas.
Datos:	hechos conocidos que pueden registrarse y tienen un significado implícito.
DBMSs centralizados:	significa que sus datos se almacenan en un solo sitio. Un DBMS centralizado puede soportar múltiples usuarios, pero los DBMSs y las bases residen en el mismo lugar.

DBMSs distribuidos:	pueden tener a la base y al software del DBMS distribuidos en varios lugares, conectados mediante una red de computadoras.
Decodificador:	dispositivo utilizado para restaurar señales procesadas (codificadas) a su formato original.
Definición de una base de datos:	involucra especificar los tipos de datos, estructuras y argumentos de los datos que serán almacenados.
Dependencia funcional:	denotada por X * Y, entre dos conjuntos de atributos X y Y que son subconjuntos de R especifica un argumento sobre las posibles tuplas que pueden formar una instancia de relación r de R. El argumento establece que, para cualquier par de tuplas t1 y t2 en r tal que t1[X] = t2[X], debemos tener que t1[Y] = t2[Y]. Esto significa que los valores del componente Y de una tupla en R dependen de, o están determinadas por los valores del componente X, o alternativamente, los valores del componente X de una tupla única (o funcionalmente) determinan los valores del componente Y.
Diagrama esquema:	se le llama esquema a la descripción de la base de datos.
Diccionario activo:	diccionario combinado catálogo/datos, el cual puede accesarse por los usuarios y el software del DBMS.
Diccionario pasivo:	diccionario de datos que puede accesarse por los usuarios y el DBA pero no por el software del DBMS.
Dominio:	especifica el conjunto de valores que puede asignarse a un atributo para cada entidad individual.
DRAM (Memoria de Acceso Aleatorio Dinámico):	dispositivo semiconductor el cual puede almacenar "1" y "0" en una serie de capacitores. El formato MD utiliza DRAM para lmacenar señales.
Efecto de enmascaraje:	fenómeno fisico acústico en el cual ciertos sonidos se vuelven inaudibles mediante sonidos de mas alto nivel a frecuencias adyacentes.

Efecto Kerr:	fenómeno en el cual la polarización plana de la luz láser reflejada de un material se desplaza en una de dos direcciones dependiendo su polarización magnética "más" o "menos".
EFM (Modulación Ocho a Catorce):	sistema de modulación empleado para convertir señales codificadas en 8 bits a señales de 14 bits para acoplar las señales digitales a las características de transmisión del disco.
Entidad:	representa a un objeto o concepto del mundo real que se almacena en la base de datos.
Espacio:	colección de páginas dinámicamente extendible.
Esquema:	se le llama esquema a la descripción de la base de datos.
Estado de la base:	se le llama así a los estados de la base en un momento en particular.
Estado válido:	estado que satisface la estructura y argumentos especificados en el esquema.
Estructura de la base de datos:	tipos de datos, relaciones y argumentos que debe retener la información. Los modelos de datos de alto nivel o conceptual proporcionan conceptos que están cerca de la forma en que muchos usuarios perciben los datos, mientras que los de bajo nivel o físicos proporcionan conceptos que describen los detalles de almacenamiento de los datos en la computadora. Los modelos representacionales ocultan algunos detalles de almacenamiento pero pueden implementarse de manera directa en un sistema de cómputo.
Etiqueta:	identificador único creado por el DBMS para una transacción.
Extensiones legales:	extensiones de relación r(R) que satisfacen los argumentos de dependencia funcional.

FIND ANY:	encuentra al primer registro de la base de datos del <nombre registro> especificado de tal forma que los valores de campo del registro correspondan con los valores inicializados anteriormente en los campos UWA correspondientes especificados en la cláusula del comando USING.
Grado de un tipo de relación:	es el número de tipos de entidades participantes. A un tipo de relación de grado dos se le llama binaria, a una de grado tres ternaria, etc.
Grado de almacenamiento:	colección de áreas de almacenamiento de acceso directo del mismo tipo de dispositivo. No existe correspondencia 1:1 entre una base de datos y un grupo de almacenamiento.
Grupo de repetición:	permite la inclusión de un conjunto de valores compuestos para un dato en un solo registro.
Herramientas:	son paquetes de software que facilitan el diseño y uso del sistema de base de datos, y ayudan a mejorar su desempeño.
Independencia de operación:	los programas de aplicación pueden operar sobre los datos invocando a estas operaciones a través de sus nombres y argumentos, sin importar cómo se implementan las operaciones.
Independencia de programas y datos:	propiedad del DBMS de almacenar separadamente la estructura de los archivos de datos en el catálogo de los programas de acceso.
Independencia física de datos:	capacidad de cambiar el esquema interno sin tener que cambiar los esquemas conceptuales.
Independencia lógica de datos:	capacidad de cambiar el esquema sin tener que cambiar los esquemas externos o programas de aplicación.
Indicador de actualidad:	mecanismo que permite al DBMS conocer el número de registros actuales y ocurrencias de conjuntos.

Indicador de status:	retornan una indicación de éxito o falla después de ejecutar cada comando DML.
Integridad de entidad:	establece que ningun valor de llave primaria puede ser nulo.
kA/m (Oersted):	KiloAmper/metro. Unidad de medición para expresar la fuerza magnética.
Lenguaje de Definición de Almacenamiento (SDL):	en DBMSs donde se mantiene una clara separación entre los niveles conceptuales e internos, el DDL se usa para especificar el esquema conceptual únicamente. Para especificar el esquema interno se usa el Lenguaje de Definición de Almacenamiento.
Lenguaje de Definición de Datos (DDL):	se usa para definir los esquemas físico y conceptual. El DBMS tendrá un compilador DDL cuya función será procesar instrucciones DDL para identificar y almacenar la descripción del esquema en el catálogo del DBMS.
Lenguaje de Definición de Vistas (VDL):	se usa para especificar vistas a los usuarios y sus mapeos al esquema conceptual.
Lenguaje de Manipulación de Datos (MDL):	una vez que se compilan los esquemas y se popula a la base, los usuarios deben tener algunos medios para manipular la base de datos. Las manipulaciones típicas incluyen recuperación, inserción, borrado y modificación de los datos. Para estos propósitos el DBMS proporciona un DML. Puede usarse un DML de alto nivel o no procedural para especificar operaciones complejas de manera concisa. Un DML de bajo nivel oprocedural debe estar contenido en un lenguaje de programación de propósito general. Este tipo de DML típicamente recupera registros individuales de la base y los procesa separadamente.
Llave parcial:	conjunto de atributos que pueden identificar de manera única a entidades débiles relacionadas a la misma entidad propietaria.
Macro ciclo de vida:	ciclo de vida de un sistema de información.

Manipulación de una base de datos:	la base de datos incluye funciones tales como queries para la recuperación de información especifica, la actualización de la base refleja los cambios en el mínimundo, y generar reportes de los datos.
Mapeo:	proceso de transformar requisiciones y resultados entre niveles.
Mbit:	Megabit. Unidad para expresar volúmen de datos digitales. Un millón de bits.
Meta-data:	información almacenada en el catálogo y que describe la estructura de la base de datos primaria.
Métodos:	procedimientos predefinidos que especifican las operaciones de cada clase.
Micro ciclo de vida:	ciclo de vida de un sistema de base de datos.
Modelo de datos:	conjunto de conceptos que pueden usarse para describir la estructura de una base de datos.
Modelo de red:	representa a los datos como tipos de registro y también representa a un tipo limitado de 1:N relaciones, llamado conjunto tipo.
Modelo jerárquico:	representa a la información como árboles de structuras jerárquicas. Cada jerarquía representa a un número de registros relacionados. No existe lenguaje estándar para el modelo jerárquico, a pesar de que la mayoría de ellos tienen lenguajes de registro a la vez.
Modelo orientado a objetos:	define a una base de datos en términos de objetos, sus propiedades y operaciones.
Modelo relacional:	representa a una base de datos como una colección de tablas, donde cada tabla puede almacenarse como un archivo separado.
Modulación por campo magnético:	modulación de alta velocidad en un campo magnético en una grabación magneto-óptica para representar una señal de entrada mediante orientación magnética.

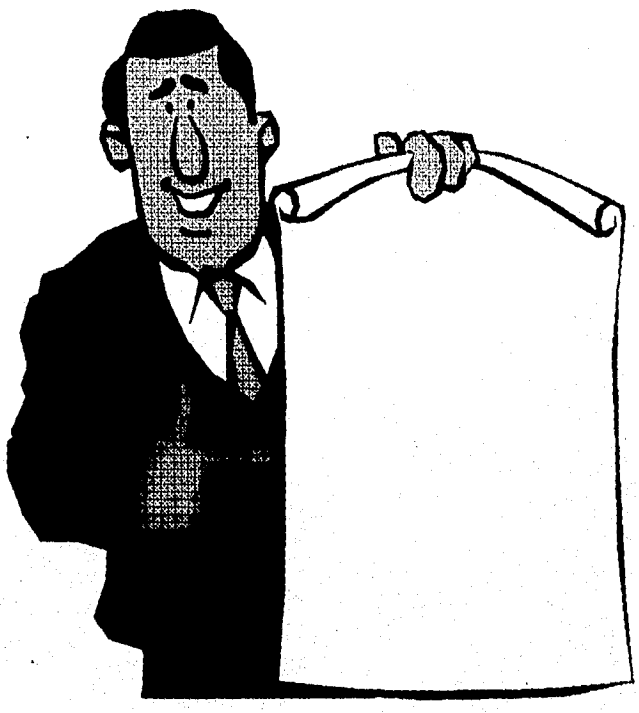
Multiprogramación:	permite que la computadora procese diferentes programas (o transacciones) al mismo tiempo, varios usuarios pueden usar sistemas de cómputo simultáneamente. Si existe un sólo CPU, puede procesar al menos un programa a la vez. Sin embargo, los sistemas operativos multiprogrmación ejecutan algunos comandos de un programa, luego los suspenden y ejecutan comandos del siguiente programa, y así sucesivamente. Un programa es continuado en el punto donde fue suspendido cuando le toca su turno de usar el CPU otra vez.
Nivel de manipulación:	forma en que las tuplas de una relación se almacenan y actualizan.
Nivel lógico:	forma en que los usuarios interpretan los esquemas de relación y el significado de sus atributos.
Normalización:	puede observarse como un proceso durante el cual se descomponen esquemas de relaciones no satisfactorios, rompiendo sus atributos en esquemas de relación más pequeños que poseen propiedades deseables. Un objetivo del proceso original de normalización es asegurar que no ocurran las anomalías de actualización.
Operación (función):	se especifica en dos partes. La interface (o signatura) de una operación incluye el nombre de la operación y los tipos de datos de sus argumentos (o parámetros). La implantación (o método) de la operación se especifica separadamente y puede cambiarse sin afectar la interface.
Organizaciones primarias de archivos:	determinan cómo están colocados físicamente en el disco los registros de un archivo. Un archivo pila (o archivo desordenado) coloca los registros en disco sin un orden en particular, mientras que un archivo sorteado (o secuencial) guarda los registros almacenados por el valor de un campo en particular. Un archivo seccionado utiliza una función de seccionamiento para determinar la colocación del registro en disco.
Página:	unidad básica de transferencia de datos entre almacenamiento secundario y primario.

Persistencia:	propiedad de un objeto de sobrevivir a la terminación de la ejecución del programa y después poder recuperarse directamente por otro programa.
Precompilador:	extrae comandos DML de una aplicación escrita en un lenguaje de programación huésped.
Primera Forma de Normalización (1NF):	establece que los dominios de los atributos deben incluir sólo valores atómicos (sencillos, indivisibles) y que el valor de cualquier atributo en una tupla debe ser un valor sencillo del dominio del atributo.
Programadores de aplicaciones:	implementan las especificaciones como programas; después prueban, depuran, documentan y mantienen las transacciones enlatadas. Tanto analistas como programadores debene star familiarizados con la capacidad total proporcionada por el DBMS para cumplir sus tareas.
PROJECT:	selecciona ciertas columnas de la tabla y descarta a las demás. La forma general de la operación PROJECT es: π<lista de atributos>(<nombre de la relación>) donde π (pi) es el símbolo utilizado para representar la operación PROJECT y <lista de atributos> es una lista de atributos de la relación especificada por < nombre de la relación>.
Radio de cardinalidad:	especifica el número de instancias de relación en la que una entidad puede participar.
Redundancia:	almacenar la misma información varias veces.
Registro almacenado:	cadena de bytes que contiene un prefijo con información de control del sistema y hasta n campos almacenados, donde n es el número de columnas en la tabla base. Los campos nulos al final de un registro de longitud variable no se almacenan.
Relación:	representa una interacción entre entidades; por ejemplo, una relación de trabajo entre un EMPLEADO y un PROYECTO.

Runtime de la base de datos:	maneja el acceso a la misma, al momento de la corrida; recibe operaciones de recuperación o actualización y las lleva a cabo en la base de datos.
Sector de enlace:	conversión para rearreglar señales digitales bajo cierto protocolo para mejorar la capacidad de corrección de errores.
Segunda Forma de Normalización (2NF):	un esquema de relación R está en (2NF) si cada tributo no primo A en R no es parcialmente dependiente de ninguna llave de R.
Seguridad y autorización:	es responsable de preservar la seguridad de porciones de una base de datos contra acceso no autorizado.
Semántica:	especifica cómo interpretar los valores de atributos almacenados en una tupla de la relación -en otras palabras, cómo se relacionan los valores de los atributos de una tupla con otra.
Sistema de base de datos:	combinación de base de datos y software.
Sistema de información:	en un ambiente computarizado, estos recursos incluyen los datos, el software del DBMS, el hardware y medios de almacenamiento, el personal que usa y maneja los datos (DBA, usuarios finales, usuarios paramétricos, y otros), el software de aplicación que accesa y actualiza los datos, y los programadores que desarrollaron las aplicaciones.
Sistema Manejador de Base de Datos (DBMS):	es una colección de programas que permite a los usuarios crear y mantener una base de datos.
Sistema diccionario de datos:	son mini-DBMSs que manejan meta-data para un sistema -esto es, los datos que describen la estructura, argumentos, aplicaciones, autorizaciones, etc. Estas se usan con frecuencia como una herramienta integral para el manejo de fuentes de información.
Sistema monousuario:	soportan a un sólo usuario a la vez y se usan casi siempre en PCs.

Sistema multiusuario:	incluyen a la mayoría de los DBMSs, soportan amuchos usuarios de manera concurrente.
Software de comunicaciones:	software cuya función es permitir a los usuarios en localidades remotas del sistema de base de datos acceso a la misma a través de terminales de cómputo, estaciones de trabajo, o sus mini o microcomputadoras locales.
Superllave:	es un conjunto de atributos S * R con la propiedad de que ningun par de tuplas t1 y t2 en ningun estado legal de la relación r(R) tendrán t1[S] = t2[S].
Temperatura Curie:	temperatura a la cual el magnetismo de un material específico se disipa. Esa temperatura varía de acuerdo al material.
Tercera Forma de Normalización (3NF):	un esquema de relación R está en 3NF, cada vez que una dependencia funcional X*A se retiene en R, ya sea que (a) X es superllave de R, o (b) A es un atributo primo de R.
Tipo de entidad:	define a un conjunto de entidades que tienen los mismos atributos.
Transacción:	ejecución de un programa que accesa o cambia los contenidos de la base de datos.
Transacción de actualización:	se usan para ingresar nuevos datos o modificar a los existentes en la base.
Transacciones de recuperación:	se usan para ingresar nuevos datos en pantalla o producir un reporte.
Transacciones mixtas:	se usan para aplicaciones más complejas que hacen alguna recuperación y algo de actualización.
Trayectoria de acceso:	es una estructura que eficienta la búsqueda de registros en una base de datos.
Tupla:	conjunto de pares (<atributo>,<valor>), donde cada par da el valor del mapeo de un atributo Ai para un valor Vi del dom(Ai).

Vector:	es un elemento de datos que puede tener múltiples valores en un sólo registro.
Vista:	puede ser un subconjunto de la base de datos o puede contener datos virtuales que se derivan de los archivos pero no se almacenan explícitamente.



APENDICES

APENDICE A

Sistemas de Bases de Datos
Una Breve Línea en el Tiempo

Evento		Consecuencia
Pre 60's		
1945	Desarrollo de cintas magnéticas (primer medio que permitía búsqueda).	Reemplazó a las tarjetas perforadas y papel.
1957	Primer computadora comercial instalada	
1959	McGee propuso la noción de acceso generalizado a información almacenada electrónicamente.	
1959	IBM introduce el sistema Ramac	Se hizo factible la lectura de información y el acceso a archivos de manera no secuencial.
Los 60's		
1961	Primer DBMS generalizado - Almacenamiento Integrado de Información de GE (IDS) diseñado por Bachman; amplia distribución en 1964. Bachman popularizó los diagramas de estructuras de datos.	Formuló las bases para el modelo de Red de Información desarrollado por la Conferencia en Lenguajes de Sistemas de Información, Grupo de Bases de Datos Task. (CODASYL, DBTG)
1965-1970	Desarrollo de sistemas de administración de información generalizados Sistemas de Administración de Información (IMS) desarrollados por IBM IMS DB/DC (base de datos/comunicaciones). System fue el primero a gran escala. SABRE, desarrollado por IBM y American Airlines	Proporcionó dos niveles de organización de la información conceptual/del usuario. Soportaba panoramas de la red en lo alto de las jerarquías Permitía acceso multiusuario a la información involucrando a una red de comunicaciones.
Los 70's	La tecnología de las bases de datos exprimentó un rápido crecimiento.	Sistemas comerciales seguidos de la propuesta CODASYL DBTG, pero ninguno completamente implementado. Sistema IDMS usado por BF Goodrich, Honeywells IDS II, UNIVAC DMS 1100, Burroughs DMS II, CDC DMS 170, Phillips PHOLAS y Digital DBMS II. Varios sistemas integrados DB/DC DBMS desarrollado como una disciplina académica y un área de investigación.
1970	Desarrollo del modelo relacional por Ted Codd, investigador de IBM.	Fundación de la teoría de las bases de datos.
1971	CODASYL Grupo de Reportes de Bases de Datos	

1975	Primer Conferencia Internacional de VLDB. Primera conferencia intrnacional del SIGMOS Grupo de Interés Especial en la Administración de Información Organizada.	Proporcionó otro fórum para la diseminación de la investigación en bases de datos. Proporcionó un fórum para la diseminación de investigación en las bases de datos.
1976	Modelo Entidad-Relación. (ER) introducido por Chen. Proyectos de investigación en los 70's: Sistema R (IBM), INGRES (Universidad de California, Berkeley), Sistema 2000 (Universidad de Texas, Austin), Proyecto Sócrates (Universidad de Grenoble, Francia), ADABAS (Universidad Técnica de Darmstadt, Alemania) Desarrollo de los lenguajes query en los 70's: SQUARE, SEQUEL (SQL), QBE, QUEL.	
Los 80's		
	DBMS desarrollados para computadoras personales (DBASE, PARADOX, etc)	Permitieron a los usuarios de PC definir y manipular datos.
1983	ANSI/SPARC revelaron haber implementado más de 100 sistemas relacionales a inicios de los 80's.	Emergieron los DBMS relacionales comerciales (DB2, ORACLE, SYBASE, INFORMIX, etc)
1985	Publicación preliminar del SQL estandar. El mundo de los negocios se ve influenciado por los "Lenguajes de Cuarta Generación". Propuesta de Definición de Lenguajes para Red (NDL) hecha por ANSI. Tendencias en los 80's: Sistemas Expertos de Bases de Datos, DBMs orientados a objetos, arquitectura cliente servidos para sistemas distribuidos.	Generación de completos programas de aplicación iniciando desde una interface de alto nivel de lenguaje no programable. Permitieron nuevas aplicaciones a as bases de datos, redes, y administración de datos distribuidos.
Los 90's	Demanda de capacidades extendidas a los DBMS para cumplir con nuevas aplicaciones	Características espaciales, temporales, y de información multimedia, incorporando capacidades activas y deductivas.
	Emergieron los DBMS orientados a objetos comerciales	
	Demanda para el desarrollo de aplicaciones usando información de una amplia variedad de recursos.	Emergieron estándares para queries e intercambios (SQL2, PIDES, STEP); extensión de las capacidades del DBMS a sistemas heterogéneos y multibases.
	Demanda para explotar los procesadores en paralelo de forma masiva (MPPS).	Mejora en el desempeño de los DBMS comerciales.

APENDICE B

Bases de Datos para PC

Los paquetes de bases de datos para PC han tenido gran éxito y las ventas de algunos de ellos sobrepasan a las de algunas BD de máquinas grandes, que por decirles de alguna manera les llamaremos BD profesionales. Pero este impacto comercial no es sorprendente. Tradicionalmente el cómputo de escritorio se vió como un cómputo no crítico, una falla por más garrafal que fuera afectaba a un solo usuario y localmente, pero ahora se ve que si puede ser crítico. Esto explica por qué los primeros productos de BD para escritorio carecían de seguridad y otras caracteristicas concernientes a las BD profesionales. Ahora esta situación está cambiando y vemos productos recientes como Access y Paradox, entre otros, que incorporan caracteristicas de las BD profesionales.

En un futuro cercano las BD de escritorio serán una réplica (quizá no exacta) de las BD profesionales; aplicando una especie de *downsizing* de sus funciones van a tener y hacer prácticamente lo mismo que las BD profesionales en la proporción debida.

Una característica importante de las BD para PC es que todas las que han tenido éxito en la industria cuantan con la característica de apegarse al modelo Relacional, en el que la información se encuentra en Tablas (o archivos, la mayoría de ellas) que representan objetos o entidades; sus columnas son los atributos de esos objetos y los renglones son las ocurrencias. Por su parte, los modelos Jerárquicos y de Red han tenido poco impacto en las BD para PC.

La idea con la que nacieron las BD para PC era permitir llevar aplicaciones que estaban contenidas integramente en un solo equipo. Sin embrago, con el advenimiento de las redes de microcomputadoras apareció la necesidad de incluir en estos paquetes la capacidad de compartir información entre varios usuarios de manera simultánea. Además, dad la aparición de la arquitectura cliente-servidor en el que no solo se comparten datos en la red sino que existe un servidor para los procesos de la BD, las casas de software se han visto en la necesidad de incluir en sus paquetes la necesidad de compartir y mezclar la información de la BD de la PC con la del (de los) servidor(es), es decir, con las BD profesionales.

Para completar el panorama de las BD en PC y sus tendencias, no puede omitirse el hecho de que en los últimos 3 o 4 años, virtualmente todos los paquetes que originalmente habían sido diseñados para su ejecución en el sistema operativo DOS se han ido convirtiendo para trabajar en Windows, con las implicaciones de manejos gráficos que conlleva.

Para el presente apéndice se tomaron en cuanta algunos de de los paquetes de BD para PC que en la actualidad se usan más: Access, FoxPro, Paradox dBASE. Además de los anteriores cabe destacar que además de no pretender ser un DBMS, Clipper, tiene un uso muy extendido en el desarrollo de aplicaciones que utilizan archivos dBASE.

dBASE

En sus diferentes versiones, dBASE es probablemente el DBMS que más éxito ha tenido quizá debido a que fue uno de los primeros en aparecer. Introdujo el estándar para los datos y gracias a su sencillez tuvieron éxito sus lenguajes de definición de datos y programación.

Entre las facilidades que ofrece, que no son del todo versátiles y rápidas, se encuentran la creación de consultas, formas, reportes y etiquetas. dBASE IV además incluye la opción de utilizar SQL para hacer uso de BD Relacionales.

dBASEIII no posee caracteristicas propias de control de seguridad e integridad de información, y prácticamente cualquier persona que tenga acceso a sus archivos puede leerlos e incluso modificarlos, situación que ha cambiado con dBASE IV. Otra limitación de estos productos es su velocidad de procesamiento, la cual es muy baja debido a que los programas se ejecutan por medio de un intérprete, es decir, no genera código binario sino que cada instrucción se ejecuta conforme se va encontrando. Debido a lo anterior, para ejecutar las aplicaciones en dBASE es necesario que el usuario tenga el paquete completo o, en el caso de dBASE IV, adquirir el compilador por separado para obtener código binario.

A pesar de que dBASE IV -sin el compilador- genera "código ejecutable" (que requiere de un *runtime*), por el lado de la velocidad no puede decirse que haya mucho avance, además de que ahora requiere muchísimos más recursos de procesador, memoria y disco que dBASE III. Por otro lado, dBASE IV compilado si proporciona un desempeño considerablemente mayor en equipos de 32 bits.

A la fecha hay una versión de dBASE IV (*Server Edition*, producto separado) que toma en cuenta la arquitectura cliente-servidor. Además existen versiones para DOS y UNIX de AT&T, SCO, AIX-RS/6000 y Sun.

Clipper

Nació como un compilador del lenguaje de dBASE III que permitía generar código binario para la PC, que se ejecutaba de manera muy rápida.

Básicamente Clipper permite manipular el lenguaje de programación de dBASE III con algunas pocas restricciones, sobre todo en cuanto a la consulta de datos. Sin embargo, las continuas extensiones al paquete le han agregado mucho más instrucciones de las que contiene dBASE y que lo hacen más versátil; le permiten interactuar con otros lenguajes (C, ensamblador, etc) y generar aplicaciones para red debido a su control de concurrencia. Además, no se requiere tener dBASE ya que con el propio Clipper se pueden generar tanto archivos como índices.

En la actualidad Clipper no tiene la posibilidad de conectarse a servidores de BD y tampoco existe una versión para Windows.

FoxPro

Otra opción que surgió para los desarrolladores en dBASE fue FoxBase que con el tiempo se llamó FoxPro y que además emigróa apltaformas Unix y Macintosh. Este ambiente se creó para manejar BD en forma muy semejante a como lo hace dBASE, pero con la característica de que permite una velocidad de ejecución de las aplicaciones mucho mayor gracias a que, además de tener un modo de operación como intérprete, permitía generar una versión ejecutable que se podía distribuir junto con un programa *runtime* (no obstante, comparado con Clipper, FoxPro es más elento en la ejecución). Asímismo, FoxPro extendió el lenguaje original de dBASE para darle mayor funcionalidad y manejo de multiusuarios, además de incluir un generador automático de pantalls y reportes.

La versión que apareció para Windows, además de permitir el diseño con la interfaz gráfica y el uso de SQL para hacer consultas, integra la capacidad de usar BD residentes en servidores tales como Oracle y Sybase. Esto último redunda en la posibilidad de crear aplicaciones cliente-servidor en las que se optimiza el acceso a los datos siempre que no se tengan localmente. Sin embrago, una consecuencia del cambio de plataforma de software (de DOS a Windows) es que FoxPro para Windows disminuyó su rendimiento en cuanto a velocidad de ejecución (para aplicaciones locales) e incrementó en buena medida sus requerimientos de espacio en disco y memoria.

Paradox

Es una vertiente a la de los manejadores xBase. Desde su inicio mostró un esquema de funcionamiento que daba una idea un poco más cercana a un DBMS profesional que dBASE: en lugar de archivos maneja tablas, contiene el concepto de vistas lógicas, realiza validación de datos, integridad referencial, maneja índices primarios y secundarios y seguridad de archivos. Además posee una tendencia a administrar en forma coherente todos los elementos de un proyecto.

Una consecuencia obvia de estas capacidades es una mayor dificultad para aprenderlo y operarlo que dBASE, y un mayor requerimiento de recursos, pero a la vez se obtienen tiempos de ejecución un poco menores en modelo local.

Las versiones actuales de Paradox están hechas para windows y DOS. Para Wondows incluye el modelado de datos en forma visual para formas y reportes, un ambiente de desarrollo orientado a objetos (lo cual a futuro simplifica el desarrollo), capacidad de utilizar simultáneamente datos de Paradox y dBASE y la conectividad a servidores de BD como Oracle, Sybase e Informix, mediante IDAPI -tecnología propiedad de Borland- y ODBC -estandar ampliamente aceptado. Además, sus lenguajes de programación PAL (para DOS) y Object PAL (para Windows) son muy poderosos.

Access

Es uno de los productos de más reciente aparición para manejo de BD en PC. En su diseño, a semejanza de Paradox, contiene muchos de los elementos de un DBMS formal: manejo de tablas, índices, llaves primarias y externas, integridad referencial, soporte de transacciones, seguridad de datos, y consultas mediante SQL.

Existen varias formas de utilizar Access: generación automática de declaraciones SQL mediante una herramienta gráfica de consultas, el uso de macros para controlar las pantallas y reportes generados automáticamente y para validar la información, y mediante el lenguaje de programación Access Basic. Cabe mendionar que este lenguaje tiene mucha semejanza con el aún más poderoso Visual Basic para Windows, y que este último a su vez tiene un *Engine* (parte básica del funcionamiento) de Access, por lo que es posible realizar programas en Visual Basic que exploten los datos de Access.

Dentro de Access es posible utilizar en forma transparente datos Btrieve, dBASE, Fox, además de acceso a datos en servidores de BD mediante el estandar ODBC (Open Database Conectivity).

De manera similar a Paradox, las demandas de espacio en disco, memoria y procesador son altas, y la ejecución de aplicaciones en forma local es muy lenta, por lo que no se le debe considerar como una opción para el desarrollo de aplicaciones en máquinas automáticas.

Mercados para Cada BD y Otras Consideraciones

Sin tratar de ser pretenciosos y sin considerar las estrategias de posicionamiento de cada uno de los productos, y con el objeto de que las tablas que continúan puedan entenderse mejor -comparativamente hablando- consideramos que:

Access de Microsoft, es un producto dual; sirve como *front-end* en cliente-servidor y como BD local para aplicaciones de cierto calibre. Está orientado a un usuario final no especialista en computación, un profesionista libre que quiere administrar su pequeño negocio con un sistema, o para departamentos pequeños que desean utilizarlo como *front-end* (conectividad).

Paradox de Borland, se parece más a un DBMS profesional, aplica tantoa profesionistas libres como a pequeñas y medianas empresas o departamentos. Como DBMS es un producto más robusto que puede utilizarse como *front-end*, como BD en redes locales y en modo local independientemente de servidores, tanto para desarrollo de aplicaciones de alto volúmen como de menor escala. Desde luego guardando las debidas proporciones con respecto a BD profesionales.

FoxPro de Microsoft, es parecido a Paradox sin la robustez de éste, aplica al desarrollo de aplicaciones de alto volúmen, para pequeñas y medianas empresas o departamentos. En

aspectos de conectividad es mejor Access. La versión de windows puede -aunque no se recomienda- usarse con cierta facilidad por profesionistas con pocos conocimientos de computación.

dBASE IV de Borland, aplica prácticamente a los mismos mercados que FoxPro.

Clipper de Computer Associates, difícilmente será elegido por profesionales libres que no tinen conocimientos de programación. Tradicionalmente lo utilizan pequeñas empresas y departamentos de medianas empresas o programadores independientes.

Tablas Comparativas

Los resultados mostrados en las siguientes tablas fueron obtenidos tomando en cuenta la documentación de los paquetes, información técnica adicional proporcionada por algunos proveedores, así como la experiencia en su uso. Además, se han utilizado las últimas versiones a la fecha.

Para entender los conceptos de las tablas se ofrecen algunas definiciones básicas:

- Integridad referencial: se refiere al vínculo entre llaves primarias y secundarias de dos o más tablas distintas.
- Manejo de Llaves: significa que permite definir y manipular llaves primarias y secundarias.
- Seguridad de Datos: es la capacidad de un DBMS para evitar tentativas de acceso no permitidas.
- Transacciones: significa que el DBMS permite garantizar operaciones de inicio (begin transaction), de finazalización (commit) y retroceso (roll back), para prpósitos de cómputo consistente y confiable.

Conclusiones

En la elección de un DBMS para PC deben tomarse en cuanta las necesidades de información que se desean satisfacer, el número de usuarios y el alcance del proyecto. A pesar de que en general el costo del hardware ha ido disminuyendo, es importante de terminar que tan necesario es, por ejemplo, adquirir un paquete que corra en Windows, el cual requerirá más recursos. Aunado al costo del hardware está la consideración de si se necesitará o no conectividad con otros equipos, pues como ya se dijo, en las aplicaciones cliente-servidor es más importante la velocidad de proceso del servidor de BD que la de su contraparte el cliente.

Mientras la BD sea local los desempeños son los que se indican en la Tabla II o muy aproximados. Cuando la misma BD se instala en el servidor de una red local (lo que no se

aconseja con algunos productos, por su diseño y mercado) el desempeño cambia porque, por citar un ejemplo, Paradox maneja tablas (más parecido a un DBMS profesional) y con una consulta SQL incrementa su desempeño ya que la consulta se ejecuta en el servidor.Otros DBMSs manejan archivos y una consulta al servidor hace que se traigan los archivos al cliente para ejecutarla.

En la qrquitectura cliente-servidor y contando en el servidor con DBMSs como Oracle, Informix o Sybase, entre otras, y en el cliente con alguna de las ya descritas, la situación cambia. Ahora el comportmiento de estas como *front-end* será diferente y unas tendrán ventajas sobre otras. También lo puede ser el sistema SQL y sus diferencias en los distintos productos.

Característica	dBASE	Clipper	FOX	Paradox Win	Paradox DOS	Access
Corre Sobre DOS	Si	Si	Si	No	Si	No
Corre Sobre Windows	Si	No	Si	Si	Si	Si
Ejecutable requiere Runtime	Si	No	Si	Si	Si	Si
Generación Formas/Reporte	Si	No	Si	Si	Si	Si
Integridad Referencial	No	No	No	Si	Si	Si
Validación de Datos	No	No	No	Si	Si	Si
Manejo de Llaves	No	No	No	Si	Si	Si
Seguridad de Datos	Si	No	No	Si	Si	Si
Soporte de Transacciones	Si	No	No	Si	No	Si
Cliente-Servidor	No	No	Si	Si	Si	Si
Windows 4 y NT van a soportar transacciones por sí mismos y apoyarán a los DBMSs que corran en ellos.						

Tabla I Comparativo de BD para PC

Producto	Interfaz Amistosa	Facilidad Desarrollo*	Facilidad Usuario F	Recursos Mínimos**	Seguridad	Desempeño Local	Tipos de Archivos	Puntuación Total
dBASE	3	4	3	3	3	1	1	18
Clipper	1	5	1	5	1	5	1	19
Fox DOS	3	4	3	3	1	4	2	20
Fox Win	5	4	3	1	1	3	4	21
Paradox W	5	2	2	1	4	3	4	21
Paradox D	3	4	3	3	4	3	1	21
Access	5	2	4	1	3	1	5	21
*Con conocimiento de algun lenguaje estructurado. **Se refiere a los requerimientos de memoria, disco, video, etc.								

Tabla II Calificación de BD para PC (1=malo, 5=mejor)

Producto	Calificación Promedio	Explicación de la Calificación
dBASE	5.14	No tan actualizado tecnológicamente y limitantes como DBMS.
Clipper	5.43	No tan actualizado tecnológicamente, no es un DBMS, pero es el más rápido y requiere un mínimo de recursos.
Fox DOS	5.71	Las mismas limitantes que dBASE aunque es mucho más rápido.
Fox Windows	6.00	Tecnología actual, rápido, aunque carece de seguridad y requiere de mayores recursos que la versión para DOS.
Paradox Windows	6.00	Tecnología actual, rápido, buen nivel de seguridad, pero es lento en forma local y requiere muchos recursos.
Paradox DOS	6.00	Tecnología actual, rápido, buen nivel de seguridad, pero sólo trabaja archivos en formato Paradox DOS.
Access	6.00	Tecnología actual, buen nivel de seguridad, pero es muy lento en forma local y requiere muchos recursos.

Tabla III Explicación de la calificación (calculada sobre 35 puntos máximos posibles).

Producto	Precio de Lista	Proveedor
dBASE	795	Borland
Clipper	795	Computer Associates
Fox DOS	495	Microsoft
Fox Windows	495	Microsoft
Paradox Windows	250	Borland
Paradox DOS	250	Borland
Access	495	Microsoft

Tabla IV Relación costo/beneficio.

Ambiente Windows		Ambiente DOS	
Producto	Calificación	Producto	Calificación
Paradox	Primer Lugar	Paradox	Primer Lugar
Fox Pro y Access	Segundo Lugar	FoxPro	Segundo Lugar
		Clipper	Tercer Lugar
		dBASE	Cuarto Lugar

Tabla V Relación costo/beneficio.

APENDICE C

Avances Tecnológicos en el Almacenamiento de Datos

El Lector de CD ROM

El claro e indiscutible impulsor de la fiebre multimedia es el CD-ROM, que ha traído, con sus bajos precios y fácil distribución, la posibilidad de manejar cientos de megabytes de información. Todo el mundo sabe que se trata de un dispositivo para almacenamiento óptico, pero no está tan extendido el conocimiento de su verdadero funcionamiento. A continuación se describe con detalle.

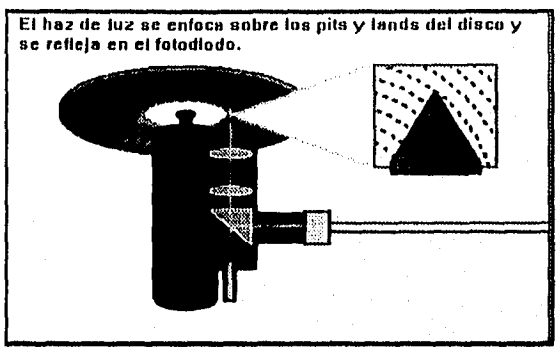
Descripción Física

En cada CD-ROM se pueden almacenar unos 680 MB en tan sólo 12 cm de diámetro y 1.2 mm de grosor. Aunque se emplean únicamente para lectura (ROM, Read Only Memory), día tras día se están extendiendo más las unidades capaces de escribir sobre discos especiales y de crear nuevos CDs. Un disco compacto se compone principalmente de un soporte de policarbonato transparente que le da el grosor característico, una capa reflectante que suele ser de aluminio en los CDs normales y de oro en los CDs grabables, una laca protectora y una etiqueta o zona para la impresión informativa de los contenidos del disco. La lectura se realiza por debajo, a través del policarbonato.

Unidad de CD-ROM

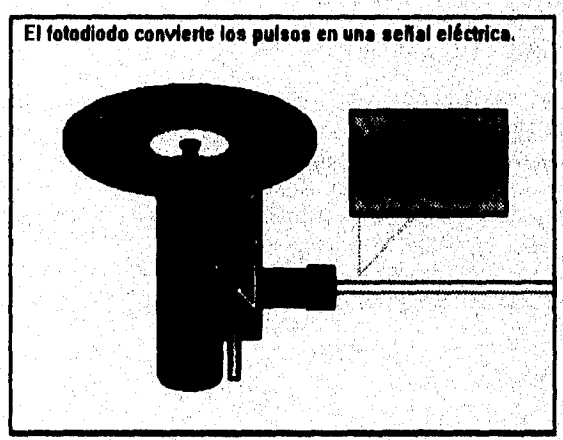
El lector de CDs está diseñado para actuar con una velocidad lineal constante. Pero los datos sobre el disco están escritos siguiendo una espiral desde el interior hacia el exterior, variando constantemente su diámetro en cada vuelta.

Debido a esto, la velocidad de rotación de la unidad lectora varía entre 3.5 rpm en la zona central próxima al agujero. La longitud total de la pista alcanza unos 4.5 Km y se halla dividida en sectores o bloques de longitud idéntica. La anchura de dicha pista es de unos 600 nanómetros y la distancia entre una vuelta y otra es de 1600 nanómetros. Así, se obtiene una densidad cercana a las 16,000 pistas por pulgada. En la pista, los datos se representan mediante pequeñas depresiones, llamadas pits, separadas entre sí por zonas planas o lands. La profundidad de los agujeros suele estar alrededor de los 120 nanómetros. El lector o cabeza lectora puede leer tanto los pits como los lands.



Lector de Datos

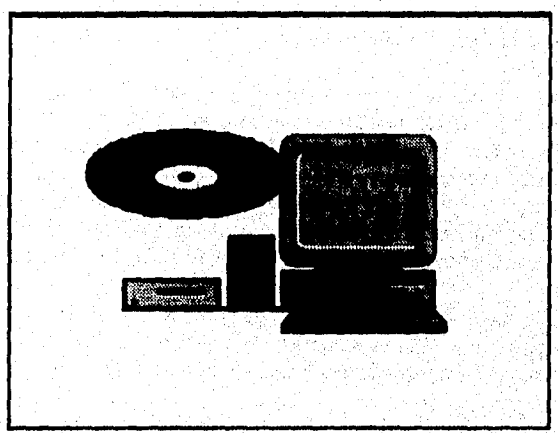
La cabeza lectora se compone de varios elementos. Pero principalmente consta de un emisor láser y un fotodetector que va registrando los cambios en el reflejo de la luz producido por los pits o lands en la superficie del disco: los pits dispersan la luz y los lands la reflejan. La luz refelejada sigue el mismo camino que la producida por el emisor láser; pero un espejo unidireccional situado en un punto desvía al fotodiodo (fotodetector) el haz de regreso por el disco, dejando pasar la que viene del láser. El fotodiodo convierte la luz en pulsos eléctricos, o lo que es lo mismo, en señal binaria de unos y ceros utilizable directamente por la computadora. Si los datos que se están leyendo son analógicos, tales como sonido o imágenes de video, se utiliza un ADC (Analogic Digital Converter). Con él se obtiene una señal apta para la circuitería convencional analógica, utilizable por cualquier amplificador o video.



La aparición del CD-ROM supuso toda una revolución en los medios de almacenamiento existentes. Sin embargo, le faltaba un pequeño detalle para alcanzar la perfección, el de ser regrabable. Y fue entonces cuando entró en escena el complemento ideal: la unidad grabadora de CD-ROM.

El método tradicional de "estampar" los CD-ROMs (perforando el disco virgen con un molde llamado master) no puede trasladarse al ámbito doméstico. Para ello se necesitaría un aparato que actuase de horno y de prensa a la vez, es decir, algo más parecido a un micoondas futurista que a un dispositivo informático. Y mejor no hablar de la posibilidad de crear discos masters.

Así, en un principio se pensó que debía hacerse uso del láser para realizar las pequeñas perforaciones (o pits), que representan la información binaria del CD-ROM. Pero esta idea fue pronto desechada por dos razones. 1) El diámetro del láser hubiese sido demasiado grande para las pistas del CD (de tan solo 0.6 micras de ancho). 2) Era imposible controlar el láser de tal forma que este profundizase en la superficie plástica del disco en la medida precisa, ni más ni menos. Por ello, se abandonó la pretención de grabar la información en la capa plástica (compuesta de policarbonato), para registrarla en otra más fácil de manipular, y que se situaría sobre esta. Así, pues, primero se empleó un barniz fotosensible sobre el cual incidía un rayo láser concentrado que lo quemaba y perforaba. Otro de los métodos se basa en emplear dos capas diferentes: una compuesta de polímero plástico, y otra metálica situada sobre la anterior. De esta forma, el láser calienta la capa metálica en el punto y hace que se evapore el polímero. Así, se forma una burbuja que, al intentar salir al exterior, deforma el metal que se encuentra sobre ella. Con esto se consigue desviar la reflexión del láser en ese punto, que es precisamente de lo que se trata.



Solución Definitiva hasta Ahora

De entre todos los métodos surgidos hay uno que destaca sobre los demás, y es el desarrollado por el japonés Taiyo Yuden. De hecho, este es el sistema empleado en la casi totalidad de las unidades grabadoras de CD-ROM actuales.

Los discos empleados en esta técnica, conocidos por los nombres de CD-R, CD-WO y CD-WORM, tienen en su superficie exterior, al igual que los corrientes, una cobertura plástica de policarbonato. Bajo esta se halla una capa orgánica de aspecto azulado llamada Dye. Cuando el láser incide sobre el Dye, se forman en él pequeñas burbujas que consiguen alterar las propiedades reflexivas del material. Así, cuando llegue la hora de leer, el láser lector (de mucha menor potencia que el grabador), ve desviando su reflejo por efecto de estas burbujitas.

La razón por la que los discos CD-R son dorados en lugar de plateados, como suele ser habitual, es que se necesita una superficie con mejores propiedades reflectantes de las que posee el aluminio tradicional. Por último, se cubre el disco con una capa de barniz protector y se le coloca la etiqueta, si la tuviera.

Proceso de Grabación

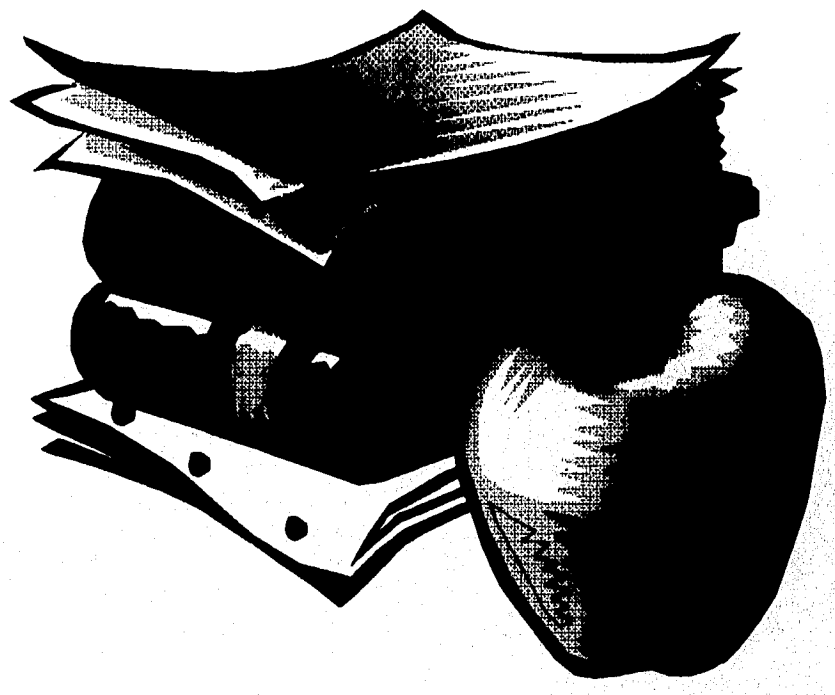
Una vez revelado el método de grabación, el resto del proceso no tiene ningun misterio. A la unidad le llega un flujo constante de datos desde un disco duro. Este contendrá la información que se quiere grabar tal y como será escrita (se le denomina "archivo imágen"). Es importante que no se produzca ningun problema en la transmisión constante de datos, ya que la más mínima incidencia, como un error de lectura o una vibración externa fuerte, puede dejar al CD-R virgen parcial o totalmente inservible.

Los datos llegan a la unidad y se van introduciendo en un buffer de memoria intermedio, que sirve de filtro en el caso de que lleguen más datos de los necesarios. Esto suele ser normal, ya que, en caso contrario, no puede llevarse a cabo la grabación. Por último, la información digital es convertida en pulsos eléctricos que activarán y desactivarán el láser escritor para alterar el Dye. De este modo se grabará el disco virgén, a medida que el láser vaya girando con velocidad lineal (no angular) constante. Dicha velocidad dependerá de cada unidad, repercutiendo directamente en el tiempo de grabación necesario.

Los CD-WORM suelen necesitar un formateado previo a la grabación, que consistirá en una hendidura guía en la capa de policarbonato. Asímismo, en unidades multisesión, es posible completar un disco parcialmente grabado en otra ocasión.

El formato de la caja donde se reunen todos los mecanismos puede variar: desde uno igual a una unidad de 5 1/4 a otro de dimensiones similares a las de un video doméstico. La explicación de que existan diferencias tan grandes entre los tamaños de las unidades radica en la gran cantidad de calor que se produce durante el proceso de grabación. Así, a mayor tamaño menor riesgo de dañar el disco.

La mayoría de las unidades permiten escuchar los CDs de audio, y algunas pueden comportarse incluso como un lector de CD-ROM convencional. Otras características adicionales son la capacidad de identificar los discos introducidos mediante códigos de colores (como el sistema IngoGuard de Kodak), o la posibilidad de conectarlos a dispositivos cargadores de forma que puedan realizarse varias copias de forma automática.



BIBLIOGRAFIA

BIBLIOGRAFIA

Ahmed, R y Navathe, S
"Database Fundamentals"
1994

Borland
Paradox for Windows
"Getting Started"
1992

Borland
Paradox for Windows
"Quick Reference"
1992

Cervantes de Poty, Ofelia
"Bases de Datos Orientadas a Objetos: Realidades y Promesas"
Soluciones Avanzadas Tecnologías de Información y Estrategias de Negocios, Noviembre 93
Publicación Mensual

Palomino Haddas, Carlos
"Evaluación: Bases de Datos para PC"
Soluciones Avanzadas Tecnologías de Información y Estrategias de Negocios, Mayo 94
Publicación Mensual

Simpson, Alan
"Mastering Paradox for Windows"
SYBEX, 1993

Stevens, Al
"C Database Development"
MIS: Press, 1992

BBS e INTERNET

Creative Labs BBS
(405)742-6660
Baudaje: 14400
Bits: 8
Paridad: Ninguna
Alto: 1

ftp.creaf.com
/pub/creative/files/patches