CAPÍTULO 6

Sistemas de gestión de bases de datos relacionales y SQL

CONTENIDO

Objetivos del capítulo

En este capítulo aprenderá lo siguiente:

- La historia de los sistemas de bases de datos relacionales y SQL
- Cómo se implementa la arquitectura de tres niveles en los sistemas de gestión de bases de datos relacionales
- Cómo crear y modificar una estructura de base de datos a nivel lógico usando SQL DDL
- Cómo recuperar y actualizar datos en una base de datos relacional usando SQL DML
- Cómo reforzar las restricciones en las bases de datos relacionales
- Cómo terminar transacciones relacionales
- Cómo se usa SQL en un ambiente de programación
- Cómo crear vistas relacionales
- Cuándo y cómo realizar operaciones sobre vistas relacionales
- Estructura y funciones de un catálogo de sistema de una base de datos relacional
- Las funciones de los varios componentes de un sistema de gestión de base de datos relacional

PROYECTO DE MUESTRA: Creación y manipulación de una base de datos relacional para la Galería de Arte
PROYECTOS ESTUDIANTILES: Creación y uso de una base de datos relacional para los proyectos estudiantiles

6.1 Breve historia de SQL en sistemas de bases de datos relacionales

Como se describe en el capítulo 4, el modelo relacional fue propuesto por primera vez por E. F. Codd en 1970. D. D. Chamberlin y otros en el Laboratorio de investigación San José de IBM desarrollaron un lenguaje ahora llamado SQL, o Structured Query Language (lenguaje de consulta estructurado) como un sublenguaje de datos para el modelo relacional. Originalmente nombrado SEQUEL, el lenguaje se presentó en una serie de ponencias que comenzaron en 1974, y se usó en un sistema relacional prototipo llamado System R, que desarrolló IBM a finales de la década de 1970. Otros primeros sistemas de gestión de bases de datos relacionales prototipos incluyeron INGRES, que se desarrolló en la Universidad de California en Berkeley, y el Peterlee Relational Test Vehicle, creado en el Laboratorio científico IBM del Reino Unido. El System R se evaluó y refinó durante un periodo de varios años y se convirtió en la base del primer sistema IBM de gestión de base de datos relacional disponible comercialmente, SQL/DS, que se anunció en 1981. Otro primer sistema de gestión de base de datos comercial, Oracle, se desarrolló a finales de la década de 1970 con el uso de SQL como su lenguaje. El DB2 de IBM, que también usa SQL como su lenguaje, se lanzó en 1983. Microsoft SQL Server, MySQL, Informix, Sybase, dBase, Paradox, r:Base, FoxPro y muchos otros sistemas de gestión de bases de datos relacionales han incorporado SQL.

Tanto el American National Standards Institute (ANSI) como la International Standards Organization (ISO) adoptaron SQL como un lenguaje estándar para bases de datos relacionales y publicaron especificaciones para el lenguaje SQL en 1986. Este estándar usualmente se llama SQL1. Una revisión menor, llamada SQL-89, se publicó tres años después. ANSI e ISO adoptaron una revisión mayor, SQL2, en 1992. Las primeras partes del estándar SQL3, que se conocen como SQL: 1999, se publicaron en 1999. Las grandes nuevas características incluyeron capacidades de gestión de datos orientados a objetos y tipos de datos definidos por el usuario. La mayoría de los proveedores de sistemas de gestión de bases de datos relacionales usan sus propias extensiones del lenguaje, lo que crea una variedad de dialectos alrededor del estándar.

SQL tiene un lenguaje de definición de datos (DDL) completo y lenguaje de manipulación de datos (DML) descritos en este capítulo, y un lenguaje de autorización, descrito en el capítulo 9. Los lectores deben notar que diferentes implementaciones de SQL varían ligeramente de la sintaxis estándar que se presentó aquí, pero las nociones básicas son las mismas.

6.2 Arquitectura de un sistema de gestión de bases de datos relacional

Los sistemas de gestión base de datos relacional soportan la arquitectura estándar en tres niveles descrita en la sección 2.6. Como se muestra en la figura 6.1, las bases de datos relacionales proporcionan independencia de datos tanto lógica como física, porque separan los niveles externo, lógico e interno. El nivel lógico para bases de datos relacionales consiste en tablas base que se almacenan físicamente. Estas tablas se crean mediante el administrador

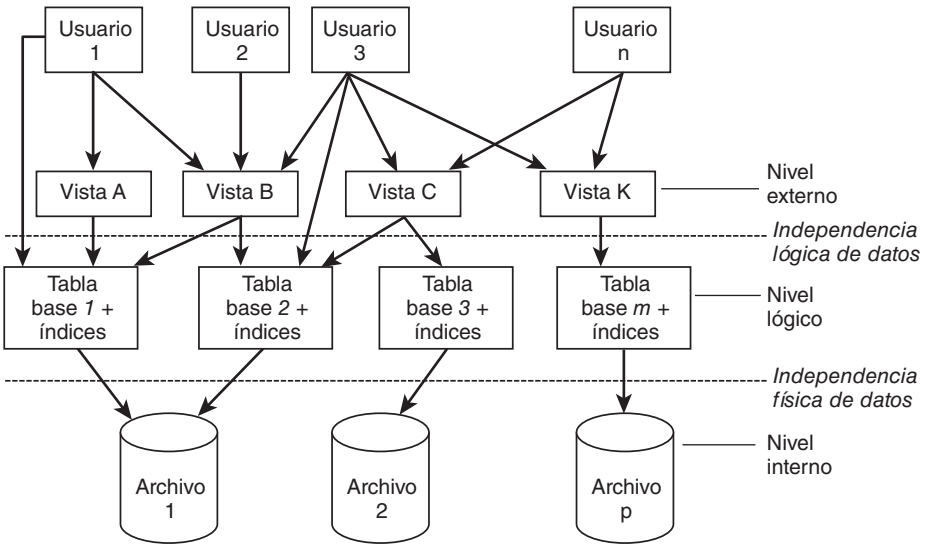


FIGURA 6.1
Arquitectura en tres niveles para bases de datos relacionales

de base de datos con el uso de un comando CREATE TABLE (crear tabla), como se describe en la sección 6.3. Una tabla base puede tener cualquier número de índices, creados por el ABD usando el comando CREATE INDEX (crear índice). Un índice se usa para acelerar la recuperación de registros con base en el valor en una o más columnas. Un índice menciona los valores que existen para la columna indexada y la ubicación de los registros que tienen dichos valores. La mayoría de los sistemas de gestión de bases de datos relacionales usan árboles B o árboles B+ para los índices (vea el apéndice A). En el nivel físico, las tablas base se representan, junto con sus índices, en archivos. La representación física de las tablas puede no corresponder exactamente con la noción de una tabla base como un objeto bidimensional que consiste en filas y columnas. Sin embargo, las filas de la tabla sí corresponden con los registros almacenados físicamente, aunque su orden y otros detalles de almacenamiento pueden ser diferentes del concepto de ellos. El sistema de gestión de bases de datos, no el sistema operativo, controla la estructura interna de archivos e índices. El usuario generalmente no está al tanto de cuáles índices existen, y no tiene control sobre cuáles índices usará para localizar un registro. Una vez creadas las tablas base, el ABD puede crear "vistas" para los usuarios, con el uso del comando CREATE VIEW (crear vista), descrita en la sección 6.8. Una vista puede ser un subconjunto de una sola tabla base, o bien crearse al combinar tablas base. Las vistas son "tablas virtuales", que no se almacenan de modo permanente, sino que se crean cuando el usuario necesita acceder a ellas. Los usuarios no están al tanto del hecho de que sus vistas no se almacenan de manera física en forma de tabla. Esto no es exactamente lo mismo que el término "vista externa", que significa la base de datos como aparece a un usuario particular. En esta terminología, una vista externa puede consistir en varias tablas base y/o vistas.

Una de las características más útiles de una base de datos relacional es que permite definición dinámica de base de datos. El ABD, y los usuarios a quienes autoriza, pueden crear nuevas tablas, agregar columnas a las anteriores, crear nuevos índices, definir vistas y eliminar cualquiera de estos objetos en cualquier momento. En contraste, muchos otros sistemas requieren que toda la estructura de la base de datos se defina en el momento de creación, y que todo el sistema se detenga y vuelva a cargar (reload) cuando se haga cualquier cambio estructural. La flexibilidad de las bases de datos relacionales alienta a los usuarios a experimentar con varias estructuras y permite la modificación del sistema para satisfacer sus necesidades cambiantes. Esto permite al ABD a asegurar que la base de datos sea un modelo útil de la empresa a lo largo de su ciclo de vida.

6.3 Definición de la base de datos: SQL DDL

Los comandos más importantes del lenguaje de definición de datos (DDL) SQL son los siguientes:

CREATE TABLE
CREATE INDEX
ALTER TABLE
RENAME TABLE
DROP TABLE
DROP INDEX

Estos enunciados se usan para crear, cambiar y destruir las estructuras lógicas que constituyen el modelo lógico. Estos comandos se pueden usar en cualquier momento para realizar cambios a la estructura de la base de datos. Existen comandos adicionales disponibles para especificar detalles físicos de almacenamiento, pero no se les discutirá aquí, pues son específicos al sistema.

Se aplicarán estos comandos al siguiente ejemplo, que se usó en capítulos anteriores:

```
Student (stuId, lastName, firstName, major, credits)
Faculty (facId, name, department, rank)
Class (classNumber, facId, schedule, room)
Enroll (classNumber, stuId, grade)
```

6.3.1 Create Table (crear tabla)

Este comando se usa para crear las tablas base que forman el corazón de una base de datos relacional. Dado que se puede usar en cualquier momento durante el ciclo de vida del sistema, el desarrollador de la base de datos puede comenzar con un pequeño número de tablas y agregarlas conforme se planeen y desarrollen aplicaciones adicionales. Una tabla base está bastante cercana a la noción abstracta de una tabla relacional. Consiste de uno o más **encabezados (headings) de columna**, que proporcionan el **nombre de columna** y **tipo de datos**, y cero o más **filas de datos**, que contienen un valor de datos del tipo de datos especificado para cada una de las columnas. Como en el modelo relacional abstracto, las filas se consideran sin orden. Sin embargo, las columnas están ordenadas de izquierda a derecha, para coincidir con el orden de las definiciones de columna en el comando CREATE TABLE. La forma del comando es:

```
CREATE TABLE nombre tabla base (nombre_col tipo_dato [restricciones columna -
NULL/NOT NULL, DEFAULT . . . , UNIQUE, CHECK . . . , PRIMARY KEY . . .]]
[,nombre_col tipo_dato [restricciones columna]
. . .
[restricciones tabla - PRIMARY KEY . . . , FOREIGN KEY . . . , UNIQUE
. . . , CHECK . . .]
[especificaciones almacenamiento]);
```

Aquí, *nombre tabla base* es un nombre proporcionado por el usuario para la tabla. No se pueden usar palabras clave SQL, y el nombre de la tabla debe ser único dentro de la base de datos. Para cada columna, el usuario debe especificar un nombre que sea único dentro de la tabla, y un tipo de datos. La sección opcional de especificaciones de almacenamiento del comando CREATE TABLE permite al ABD nombrar el espacio de la tabla donde se almacenará la tabla. Si el espacio de tabla no se especifica, el sistema de gestión de base de datos creará un espacio por defecto para la tabla. Quienes quieran pueden ignorar los detalles del sistema y quienes deseen más control pueden ser muy específicos acerca de las áreas de almacenamiento.

La figura 6.2 muestra los comandos para crear las tablas base para una base de datos para el ejemplo University.

6.3.1.1 Tipos de datos

Los tipos de datos disponibles incluyen varios tipos numéricos, cadenas de caracteres de longitud fija y de longitud variable, cadenas de bits y tipos definidos por el usuario. Los tipos de datos disponibles varían de DBMS a DBMS. Por ejemplo, los tipos más comunes en Oracle son CHAR(N), VARCHAR2(N), NUMBER(N,D), DATE y BLOB (gran objeto binario). En DB2, los tipos incluyen SMALLINT, INTEGER, BIGINT, DECIMAL/NUMERIC, REAL, DOUBLE, CHAR(N), VARCHAR(N), LONG VARCHAR, CLOB, GRAPHIC, DBCLOB, BLOB, DATE, TIME y TIMESTAMP. Microsoft SQL Server incluye NUMERIC, BINARY, CHAR, VARCHAR, DATE-TIME, MONEY, IMAGE y otros. Microsoft Access soporta varios tipos de NUMBER, así como TEXT, MEMO, DATE/TIME, CURRENCY, YES/NO y otros. Además, algunos sistemas como Oracle permiten a los usuarios crear nuevos dominios, y construir sobre los tipos de datos existentes. En lugar de usar uno de los tipos de datos disponibles, los usuarios pueden especificar dominios por adelantado y pueden incluir una condición de verificación para el dominio. SQL:1999 permite la creación de nuevos tipos de datos **distintos** usando uno de los tipos anteriormente definidos como el tipo fuente. Por ejemplo, se podría escribir:

```
CREATE DOMAIN creditValues INTEGER
   DEFAULT 0
   CHECK (VALUE >=0 AND VALUE <150);
```

Una vez creado el dominio, puede usarlo como un tipo de datos para atributos. Por ejemplo, cuando se crea la tabla Student, para la especificación de `credits` se podría escribir entonces:

```
credits creditValues, . . .
```

en lugar de

```
credits SMALLINT DEFAULT 0,
. . .
CONSTRAINT Student_credits_cc CHECK (credits>=0 AND credits < 150)
```

Sin embargo, cuando se crean tipos distintos, SQL:1999 no permite comparar sus valores con valores de otros atributos que tengan el mismo tipo fuente subyacente. Por ejemplo, si usa el dominio `creditValues` para `credits`, no se puede comparar `credits` con otro atributo cuyo tipo también sea SMALLINT, por ejemplo, con `age`, si tiene almacenado dicho atributo. No se pueden usar las funciones SQL disponibles como COUNT, AVERAGE, SUM, MAX o MIN en tipos distintos, aunque se pueden escribir las definiciones de las funciones para los nuevos tipos.

6.3.1.2 Restricciones (constraints) de columna y tabla

El sistema de gestión de base de datos tiene facilidades para reforzar la exactitud de los datos, las que debe usar el ABD cuando cree tablas. Recuerde de la sección 4.4 que el modelo relacional usa restricciones de integridad para proteger la exactitud de la base de datos y permitir la creación sólo de instancias legales. Estas restricciones protegen el sistema de errores en entrada de datos que crearían datos inconsistentes. Aunque el nombre de la tabla, los nombres de columna y los tipos de datos son las únicas partes requeridas en el comando CREATE TABLE, se pueden y deben agregar restricciones opcionales, tanto a nivel columna como a nivel tabla.

Las restricciones de columna incluyen opciones para especificar NULL/NOT NULL, UNIQUE, PRIMARY KEY, CHECK y DEFAULT para cualquier columna, inmediatamente des-

FIGURA 6.2
Enunciados DDL SQL para crear tablas para el ejemplo University

```
CREATE TABLE Student (
stuId CHAR(6),
lastName CHAR(20) NOT NULL,
firstName CHAR(20) NOT NULL,
major CHAR(10),
credits SMALLINT DEFAULT 0,
CONSTRAINT Student_stuId_pk PRIMARY KEY (stuId),
CONSTRAINT Student_credits_cc CHECK (CREDITS>=0 AND credits < 150);

CREATE TABLE Faculty (
facId CHAR(6),
name CHAR(20) NOT NULL,
department CHAR(20) NOT NULL,
rank CHAR(10),
CONSTRAINT Faculty_facId_pk PRIMARY KEY (facId));

CREATE TABLE Class (
classNumber CHAR(8),
facId CHAR(6) NOT NULL,
schedule CHAR(8),
room CHAR(6),
CONSTRAINT Class_classNumber_pk PRIMARY KEY (classNumber),
CONSTRAINT Class_facId_fk FOREIGN KEY (facId) REFERENCES Faculty (facId));

CREATE TABLE Enroll (
stuId CHAR(6),
classNumber CHAR(8),
grade CHAR(2),
CONSTRAINT Enroll_classNumber_stuId_pk PRIMARY KEY (classNumber, stuId),
CONSTRAINT Enroll_classNumber_fk FOREIGN KEY (classNumber) REFERENCES Class (classNumber),
CONSTRAINT Enroll_stuId_fk FOREIGN KEY (stuId) REFERENCES Student (stuId) ON DELETE CASCADE);
```

pués de la especificación del nombre de columna y el tipo de datos. Si no se especifica NOT NULL, el sistema permitirá que la columna tenga valores nulos, lo que significa que el usuario puede insertar registros que no tengan valores para dichos campos. Cuando un valor nulo aparezca en un campo de un registro, el sistema es capaz de distinguirlo de una cadena en blanco o valor cero, y tratarlos de manera diferente en cálculos y comparaciones lógicas. Es deseable poder insertar valores nulos en ciertas situaciones; por ejemplo, cuando un estudiante universitario todavía no haya declarado una especialidad es posible que quiera establecer un campo `major` a nulo. Sin embargo, el uso de valores nulos puede crear complicaciones, especialmente en operaciones como combinaciones, así que debe usar NOT NULL cuando sea adecuado. También puede especificar un valor por defecto para una columna, si quiere hacerlo así. Entonces, a todo registro que se inserte sin un valor para el

campo se le dará de manera automática el valor por defecto. Opcionalmente puede especificar que un campo dado tenga valores únicos al escribir la restricción UNIQUE. En este caso, el sistema rechazará la inserción de un nuevo registro que tenga en dicho campo el mismo valor que un registro ya presente en la base de datos. Si la clave primaria no es compuesta, también es posible especificar PRIMARY KEY como una restricción de columna, simplemente al agregar las palabras PRIMARY KEY después del tipo de datos para la columna. Claramente no se puede permitir valores duplicados para la clave primaria. También se deshabilitan los valores nulos, pues no se podría distinguir entre dos diferentes registros si ambos tuvieran valores nulos, así que la especificación de PRIMARY KEY en SQL lleva implícita una restricción NOT NULL, así como una restricción UNIQUE. Sin embargo, tal vez también quiera asegurar la exclusividad de las claves candidatas, y debe especificar UNIQUE para ellas cuando cree la tabla. El sistema automáticamente comprueba cada registro que intenta insertar para garantizar que los ítems de datos para columnas descritas como únicas no tengan valores que dupliquen cualquier otro ítem de datos en la base de datos para dichas columnas. Si es posible una duplicación rechazará la inserción. También es deseable especificar una restricción NOT NULL para claves candidatas, cuando es posible asegurar que los valores para dichas columnas siempre estarán disponibles. La restricción CHECK se puede usar a fin de verificar que los valores proporcionados para los atributos sean adecuados. Por ejemplo, podría escribir:

```
credits SMALLINT DEFAULT 0 CHECK ((credits>=0) AND (credits < 150)),
```

Las restricciones de tabla, que aparecen después de declarar todas las columnas, pueden incluir la especificación de una clave primaria, claves externas, exclusividad, comprobaciones y restricciones generales que se pueden expresar como condiciones a verificar. Si la clave primaria es compuesta, debe identificarla usando una restricción de tabla en lugar de una restricción de columna, aunque incluso una clave primaria que consista en una sola columna se puede identificar como una restricción de tabla. La restricción PRIMARY KEY fortalece la exclusividad y las restricciones de no nulo para la columna identificada como la clave primaria. La restricción FOREIGN KEY requiere la identificación de la tabla referenciada donde la columna o combinación de columnas sea una clave primaria. El estándar SQL permite especificar qué se hará con los registros que contengan los valores de clave externa cuando los registros que relaciona se actualicen o borren en su tabla origen (home). Para el ejemplo University, ¿qué debe ocurrir a un registro `Class` cuando el registro de los miembros del personal docente asignados a impartir la clase se borren o se actualice el `facId` del registro `Faculty`? Para el caso de borrado, el DBMS automáticamente podría:

- Borrar (delete) todos los registros `Class` para dicho miembro docente, una acción que se realiza cuando se especifica ON DELETE CASCADE (al borrar cascada) en la especificación de clave externa en SQL.
- Establecer el `facId` en el registro `Class` a un valor nulo, una acción que se realiza cuando se escribe ON DELETE SET NULL (al borrar establecer nulo) en SQL.
- Establecer el `facId` a algún valor por defecto como F999 en la tabla `Class`, una acción que se realiza cuando se escribe ON DELETE SET DEFAULT (al borrar establecer por defecto) en SQL (esta opción requiere que se use la restricción de columna DEFAULT para esta columna previo a la especificación de la clave externa).
- No permitir el borrado de un registro `Faculty` si existe un registro `Class` que se refiera a él, una acción que se realiza cuando especifica ON DELETE NO ACTION (al borrar no acción) en SQL.

Las mismas acciones, con significados similares, se pueden especificar en una cláusula ON UPDATE (al actualizar); esto es

```
ON UPDATE CASCADE/SET NULL/SET DEFAULT/NO ACTION
```

Tanto para borrado como para actualización, el defecto es NO ACTION, lo que en esencia no permite los cambios a un registro en una relación origen que causaría inconsistencia con los registros que se refieran a él. Como se muestra en la figura 6.2, para la tabla `Class` se eligió el defecto. Note también las opciones realizadas para la tabla `Enroll`, para los cambios hechos tanto a `classNumber` como a `stuId`.

El mecanismo de restricción de exclusividad de tabla se puede usar para especificar que los valores en una combinación de columnas deben ser únicos. Por ejemplo, para garantizar que ningunas dos clases tengan exactamente el mismo horario y salón, se escribiría:

```
CONSTRAINT Class_schedule_room_uk UNIQUE (schedule, room)
```

Recuerde de la sección 4.4 que la restricción de exclusividad permite especificar claves candidatas. La restricción anterior dice que `{schedule,room}` es una clave candidata para `Class`. También se podría especificar que `{facId,schedule}` es una clave candidata mediante

```
CONSTRAINT Class_facId_schedule_uk UNIQUE (facId, schedule)
```

pues un miembro del personal docente no puede impartir dos clases exactamente en el mismo horario.

A las restricciones, ya sean a nivel columna o tabla, en forma opcional se les puede dar un nombre, como se ilustra en los ejemplos. Si no se les nombra, el sistema generará un nombre de restricción único para cada restricción. La ventaja de las restricciones de nomenclatura es que luego se puede hacer referencia a ellas con facilidad. Existen comandos SQL para permitir la deshabilitación, habilitación, alteración o eliminación de restricciones a voluntad, siempre que se conozcan sus nombres. Es una buena práctica usar un patrón consistente en la nomenclatura de las restricciones. El patrón que se ilustra aquí es el nombre de tabla, nombre de columna y una abreviatura para el tipo de restricción (pk, fk, nn, uk, cc), separados por guiones inferiores.

6.3.2 Create Index (crear índice)

Opcionalmente puede crear índices para las tablas con el fin de facilitar la rápida recuperación de registros con valores específicos en una columna. Un índice sigue la pista de cuáles valores existen para la columna indexada y cuáles registros tienen dichos valores. Por ejemplo, si se tiene un índice en la columna `lastName` de la tabla `Student`, y se escribe una solicitud de consulta para todos los estudiantes con apellido Smith, el sistema no tendrá que explorar todos los registros `Student` para elegir los deseados. En lugar de ello, leerá el índice, que apuntará a los registros con el nombre deseado. Una tabla puede tener cualquier número de índices, que se almacenan como B-trees (árboles B) o B+ trees (árboles B+) en archivos índice separados, por lo general cerca de las tablas que indexan (vea el apéndice A para una descripción de los índices árbol). Los índices pueden crearse en campos sencillos o combinaciones de campos. Sin embargo, dado que el sistema debe actualizar los índices cada vez que se actualizan las tablas subyacentes, se requiere uso de sistema adicional. Además de elegir cuáles índices existirán, los usuarios no tienen control sobre el uso o mantenimiento de los índices. El sistema elige cuál, si hay alguno, índice usar en la búsqueda de registros. Los índices no son parte del estándar SQL, pero la mayoría de los DBMS soportan su creación. El comando para crear un índice es:

CREATE [UNIQUE] INDEX *nombre índice* ON *nombre tabla base* (*nombre_col* [*order*] [,*colname* [*orden*]] . . .) [CLUSTER] ;

Si se usa la especificación UNIQUE, el sistema reforzará la exclusividad del campo o combinación de campos indexados. Aunque los índices se pueden crear en cualquier momento,

puede tener problemas si intenta crear un índice único después de que la tabla tenga registros almacenados en ella, porque los valores almacenados para el campo o campos indexados pueden ya contener duplicados. En este caso, el sistema no permitirá la creación del índice único. Para crear el índice en `lastName` para la tabla `Student` se escribiría:

```
CREATE INDEX Student_lastName ON STUDENT (lastName);
```

El nombre del índice se debe elegir para indicar la tabla y el campo o campos usados en el índice. Cualquier número de columnas, sin importar dónde aparecen en la tabla, se puede usar en un índice. La primera columna nombrada determina el orden mayor, la segunda da el orden menor, etc. Para cada columna es posible especificar que el orden es ascendente, ASC, o descendente, DESC. Si elige no especificar el orden, ASC es el defecto. Si escribe

```
CREATE INDEX Faculty_department_name ON Faculty (department ASC, name ASC);
```

entonces se creará el archivo índice llamado `Faculty_Department_Name` para la tabla `Faculty`. Las entradas estarán en orden alfabético por departamento. Dentro de cada departamento, las entradas estarán en orden alfabético por nombre de docente.

Algunos DBMS permiten una especificación CLUSTER (grupo) opcional sólo para un índice para cada tabla. Si usa esta opción, el sistema almacenará registros con los mismos valores para el campo indexado que estén juntos físicamente, en la misma página o páginas adyacentes si es posible. Si crea un índice agrupado para el campo que se utiliza con más frecuencia para recuperación, puede mejorar sustancialmente el rendimiento de aquellas aplicaciones que necesiten dicho orden particular de recuperación, pues se minimizará el tiempo de búsqueda y el tiempo de lectura. Sin embargo, es el sistema, no el usuario, el que elige usar un índice particular, incluso uno agrupado, para la recuperación de datos.

Oracle crea automáticamente un índice en la clave primaria de cada tabla que se crea. El usuario debe crear índices adicionales en cualquier campo que se use con frecuencia en consultas, para acelerar la ejecución de dichas consultas. Los campos de clave externa, que se usan con frecuencia en combinaciones, son buenas candidatas para indexar.

6.3.3 ALTER TABLE, RENAME TABLE

Una vez creada una tabla, los usuarios pueden encontrarla más útil si contiene un ítem de datos adicional, no tiene una columna particular o tiene diferentes restricciones. Aquí, la naturaleza dinámica de una estructura de base de datos relacional hace posible cambiar las tablas base existentes. Por ejemplo, para agregar una nueva columna a la derecha de la tabla, use un comando de la forma:

ALTER TABLE *nombre tabla base* ADD *nombre_col tipo_dato*;

Note que no se puede usar la especificación NULL para la columna. Un comando ALTER TABLE ...ADD (alterar tabla ...agregar) hace que el nuevo campo se agregue a todos los registros ya almacenados en la tabla, y los valores nulos se asignen a dicho campo en todos los registros existentes. Los registros recientemente insertados, desde luego, tendrán el campo adicional, pero no se tiene permiso para especificar no nulos incluso para ellos.

Suponga que quiere agregar una nueva columna, `cTitle`, a la tabla `Class`. Esto se puede hacer al escribir

```
ALTER TABLE Class ADD cTitle CHAR(30);
```

El esquema de la tabla `Class` sería entonces:

```
Class(classNumber,facId,schedule,room,cTitle)
```

Todos los antiguos registros `Class` ahora tendrían valores nulos para `cTitle`, pero se podría proporcionar un título para cualquier nuevo registro `Class` que se inserte, y actua-

lizar los antiguos registros `Class` al agregarles títulos. También se pueden eliminar columnas de las tablas existentes mediante el comando:

```
ALTER TABLE nombre tabla base DROP COLUMN nombre columna;
```

Para eliminar la columna `cTitle` y regresar a la estructura original para la tabla `Class`, se escribiría:

```
ALTER TABLE Class DROP COLUMN cTitle;
```

Si quiere agregar, eliminar o cambiar una restricción, puede usar el mismo comando ALTER TABLE. Por ejemplo, si creó la tabla `Class` y despreció hacer `facId` una clave externa en `Class`, podría agregar la restricción en cualquier momento al escribir:

```
ALTER TABLE Class ADD CONSTRAINT Class_facId_fk FOREIGN KEY (facId) REFERENCES Faculty
(facId));
```

Podría eliminar una restricción nombrada existente al usar el comando ALTER TABLE. Por ejemplo, para eliminar la condición check (verificar) en el atributo `credits` de `Student` que se creó antes, podría escribir:

```
ALTER TABLE Student DROP CONSTRAINT Student_credits_cc;
```

Puede cambiar fácilmente el nombre de una tabla existente mediante el comando:

```
RENAME TABLE nombre tabla anterior TO nombre tabla nueva;
```

6.3.4 Enunciados DROP

Las tablas se pueden eliminar en cualquier momento mediante el comando SQL:

```
DROP TABLE nombre tabla base;
```

Cuando se ejecuta este enunciado se remueven la tabla en sí y todos los registros contenidos en ella. Además, todos los índices y, como verá más tarde, todas las vistas que dependen de ella se eliminan. Naturalmente, el ABD conferencia con los usuarios de la tabla antes de tomar tan drástico paso. Cualquier índice existente puede destruirse mediante el comando:

```
DROP INDEX nombre índice;
```

El efecto de este cambio puede o no verse en el rendimiento. Recuerde que los usuarios no pueden especificar cuándo el sistema usa un índice para recuperación de datos. Por tanto, es posible que exista un índice que realmente nunca se usó, y su destrucción no tendría efecto sobre el rendimiento. Sin embargo, la pérdida de un índice eficiente que el sistema use para muchas recuperaciones ciertamente afectaría el rendimiento. Cuando se elimina un índice, cualquier plan de acceso para aplicaciones que dependan de ella se marca como inválido. Cuando una aplicación las llama, se proyecta un nuevo plan de acceso para sustituir el anterior.

6.4 Manipulación de la base de datos: DML SQL

El lenguaje de consulta de SQL es **declarativo**, también llamado **no procedural**, lo que significa que permite especificar cuáles datos se recuperan sin dar los procedimientos para recuperarlos. Se puede usar como un lenguaje interactivo para consultas, incrustado en un lenguaje de programación huésped, o como un lenguaje completo en sí para cálculos con el uso de SQL/PSM (Persistent Stored Modules = módulos de almacenamiento persistentes).

Los enunciados DML SQL son:

SELECT
UPDATE
INSERT
DELETE

6.4.1 Introducción al enunciado SELECT

El enunciado SELECT se usa para recuperar datos. Es un comando poderoso, que realiza el equivalente de SELECT, PROJECT y JOIN del álgebra relacional, así como otras funciones, en un solo enunciado simple. La forma general de SELECT es

```
SELECT    [DISTINCT] nombre col [AS nombre nuevo], [,nombre col..] . . .
FROM      nombre tabla [alias] [,nombre tabla] . . .
[WHERE  predicado]
[GROUP BY nombre col [,nombre col] . . . [HAVING predicado]
```

o

```
[ORDER BY nombre col [,nombre col] . . .];
```

El resultado es una tabla que puede tener filas duplicadas. Dado que los duplicados se permiten en tal tabla, no es una relación en el sentido estricto, pero se le conoce como **multiconjunto** o **bolsa**. Como se indica por la ausencia de corchetes se requieren las cláusulas SELECT y FROM, pero no WHERE u otras cláusulas. Las muchas variaciones de este enunciado se ilustrarán mediante los ejemplos que siguen, con el uso de las tablas `Student`, `Faculty`, `Class` y/o `Enroll` como aparecen en la figura 6.3.

- Ejemplo 1. Recuperación simple con condición

  Pregunta: Obtener nombres, identificaciones y número de créditos de todos los que tienen especialidad Math.

  Solución: La información solicitada aparece en la tabla `Student`. Para dicha tabla se seleccionan sólo las filas que tienen un valor de 'Math' para `major`. Para dichas filas, sólo se despliegan las columnas `lastName`, `firstName`, `stuId` y `credits`. Note que se hace el equivalente SELECT (al encontrar las filas) y PROJECT (al desplegar sólo ciertas columnas) del álgebra relacional. También se reordenan las columnas.

  Consulta SQL:

```
SELECT      lastName, firstName, stuId, credits
FROM        Student
WHERE       major = 'Math';
```

  Resultado:

lastName	firstName	stuId	credits
Chin	Ann	S1002	36
McCarthy	Owen	S1013	0
Jones	Mary	S1015	42

Note que el resultado de la consulta es una tabla o un multiconjunto.

- Ejemplo 2. Uso de notación asterisco para "todas las columnas"

  Pregunta: Obtener toda la información acerca de Faculty CSC.

  Solución: Se quiere todo el registro `Faculty` de cualquier miembro del personal docente cuyo departamento sea 'CSC'. Dado que muchas recuperaciones SQL requieren todas las columnas de una sola tabla, existe una forma corta de expresar "todas

Student

stuId	lastName	firstName	major	credits
S1001	Smith	Tom	History	90
S1002	Chin	Ann	Math	36
S1005	Lee	Perry	History	3
S1010	Burns	Edward	Art	63
S1013	McCarthy	Owen	Math	0
S1015	Jones	Mary	Math	42
S1020	Rivera	Jane	CSC	15

Faculty

facId	name	department	rank
F101	Adams	Art	Professor
F105	Tanaka	CSC	Instructor
F110	Byrne	Math	Assistant
F115	Smith	History	Associate
F221	Smith	CSC	Professor

Class

classNumber	facId	schedule	room
ART103A	F101	MWF9	H221
CSC201A	F105	TuThF10	M110
CSC203A	F105	MThF12	M110
HST205A	F115	MWF11	H221
MTH101B	F110	MTuTh9	H225
MTH103C	F110	MWF11	H225

Enroll

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

FIGURA 6.3

La base de datos University (igual que la figura 1.1)

las columnas", a saber, con el uso de un asterisco en lugar de los nombres de columna en la línea SELECT.

Consulta SQL:

```
SELECT      *
FROM        Faculty
WHERE       department = 'CSC';
```

Resultado:

```
facId       name        department      rank
F105        Tanaka      CSC             Instructor
F221        Smith       CSC             Professor
```

Los usuarios que acceden a una base de datos relacional a través de un lenguaje huésped por lo general se les aconseja evitar el uso de la notación asterisco. El peligro es que se puede agregar una columna adicional a una tabla después de escribir un programa. Entonces el programa recuperará el valor de dicha nueva columna con cada registro y no tendrá una variable de programa que coincida con el valor, lo que causa una pérdida de correspon-

dencia entre variables de base de datos y variables de programa. Es más seguro escribir la consulta como:

```
SELECT     facId, name, department, rank
FROM       Faculty
WHERE      department = 'CSC';
```

- Ejemplo 3. Recuperación sin condición, uso de "distinto", uso de nombres calificados

  Pregunta: Obtener el número de curso de todos los cursos en los que están inscritos los estudiantes.

  Solución: Vaya a la tabla `Enroll` en lugar de a la tabla `Class`, porque es posible que exista un registro `Class` para una clase planeada en la que nadie esté inscrito. A partir de la tabla `Enroll` podría pedir una lista de todos los valores `classNumber`, del modo siguiente.

  Consulta SQL:

  ```
  SELECT   classNumber
  FROM     Enroll;
  ```

  Resultado:

  ```
  classNumber
  ART103A
  CSC201A
  CSC201A
  ART103A
  ART103A
  MTH101B
  HST205A
  MTH103C
  MTH103C
  ```

Dado que no necesita un predicado, no se usa la línea WHERE. Note que existen muchos duplicados en el resultado; es un multiconjunto, no una verdadera relación. A diferencia del PROJECT del álgebra relacional, el SELECT de SQL no elimina duplicados cuando se "proyecta" sobre las columnas. Para eliminar los duplicados, necesita usar la opción DISTINCT en la línea SELECT. Si escribe

```
SELECT DISTINCT classNumber
FROM Enroll;
```

El resultado sería:

```
classNumber
ART103A
CSC201A
HST205A
MTH101B
MTH103C
```

En cualquier recuperación, especialmente si existe una posibilidad de confusión debido a que el mismo nombre de columna aparece en dos tablas diferentes, especifique *nombretabla.nombrecol*. En este ejemplo se pudo haber escrito:

```
SELECT     DISTINCT Enroll.classNumber
FROM       Enroll;
```

Aquí, no es necesario usar el nombre calificado, pues la línea FROM dice al sistema que use la tabla `Enroll`, y los nombres de columna siempre son únicos dentro de una tabla. Sin embargo, nunca es equivocado usar un nombre calificado, y a veces es necesario hacerlo cuando dos o más tablas aparecen en la línea FROM.

- Ejemplo 4. Recuperación de una tabla completa

  Pregunta: Obtener toda la información acerca de todos los estudiantes.

  Solución: Puesto que se quieren todas las columnas de la tabla `Student`, use la notación asterisco. Puesto que se quieren todos los registros en la tabla, omita la línea WHERE.

  Consulta SQL:

```
SELECT     *
FROM       Student;
```

  Resultado: El resultado es toda la tabla `Student`.

- Ejemplo 5. Uso de "ORDER BY" y AS

  Pregunta: Obtener nombres e identificaciones de todos los miembros de `Faculty`, ordenados en orden alfabético por nombre. Solicite las columnas resultantes `FacultyName` y `FacultyNumber`.

  Solución: La opción ORDER BY (ordenar por) en el SELECT de SQL permite ordenar los registros recuperados en orden ascendente (ASC, por defecto) o descendente (DESC) en cualquier campo o combinación de campos, sin importar si dicho campo aparece en los resultados. Si se ordenan por más de un campo, el que se nombró primero determina orden mayor, el siguiente orden menor, etcétera.

  Consulta SQL:

```
SELECT     name AS FacultyName, facId AS
           FacultyNumber
FROM       Faculty
ORDER BY   name;
```

  Resultado:

FacultyName	FacultyNumber
Adams	F101
Byrne	F110
Smith	F202
Smith	F221
Tanaka	F105

Los encabezados de columna se cambian a los especificados en la cláusula AS. Se puede renombrar cualquier columna o columnas para desplegarla de esta forma. Note el nombre duplicado de 'Smith'. Dado que no se especifica orden menor, el sistema ordenará estas dos filas en cualquier orden que elija. Podría romper el "lazo" para dar un orden menor, como sigue:

```
SELECT   name AS FacultyName, facId AS
         FacultyNumber
FROM     Faculty
ORDER BY name, department;
```

Ahora los registros Smith se invertirán, pues F221 está asignado a CSC, que alfabéticamente está antes que History. Note también que el campo que determina el orden no necesita ser uno de los desplegados.

- Ejemplo 6. Uso de condiciones múltiples

  Pregunta: Obtener nombres de todos quienes tienen especialidades Math que tengan más de 30 créditos.

  Solución: De la tabla `Student`, elija aquellas filas donde la especialidad sea 'Math' y el número de créditos sea mayor que 30. Estas dos conexiones se expresan al conectarlas con 'AND'. Sólo se despliegan `lastName` y `firstName`.

Consulta SQL:

```
SELECT      lastName, firstName
FROM        Student
WHERE       major = 'Math'
            AND credits > 30;
```

Resultado:

lastName	firstName
Jones	Mary
Chin	Ann

El predicado puede ser tan complejo como sea necesario con el uso de los operadores de comparación estándar =, <>, <, <=, >, >= y los operadores lógicos estándar AND, OR y NOT, con paréntesis, si se necesita o desea, para mostrar orden de evaluación.

6.4.2 SELECT usando tablas múltiples

- Ejemplo 7. Combinación natural

  Pregunta: Encontrar las ID y nombres de todos los estudiantes que toman ART103A.

  Solución: Esta pregunta requiere el uso de dos tablas. Primero se busca en la tabla `Enroll` para registros donde el `classNumber` es 'ART103A'. Luego se busca en la tabla `Student` los registros con valores `stuId` coincidentes, y se combinan dichos registros en una nueva tabla. A partir de esta tabla, se encuentran `lastName` y `firstName`. Esto es similar a la operación JOIN en el álgebra relacional. SQL permite hacer una combinación natural, como se describe en la sección 4.6.2, al nombrar las tablas involucradas y expresar en el predicado la condición que deben cumplir los registros en el campo común.

  Consulta SQL:

```
SELECT      Enroll.stuId, lastName, firstName
FROM        Student, Enroll
WHERE       classNumber = 'ART103A'
            AND Enroll.stuId = Student.stuId;
```

  Resultado:

stuId	lastName	firstName
S1001	Smith	Tom
S1010	Burns	Edward
S1002	Chin	Ann

Note que se usó el nombre calificado por `stuId` en la línea SELECT. Pudo haberse escrito `Student.stuId` en lugar de `Enroll.stuId`, pero necesitaba usar uno de los nombres de tabla, porque `stuId` aparece en ambas tablas en la línea FROM. No necesita usar el nombre calificado para `classNumber` porque no aparece en la tabla `Student`. El hecho de que aparezca en la tabla `Class` es irrelevante, pues dicha tabla no se mencionó en la línea FROM. Desde luego, tiene que escribir ambos nombres calificados para `stuId` en la línea WHERE.

¿Por qué es necesaria la condición "`Enroll.stuId=Student.stuId`"? La respuesta es que es esencial. Cuando un sistema de base de datos relacional realiza una combinación, actúa como si primero forma un producto cartesiano, como se describió en la sección 4.6.2, de modo que (teóricamente) se forma una tabla intermedia que contiene las combinaciones de todos los registros de la tabla `Student` con los registros de la tabla `Enroll`. Incluso si el sistema se restringe a sí mismo a registros en `Enroll` que satisfagan

la condición "`classNumber`='ART103A'", la tabla intermedia numera 6*3 o 18 registros. Por ejemplo, uno de estos registros intermedios es:

S1015 Jones Mary Math 42 ART103A S1001 A

No se tiene interés en este registro, pues este estudiante no es una de las personas en la clase ART103A. Por tanto, se agrega la condición de que los valores `stuId` deben ser iguales. Esto reduce la tabla intermedia a tres registros.

- Ejemplo 8. Combinación natural con ordenamiento

  Pregunta: Encontrar `stuId` y `grade` de todos los estudiantes que tomen cualquier curso impartido por el miembro `Faculty` cuyo `facId` es F110. Ordenar en orden por `stuId`.

  Solución: Necesita buscar en la tabla `Class` para encontrar el `classNumber` de todos los cursos impartidos por F110. Luego se busca en la tabla `Enroll` los registros con valores `classNumber` coincidentes y obtener la combinación de las tablas. A partir de esto se encuentran los correspondientes `stuId` y `grade`. Puesto que se usan dos tablas, esto se escribirá como una combinación.

  Consulta SQL:

```
SELECT      stuId,grade
FROM        Class,Enroll
WHERE       facId = 'F110' AND Class.classNumber
            = Enroll.classNumber
ORDER BY    stuId ASC;
```

  Resultado:

stuId	grade
S1002	B
S1010	
S1020	A

- Ejemplo 9. Combinación natural de tres tablas

  Pregunta: Encontrar los números de curso y los nombres y especialidades de todos los estudiantes inscritos en los cursos impartidos por el miembro `Faculty` F110.

  Solución: Como en el ejemplo anterior, necesita empezar en la tabla `Class` para encontrar el `classNumber` de todos los cursos impartidos por F110. Luego se comparan éstos con los valores `classNumber` en la tabla `Enroll` para encontrar los valores `stuId` de todos los estudiantes en dichos cursos. Luego se busca en la tabla `Student` para encontrar los nombres y especialidades de todos los estudiantes inscritos en ellos.

  Consulta SQL:

```
SELECT    Enroll.classNumber, lastName,
          firstName, major
FROM      Class, Enroll, Student
WHERE     facId = 'F110'
          AND Class.classNumber =
          Enroll.classNumber
          AND Enroll.stuId = Student.stuId;
```

  Resultado:

classNumber	lastName	firstName	major
MTH101B	Rivera	Jane	CSC
MTH103C	Burns	Edward	Art
MTH103C	Chin	Ann	Math

Ésta fue una combinación natural de tres tablas y requirió dos conjuntos de columnas comunes. Se usó la condición de igualdad para ambos conjuntos en la línea WHERE. Es posible que haya notado que el orden de los nombres de la tabla en la línea FROM correspondía al orden en el que aparecieron en el plan de solución, pero esto no es necesario. SQL ignora el orden en el que las tablas se nombran en la línea FROM. Lo mismo es cierto del orden en el que se escriben las varias condiciones que constituyen el predicado en la línea WHERE. La mayoría de los sistemas de gestión de base de datos relacionales sofisticados eligen cuál tabla usar primero y cuál condición verificar primero, usando un optimizador para identificar el método más eficiente de lograr cualquier recuperación antes de elegir un plan.

- Ejemplo 10. Uso de alias

  Pregunta: Obtener una lista de todos los cursos que se imparten en el mismo salón, con sus horarios y números de salón.

  Solución: Esto requiere comparar la tabla `Class` consigo misma, y sería útil si hubiera dos copias de la tabla de modo que se podría usar una combinación natural. Puede pretender que existen dos copias de una tabla al darle dos "alias", por ejemplo, COPY y COPY2, y luego tratar estos nombres como si fuesen los nombres de dos tablas distintas. Los "alias" se introducen en la línea FROM al escribirlos inmediatamente después de los nombres de tabla reales. Luego se tienen los alias disponibles para usar en las otras líneas de la consulta.

  Consulta SQL:

```
SELECT   COPY1.classNumber, COPY1.schedule, COPY1.room,
         COPY2.classNumber, COPY2.schedule
FROM     Class COPY1, Class COPY2
WHERE    COPY1.room = COPY2.room
         AND COPY1.classNumber < COPY2.classNumber ;
```

  Resultado:

COPY1.classNumber	COPY1.schedule	COPY1.room	COPY2.classNumber	COPY2.schedule
ART103A	MWF9	H221	HST205A	MWF11
CSC201A	TUTHF10	M110	CSC203A	MTHF12
MTH101B	MTUTH9	H225	MTH103C	MWF11

Note que debe usar los nombres calificados en la línea SELECT incluso antes de introducir los "alias". Esto es necesario porque cada columna en la tabla `Class` ahora aparece dos veces, una vez en cada copia. Se agregó la segunda condición "COPY1.`classNumber` < COPY2.COURSE#" para evitar incluir a cada curso, pues cada curso obviamente satisface el requisito de que se imparte en el mismo salón como él mismo. También evita que aparezcan registros con los dos cursos invertidos. Por ejemplo, puesto que se tiene

ART103A MWF9 H221 HST205A MWF11

no se necesita el registro

HST205A MWF11 H221 ART103A MWF9

De modo incidental se pueden introducir alias en cualquier SELECT, aun cuando no se requieren.

- Ejemplo 11. Combinación sin condición de igualdad

  Pregunta: Encontrar todas las combinaciones de estudiantes y `Faculty` donde la especialidad del estudiante sea diferente del departamento del miembro `Faculty`.

  Solución: Esta solicitud inusual es para ilustrar una combinación en la que la condición no sea una igualdad en un campo común. En este caso, los campos que se exa-

minan, `major` y `department`, incluso no tienen el mismo nombre. Sin embargo, se les puede comparar ya que tienen el mismo dominio. Dado que no se dice cuáles columnas mostrar en el resultado, se usa su criterio.

Consulta SQL:

```
SELECT   stuId, lastName, firstName, major, facId, name, department
FROM     Student, Faculty
WHERE    Student.major <> Faculty.department;
```

Resultado:

```
stuId    lastName  firstName  major    facId  name    department
S1001    Smith     Tom        History  F101   Adams   Art
S1001    Smith     Tom        History  F105   Tanaka  CS
S1001    Smith     Tom        History  F110   Byrne   Math
S1001    Smith     Tom        History  F221   Smith   CS
S1010    Burns     Edward     Art      F202   Smith   History
.....................................
.....................................
.....................................
S1013    McCarthy  Owen       Math     F221   Smith   CS
```

Como en el álgebra relacional, una combinación se puede realizar sobre cualesquiera dos tablas al simplemente formar el producto cartesiano. Aunque por lo general se quiere la combinación natural como en los ejemplos anteriores, se puede usar cualquier tipo de predicado como la condición para la combinación. Sin embargo, si se quieren comparar dos columnas, deben tener los mismos dominios. Note que se usaron nombres calificados en la línea WHERE. Esto realmente no era necesario, porque cada nombre de columna era único, pero se hizo para que la condición fuese más fácil de seguir.

- Ejemplo 12. Uso de una subconsulta con igualdad

  Pregunta: Encontrar los números de todos los cursos impartidos por Byrne del departamento de matemáticas.

  Solución: Ya se sabe cómo hacer esto con el uso de una combinación natural, pero existe otra forma de encontrar la solución. En lugar de imaginar una combinación a partir de la cual elegir registros con el mismo `facId`, esto se podría visualizar como dos consultas separadas. Para la primera, se iría a la tabla `Faculty` y se encontrarían los registros con nombre de Byrne y `department` Math. Podría hacer una anotación del correspondiente `facId`. Luego podría tomar el resultado de dicha consulta, a saber F110, y buscar la tabla `Class` para registros con dicho valor en `facId`. Una vez encontrados se mostraría el `classNumber`. SQL permite secuenciar estas consultas de modo que el resultado de la primera se pueda usar en la segunda, como se muestra a continuación:

  Consulta SQL:

```
SELECT    classNumber
FROM      Class
WHERE     facId =
          (SELECT  facId
          FROM     Faculty
          WHERE    name = 'Byrne'
                   AND department = 'Math');
```

  Resultado:

```
classNumber
MTH101B
MTH103C
```

Note que este resultado podría producirse mediante la siguiente consulta SQL, usando una combinación:

```
SELECT      classNumber
FROM        Class, Faculty
WHERE       name = 'Byrne' AND department = 'Math' AND Class.facId = Faculty.
            facId;
```

En lugar de una combinación se puede usar una subconsulta, siempre que el resultado a mostrar esté contenido en una sola tabla y los datos recuperados de la subconsulta consistan sólo de una columna. Cuando escriba una subconsulta que involucre dos tablas, sólo nombre una tabla en cada SELECT. La consulta a realizar primero, la subconsulta, es la que está entre paréntesis, después de la primera línea WHERE. La consulta principal se realiza usando el resultado de la subconsulta. Normalmente se quiere que el valor de algún campo en la tabla mencionada en la consulta principal coincida con el valor de algún campo de la tabla en la subconsulta. En este ejemplo se sabía que sólo se obtendría un valor de la subconsulta, pues `facId` es la clave de `Faculty`, de modo que se produciría un valor único. Por tanto, se tiene posibilidad de usar igualdad como el operador. Sin embargo, se pueden usar condiciones distintas a la igualdad. Es posible usar cualquier operador de comparación sencillo en una subconsulta de la que usted sepa que se producirá un solo valor. Dado que la subconsulta se realiza primero, el SELECT . . . FROM . . . WHERE de la subconsulta en realidad se sustituye por el valor recuperado, de modo que la consulta principal cambia a lo siguiente:

```
SELECT      classNumber
FROM        Class
WHERE       facId = ('F110');
```

- Ejemplo 13. Subconsulta usando 'IN'

  Pregunta: Encontrar los nombres e ID de todos los miembros de `Faculty` que impartan una clase en el salón H221.

  Solución: Se necesitan dos tablas, `Class` y `Faculty`, para responder esta pregunta. También se ve que los nombres e ID aparecen ambos en la tabla `Faculty`, así que se tiene la opción de o una combinación o una subconsulta. Si se usa una subconsulta, comience con la tabla `Class` con el fin de encontrar los valores `facId` para cualquier curso que se imparta en el salón H221. Se encuentran dos de tales entradas, así que se hace una anotación de dichos valores. Luego se va a la tabla `Faculty` y se compara el valor `facId` de cada registro en dicha tabla con los dos valores ID de `Class` y se muestran los correspondientes `facId` y `name`.

  Consulta SQL:

```
SELECT    name, facId
FROM      Faculty
WHERE     facId IN
          (SELECT    facId
          FROM       Class
          WHERE      room = 'H221');
```

  Resultado:

name	facId
Adams	F101
Smith	F202

En la línea WHERE de la consulta principal se usó IN, en lugar de =, porque el resultado de la subconsulta es un conjunto de valores en vez de un solo valor. Se pide que el `facId` en `Faculty` coincida con cualquier miembro del conjunto de valores obtenidos de la subcon-

sulta. Cuando la subconsulta se sustituye con los valores recuperados, la consulta principal se convierte en

```
SELECT      name, facId
FROM        Faculty
WHERE       FACID IN ('F101','F202');
```

El IN es una forma más general de subconsulta que el operador comparación, que está restringido al caso donde se produce un solo valor. También se puede usar la forma negativa 'NOT IN', que se evaluará verdadero si el registro tiene un valor de campo que no está en el conjunto de valores recuperados por la subconsulta.

- Ejemplo 14. Subconsultas anidadas

  Pregunta: Obtener una lista alfabética de los nombres de ID de todos los estudiantes en cualquier clase que imparta F110.

  Solución: Se necesitan tres tablas, `Student`, `Enroll` y `Class`, para responder esta pregunta. Sin embargo, los valores a mostrar aparecen en una tabla, `Student`, así que se puede usar una subconsulta. Primero verifique la tabla `Class` para encontrar el `classNumber` de todos los cursos impartidos por F110. Se encuentran dos valores, MTH101B y MTH103C. A continuación se va a la tabla `Enroll` para encontrar el `stuId` de todos los estudiantes en cualquiera de estos cursos. Se encuentran tres valores: S1020, S1010 y S1002. Ahora se busca en la tabla `Student` para encontrar los registros con valores `stuId` coincidentes, y se despliegan `stuId`, `lastName` y `firstName` en orden alfabético por apellido.

  Consulta SQL:

```
SELECT     lastName, firstName, stuId
FROM       Student
WHERE      stuId IN
           (SELECT     stuId
           FROM        Enroll
           WHERE       classNumber IN
                       (SELECT  classNumber
                       FROM     Class
                       WHERE    facId = 'F110'))
ORDER BY lastName, firstName ASC;
```

  Resultado:

lastName	firstName	stuId
Burns	Edward	S1010
Chin	Ann	S1002
Rivera	Jane	S1020

En la ejecución, el SELECT más profundamente anidado se realiza primero, y se sustituye por los valores recuperados, así que se tiene:

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     stuId IN
          (SELECT  stuId
          FROM     Enroll
          WHERE    classNumber IN
                   ('MTH101B', 'MTH103C'))
ORDER BY  lastName, firstName ASC;
```

A continuación se realiza la subconsulta sobre `Enroll` y se obtiene:

```
SELECT      lastName, firstName, stuId
FROM        Student
WHERE       stuId IN
            ('S1020', 'S1010', 'S1002')
ORDER BY    lastName, firstName ASC;
```

Finalmente, se realiza la consulta principal, y se obtiene el resultado que se mostró con anterioridad. Note que el ordenamiento se refiere al resultado final, no a cualquier paso intermedio. Observe también que cualquier parte de la operación se podía realizar como una combinación natural y la otra parte como una subconsulta, al mezclar ambos métodos.

- Ejemplo 15. Consulta usando EXISTS

  Pregunta: Encontrar los nombres de todos los estudiantes inscritos en CSC201A.

  Solución: Ya se sabe cómo escribir esto usando una combinación o una subconsulta con IN. Sin embargo, otra forma de expresar esta consulta es usar el cuantificador existencial, EXISTS, con una subconsulta.

  Consulta SQL:

```
SELECT    lastName, firstName
FROM      Student
WHERE     EXISTS
          (SELECT  *
          FROM     Enroll
          WHERE    Enroll.stuId = Student.stuId
          AND      classNumber = 'CSC201A');
```

  Resultado:

lastName	firstName
Rivera	Jane
Chin	Ann

Esta consulta se podría parafrasear como "Encuentre `lastName` y `firstName` de todos los estudiantes tales que exista un registro `Enroll` que contenga su `stuId` con un `classNumber` de CSC201A". La prueba para inclusión es la existencia de tal registro. Si existe, el "EXISTS (SELECT FROM...;" evalúa a verdadero.

Note que necesita usar el nombre de la tabla de consulta principal (`Student`) en la subconsulta para expresar la condición `Student.stuId = Enroll.stuId`. En general, se evita mencionar una tabla no citada en el FROM para dicha consulta particular, pero es necesario y permisible hacerlo en este caso. Esta forma se llama subconsulta **correlacionada**, pues la tabla en la subconsulta se compara con la tabla en la consulta principal.

- Ejemplo 16. Consulta usando NOT EXISTS

  Pregunta: Encontrar los nombres de todos los estudiantes que no están inscritos en CSC201A.

  Solución: A diferencia del ejemplo anterior, no se puede expresar fácilmente esto usando una combinación o una subconsulta IN. En vez de ello, se usará NOT EXISTS.

  Consulta SQL:

```
SELECT    lastName, firstName
FROM      Student
WHERE     NOT EXISTS
          (SELECT     *
```

```
FROM      Enroll
WHERE     Student.stuId = Enroll.stuId
AND       classNumber = 'CSC201A');
```

Resultado:

lastName	firstName
Smith	Tom
Burns	Edward
Jones	Mary
McCarthy	Owen
Lee	Perry

Esta consulta se podría parafrasear como "Seleccionar nombres de estudiante de la tabla `Student` tales que no exista registro `Enroll` que contenga sus valores `stuId` con `classNumber` de CSC201A".

6.4.3 SELECT con otros operadores

- Ejemplo 17. Consulta usando UNION

  Pregunta: Obtener las ID de todos los `Faculty` que estén asignados al departamento de historia o que den clase en el salón H221.

  Solución: Es fácil escribir una consulta para cualquiera de las condiciones y los resultados de las dos consultas se pueden combinar usando un operador UNION. La UNION en SQL es el operador estándar del álgebra relacional para unión de conjuntos y funciona en la forma esperada, lo que elimina duplicados.

  Consulta SQL:

```
SELECT    facId
FROM      Faculty
WHERE     department = 'History'
UNION
SELECT    facId
FROM      Class
WHERE     room ='H221';
```

  Resultado:

facId
F115
F101

- Ejemplo 18. Uso de funciones

  Pregunta: Encontrar el número total de estudiantes inscritos en ART103A.

  Solución: Aunque ésta es una pregunta simple, no se tiene posibilidad de expresarla como una consulta SQL en el momento, porque todavía no se ha visto alguna forma de operar sobre colecciones de filas o columnas. Se necesitan algunas funciones para hacerlo. SQL tiene cinco funciones internas: COUNT, SUM, AVG, MAX y MIN. Se usará COUNT, que regresa el número de valores en una columna.

  Consulta SQL:

```
SELECT    COUNT (DISTINCT stuId)
FROM      Enroll
WHERE     classNumber = 'ART103A';
```

  Resultado:

```
3
```

Las funciones internas operan sobre una sola columna de una tabla. Cada una de ellas elimina primero los valores nulos y sólo opera sobre los restantes valores no nulos. Las funciones regresan un solo valor, que se definen del modo siguiente:

COUNT	regresa el número de valores en la columna
SUM	regresa la suma de los valores en la columna
AVG	regresa el promedio de los valores en la columna
MAX	regresa el valor más grande en la columna
MIN	regresa el valor más pequeño en la columna

COUNT, MAX y MIN se aplican a campos numéricos y no numéricos, pero SUM y AVG sólo se pueden usar en campos numéricos. La secuencia de recopilación se usa para determinar el orden de datos no numéricos. Si se quiere eliminar valores duplicados antes de comenzar, use la palabra DISTINCT antes del nombre de columna en la línea SELECT. COUNT(*) es un uso especial de COUNT. Su propósito es contar todas las filas de una tabla, sin importar si ocurren valores nulos o duplicados. Excepto por COUNT(*), siempre debe usar DISTINCT con la función COUNT, como se hizo en el ejemplo anterior. Si usa DISTINCT con MAX o MIN no tendrá efecto, pues el valor más grande o más pequeño permanece igual incluso si dos tuplas lo comparten. Sin embargo, DISTINCT usualmente tiene un efecto sobre el resultado de SUM o AVG, así que el usuario debe entender si los duplicados se incluyen o no para calcular éste. Las referencias a función aparecen en la línea SELECT de una consulta o una subconsulta.

Ejemplos de funciones adicionales:

Ejemplo (a) Encontrar el número de departamentos que tienen `Faculty` en ellos. Puesto que no se quiere contar un departamento más de una vez, aquí se usa DISTINCT.

```
SELECT    COUNT(DISTINCT department)
FROM      Faculty;
```

Ejemplo (b) Encontrar el número promedio de créditos que tienen los estudiantes. Aquí no se quiere usar DISTINCT, porque si dos estudiantes tienen el mismo número de créditos, ambos se deben contar en el promedio.

```
SELECT    AVG(credits)
FROM      Student;
```

Ejemplo (c) Encontrar al estudiante con el número más grande de créditos. Puesto que se quiere que los créditos del estudiante sean igual al máximo, es necesario encontrar dicho máximo primero, así que se usa una subconsulta para encontrarlo.

```
SELECT     stuId, lastName, firstName
FROM       Student
WHERE      credits =
           (SELECT   MAX(credits)
           FROM      Student);
```

Ejemplo (d) Encontrar la ID del estudiante con la calificación más alta en cualquier curso. Puesto que se quiere la calificación más alta, puede parecer que aquí se debe usar la función MAX. Un vistazo más cercano a la tabla revela que las calificaciones son letras, A, B, C, etc. Para esta escala, la mejor calificación es aquella que está primero en el abecedario, así que en realidad se quiere MIN. Si las calificaciones fuesen numéricas se habría querido MAX.

```
SELECT    stuId
FROM      Enroll
WHERE     grade =
          (SELECT  MIN(grade)
          FROM     Enroll);
```

Ejemplo (e) Encontrar los nombres e ID de los estudiantes que tienen menos que el número promedio de créditos.

```
SELECT    lastName, firstName, stuId
FROM      Student
WHERE     credits <
          (SELECT  AVG(credits)
          FROM     Student);
```

- Ejemplo 19. Uso de una expresión y una constante cadena

  Pregunta: Si supone que cada curso tiene tres créditos, para cada estudiante, el número de cursos que ha completado.

  Solución: Puede calcular el número de cursos al dividir el número de créditos entre tres. Se puede usar la expresión `credits/3` en SELECT para mostrar el número de cursos. Dado que no se tiene tal nombre de columna se usará una constante cadena como etiqueta. Las constantes cadena que aparecen en la línea SELECT simplemente se imprimen en el resultado.

  Consulta SQL:

```
SELECT       stuId, 'Number of courses =', credits/3
FROM         Student;
```

  Resultado:

```
stuId
S1001 Number of courses =  30
S1010 Number of courses =  21
S1015 Number of courses =  14
S1005 Number of courses =   1
S1002 Number of courses =  12
S1020 Number of courses =   5
S1013 Number of courses =   0
```

Al combinar constantes, nombres de columna, operadores aritméticos, funciones internas y paréntesis, el usuario puede personalizar las recuperaciones.

- Ejemplo 20. Uso de GROUP BY

  Pregunta: Para cada curso, mostrar el número de estudiantes inscritos.

  Solución: Se quiere usar la función COUNT, pero es necesario aplicarla a cada curso individualmente. El GROUP BY permite poner juntos todos los registros con un solo valor en el campo especificado. Luego se puede aplicar cualquier función a cualquier campo en cada grupo, siempre que el resultado sea un solo valor para el grupo.

  Consulta SQL:

```
SELECT       classNumber, COUNT(*)
FROM         Enroll
GROUP BY     classNumber;
```

  Resultado:

```
classNumber
ART103A         3
CSC201A         2
MTH101B         1
HST205A         1
MTH103C         2
```

Note que, en esta consulta, podría usar COUNT(DISTINCT `stuId`) en lugar de COUNT(*).

- Ejemplo 21. Uso de HAVING

  Problema: Encontrar todos los cursos en los que estén inscritos menos de tres estudiantes.

  Solución: Ésta es una pregunta acerca de una característica de los grupos formados en el ejemplo anterior. HAVING se usa para determinar cuáles grupos tienen alguna cualidad, tal como WHERE se usa con tuplas para determinar cuáles registros tienen alguna cualidad. No se permite usar HAVING sin un GROUP BY, y el predicado en la línea HAVING debe tener un solo valor para cada grupo.

  Consulta SQL:

```
SELECT     classNumber
FROM       Enroll
GROUP BY   classNumber
HAVING     COUNT(*) < 3 ;
```

  Resultado:

```
classNumber
CSC201A
MTH101B
HST205A
MTH103C
```

- Ejemplo 22. Uso de LIKE

  Problema: Obtener detalles de todos los cursos MTH.

  Solución: No se quiere especificar los números de curso exactos, lo que se quiere es que las tres primeras letras de `classNumber` sean MTH. SQL permite el uso de LIKE en el predicado con el fin de mostrar una cadena patrón para campos carácter. Se recuperarán los registros cuyas columnas especificadas coincidan con el patrón.

  Consulta SQL:

```
SELECT     *
FROM       Class
WHERE      classNumber LIKE 'MTH%';
```

  Resultado:

```
classNumber     facId        schedule       room
MTH101B         F110         MTUTH9         H225
MTH103C         F110         MWF11          H225
```

En la cadena patrón se pueden usar los siguientes símbolos:

- % El carácter porcentaje representa cualquier secuencia de caracteres de cualquier longitud >= 0.
- _ El carácter guión bajo representa cualquier carácter solo.

Todos los otros caracteres en el patrón se representan ellos mismos.

Ejemplos:

- **`classNumber` LIKE `'MTH%'`** significa que las tres primeras letras deben ser MTH, pero el resto de la cadena puede ser cualquier carácter.
- **`stuId` LIKE `'S_ _ _ _'`** significa que debe haber cinco caracteres, el primero de los cuales debe ser S.
- **`schedule` LIKE `'%9'`** significa cualquier secuencia de caracteres, de longitud al menos uno, con el último carácter un nueve.

- **`classNumber` LIKE `'%101%'`** significa una secuencia de caracteres de cualquier longitud que contenga 101. Note que 101 podría ser el primero, último o únicos caracteres, así como estar en cualquier parte en medio de la cadena.
- **`name` NOT LIKE `'A%'`** significa que el nombre no puede comenzar con A.

- Ejemplo 23. Uso de NULL

  Pregunta: Encontrar el `stuId` y `classNumber` de todos los estudiantes cuyas calificaciones faltan en dicho curso.

  Solución: De la tabla `Enroll` se puede ver que existen dos de tales registros. Puede pensar que podría acceder a ellos al especificar que las calificaciones no sean A, B, C, D o F, pero éste no es el caso. Una calificación nula se considera que tiene "desconocido" como valor, así que es imposible juzgar si es igual o no es igual a otra calificación. Si se pusiera la condición "WHERE grade <>'A' AND grade <>'B' AND grade <>'C' AND grade <>'D' AND grade <>'F'" se obtendría de vuelta una tabla vacía, en lugar de los dos registros que se quieren. SQL usa la expresión lógica

  nombre_columna IS [NOT] NULL

  a fin de probar para valores nulos en una columna.

  Consulta SQL:

```
SELECT     classNumber,stuId
FROM       Enroll
WHERE      grade IS NULL;
```

  Resultado:

classNumber	stuId
ART103A	S1010
MTH103C	S1010

Note que es ilegal escribir "WHERE `grade` = NULL", porque un predicado que involucre operadores comparación con NULL evaluará para "desconocido" en lugar de "verdadero" o "falso". Además, la línea WHERE es la única en la que NULL puede aparecer en un enunciado SELECT.

- Ejemplo 24. Consultas recursivas

  SQL: 1999 permite consultas recursivas, que son consultas que se ejecutan repetidamente hasta que no se encuentran nuevos resultados. Por ejemplo, considere una tabla CSCCourse, como se muestra en la figura 6.4(a). Su estructura es:

  `CSCCourse(cursoNumero, cursoTitulo, creditos, prerrequisitoCursoNumero)`

  Por simplicidad, se supone que un curso puede tener cuando mucho un curso prerrequisitos inmediato. El número de curso prerrequisitos funciona como una clave externa para la tabla CSCCourse, que se refiere a la clave primaria (número de curso) de un curso diferente.

  Problema: Encontrar todos los prerrequisitos de un curso, incluidos prerrequisitos de prerrequisitos para dicho curso.

  Consulta SQL:

```
WITH RECURSIVE
Prereqs (cursoNumero, prerrequisitoCursoNumero) AS
  (  SELECT cursoNumero, prerrequisitoCursoNumero
     FROM CSCCourse
     UNION
     SELECT (COPY1.cursoNumero, COPY2.prerrequisitoCursoNumero
     FROM Prereqs COPY1, CSCCourse COPY2
     WHERE COPY1.prerrequisitoCursoNumero = COPY2.cursoNumero);
```

```
SELECT *
FROM Prereqs
ORDER BY cursoNumero, prerrequisitoCursoNumero;
```

Esta consulta desplegará cada número de curso, junto con todos los prerrequisitos de dicho curso, incluidos los prerrequisitos de prerrequisitos, etc., toda la ruta hasta el curso inicial en la secuencia de sus prerrequisitos. El resultado se muestra en la figura 6.4(b).

6.4.4 Operadores para actualización: UPDATE, INSERT, DELETE

El operador UPDATE (actualizar) se usa para cambiar valores en registros ya almacenados en una tabla. Se usa en una tabla a la vez y puede cambiar cero, uno o muchos registros, dependiendo del predicado. Su forma es:

```
UPDATE    nombre tabla
SET       nombre_columna = expresión
          [nombre_columna = expresión] . . .
[WHERE    predicado];
```

Note que no es necesario especificar el valor actual del campo, aunque el valor presente se puede usar en la expresión para determinar el nuevo valor. El enunciado SET es en realidad un enunciado de asignación y funciona en la forma usual.

- Ejemplo 1. Actualización de un solo campo de un registro

  Operación: Cambiar la especialidad de S1020 a Music.

  Comando SQL:

  ```
  UPDATE  Student
  SET     major = 'Music'
  WHERE   stuId = 'S1020';
  ```

- Ejemplo 2. Actualización de varios campos de un registro

  Operación: Cambiar el departamento de Tanaka a MIS y clasificarlo como asistente (Assistant).

  Comando SQL:

  ```
  UPDATE  Faculty
  SET     department = 'MIS'
          rank = 'Assistant'
  WHERE   name = 'Tanaka';
  ```

CSCCourse

cursoNumero	cursoTitulo	creditos	prerrequisitoCursoNumero
101	Introducción a la computación	3	
102	Aplicaciones de computadoras	3	101
201	Programación 1	4	101
202	Programación 2	4	201
301	Estructuras de datos y algoritmos	3	202
310	Sistemas operativos	3	202
320	Sistemas de bases de datos	3	301
410	Sistemas operativos avanzados	3	310
420	Sistemas de base de datos avanzados	3	320

FIGURA 6.4(a)
Tabla CSCCurso para demostrar las consultas recursivas

FIGURA 6.4(b)
Resultado de consulta recursiva

Prereqs

cursoNumero	prerrequisitoCursoNumero
101	
102	
102	101
201	
201	101
202	
202	101
202	201
301	
301	101
301	201
301	202
310	
310	101
310	201
310	202
320	
320	101
320	201
320	202
320	301
410	
410	101
410	201
410	202
410	310
420	
420	101
420	201
420	202
420	301
420	320

- Ejemplo 3. Actualización usando NULL

  Operación: Cambiar la especialidad de S1013 de Math a NULL. Para insertar un valor nulo en un campo que ya tiene un valor real, debe usar la forma:

  SET *nombre_columna* = NULL

  Comando SQL:

```
UPDATE  Student
SET     major = NULL
WHERE   stuId = 'S1013';
```

- Ejemplo 4. Actualización de varios registros

  Operación: Cambiar las calificaciones de todos los estudiantes en CSC201A a A.

  Comando SQL:

```
UPDATE     Enroll
SET        grade = 'A'
WHERE      classNumber = 'CSC201A';
```

- Ejemplo 5. Actualización de todos los registros.

  Operación: Dar a todos los estudiantes tres créditos adicionales.

  Comando SQL:

```
UPDATE  Student
SET     credits = credits + 3;
```

Note que no se necesita la línea WHERE, porque se actualizaron todos los registros.

- Ejemplo 6. Actualización con una subconsulta

  Operación: Cambiar el salón a B220 para todos los cursos que imparte Tanaka.

  Comando SQL:

```
UPDATE  Class
SET     room = 'B220'
WHERE   facId =
        (SELECT     facId
        FROM        Faculty
        WHERE       name = 'Tanaka');
```

El operador INSERT se usa para poner nuevos registros en una tabla. Por lo general, no se usa para cargar toda una tabla, porque el sistema de gestión de base de datos usualmente tiene una utilidad de carga (load) para manejar dicha tarea. Sin embargo, INSERT es útil para agregar uno o algunos registros a una tabla. Su forma es:

```
INSERT
INTO      nombre_tabla [(nombre_col [,nombre_col]. . .)]
VALUES    (constante [,constante] . . .);
```

- Ejemplo 1. Insertar un solo registro, con todos los campos especificados

  Operación: Insertar un nuevo registro `Faculty` con ID de F330, nombre Jones, departamento CSC y clasificación de Instructor.

  Comando SQL:

```
INSERT
INTO    Faculty (facId, name, department, rank)
VALUES  ('F330', 'Jones', 'CSC', 'Instructor');
```

- Ejemplo 2. Insertar un solo registro, sin especificar campos

  Operación: Insertar un nuevo registro de estudiante con ID de S1030, nombre Alice Hunt, especialidad art y 12 créditos

  Consulta SQL:

```
INSERT
INTO    Student
VALUES  ('S1030', 'Hunt', 'Alice', 'Art', 12);
```

Note que no fue necesario especificar nombres de campo, porque el sistema supone que se quieren todos los campos en la tabla. Podría hacer lo mismo para el ejemplo anterior.

- Ejemplo 3. Insertar un registro con valor nulo en un campo

  Operación: Insertar un nuevo registro de estudiante con ID de S1031, nombre Maria Bono, cero créditos y sin especialidad.

  Comando SQL:

```
INSERT
INTO    Student (lastName, firstName, stuId, credits)
VALUES  ('Bono', 'Maria', 'S1031', 0);
```

Note que se reordenaron los nombres de campo, pero no hay confusión porque se entiende que el orden de los valores coincide con el orden de los campos mencionados en el INTO, sin importar su orden en la tabla. Note también que el cero es un valor real para `credits`, no un valor nulo. `major` se establecerá en nulo, pues se excluye de la lista de campos en la línea INTO.

- Ejemplo 4. Insertar múltiples registros

  Operación: Crear y llenar una nueva tabla que muestre cada curso y el número de estudiantes inscritos en ellos.

  Comando SQL:

```
CREATE TABLE Enrollment
  (classNumber CHAR(7) NOT NULL,
  Students SMALLINT);
INSERT
INTO Enrollment        (classNumber, Students)
  SELECT               classNumber, COUNT(*)
  FROM                 Enroll
  GROUP BY             classNumber;
```

Aquí se creó una nueva tabla, `Enrollment`, y se llenó al tomar datos de una tabla existente, `Enroll`. Si `Enroll` es como aparece en la figura 6.3, `Enrollment` ahora se ve como esto:

Enrollment

classNumber	Students
ART103A	3
CSC201A	2
MTH101B	1
HST205A	1
MTH103C	2

La tabla `Enrollment` ahora está disponible para que el usuario la manipule, tal como lo sería cualquiera otra tabla. Se puede actualizar según se requiera, pero no se actualizará automáticamente cuando `Enroll` se actualice.

El DELETE se usa para borrar registros. El número de registros borrados puede ser cero, uno o muchos, dependiendo de cuántos satisfagan el predicado. La forma de este comando es:

```
DELETE
FROM        nombre_tabla
WHERE       predicado;
```

- Ejemplo 1. Borrado de un solo registro

  Operación: Borrar el registro del estudiante S1020.

  Comando SQL:

```
DELETE
FROM        Student
WHERE       stuId = 'S1020';
```

- Ejemplo 2. Borrado de muchos registros

  Operación: Borrar todos los registros de inscripción para el estudiante S1020.

  Comando SQL:

```
DELETE
FROM        Enroll
WHERE       stuId = 'S1020';
```

- Ejemplo 3. Borrado de todos los registros de una tabla

  Operación: Borrar todos los registros de clase.

Si borra los registros `Class` y permite que permanezcan sus correspondientes registros `Enroll` se perdería integridad referencial, porque los registros `Enroll` se referirían entonces a clases que ya no existen. Sin embargo, si las tablas se crearon con Oracle y los comandos que se muestran en la figura 6.2, el comando DELETE no funcionará en la tabla `Class` a menos que primero se borren los registros `Enroll` para cualquier estudiante registrado en la clase, porque se escribió

```
CONSTRAINT Enroll_classNumber_fk FOREIGN KEY (classNumber) REFERENCES Class
(classNumber)
```

para la tabla `Enroll`. No obstante, si supone que se borraron los registros `Enroll`, entonces se pueden borrar los registros `Class`.

Comando SQL:

```
DELETE
FROM       Class;
```

Esto removería todos los registros de la tabla `Class`, pero su estructura permanecería, así que en ella podría agregar nuevos registros en cualquier momento.

- Ejemplo 4. DELETE con una subconsulta

  Operación: Borrar todos los registros de inscripción para Owen McCarthy.

  Comando SQL:

```
DELETE
FROM        Enroll
WHERE       stuId =
            (SELECT    stuId
            FROM       Student
            WHERE      lastName = 'Mc Carthy'
                       AND firstName = 'Owen');
```

Puesto que no existían tales registros, este enunciado no tendrá efecto sobre `Enroll`.

6.5 Bases de datos activas

El DBMS tiene más poder al asegurar la integridad en una base de datos que las restricciones de columna y tabla discutidas en la sección 6.3. Una **base de datos activa** es aquella en la que el DBMS monitorea a los contenidos con la finalidad de evitar que ocurran estados ilegales, usando restricciones y disparadores (triggers).

6.5.1 Habilitación y deshabilitación de restricciones

Las restricciones de columna y tabla descritas en la sección 6.3 se identifican cuando se crea la tabla, y se verifican siempre que se realiza un cambio a la base de datos, para garantizar

que el nuevo estado es válido. Los cambios que podrían resultar en estados inválidos incluye inserción, borrado y actualización de registros. Por tanto, el DBMS verifica las restricciones siempre que se realiza una de estas operaciones, por lo general después de cada enunciado SQL INSERT, DELETE o UPDATE. A esto se le llama el modo IMMEDIATE (inmediato) de verificación de restricción, y es el modo por defecto. Sin embargo, hay ocasiones cuando una transacción o aplicación involucra varios cambios y la base de datos será temporalmente inválida mientras los cambios están en progreso, mientras algunos aunque no todos los cambios ya se realizaron. Por ejemplo, suponga que tiene una tabla `Department` en el ejemplo University con una columna `chairPerson` en la que se menciona el `facId` del jefe de departamento, y se usa una especificación NOT NULL para dicho campo y se le convierte en clave externa al escribir:

```
CREATE TABLE Department (
   deptName       CHAR(20),
   chairPerson    CHAR(20) NOT NULL,
   CONSTRAINT Department_deptName_pk PRIMARY KEY(deptName),
   CONSTRAINT Department_facId_fk FOREIGN KEY REFERENCES Faculty (facId));
```

Por la definición de tabla `Faculty` en la figura 6.2, ya se tiene un NOT NULL para el departamento, pero ahora se podría convertir a `department` en una clave externa con la nueva tabla `Department` como la tabla de origen, al escribir:

```
ALTER TABLE Faculty ADD CONSTRAINT Faculty_department_fk FOREIGN KEY
department REFERENCES Department(deptName);
```

¿Qué ocurrirá cuando intente crear un nuevo departamento y agregar al personal docente para dicho departamento a la tabla `Faculty`? No se puede insertar el registro del departamento a menos que ya se tenga el registro del jefe del mismo en la tabla `Faculty`, pero no se puede insertar dicho registro a menos que el registro del departamento ya esté ahí, así que cada enunciado SQL INSERT fallaría. Para tales situaciones, SQL permite diferir la verificación hasta el final de toda la transacción, usando enunciados como

```
SET CONSTRAINT Department_facId_fk DEFERRED;
```

o

```
SET CONSTRAINT Faculty_department_fk DEFERRED;
```

Las restricciones se pueden habilitar o deshabilitar para permitir el éxito de tales transacciones, usando un enunciado como:

```
DISABLE CONSTRAINT Department_facId_fk;
```

Al final de la transacción escribiría

```
ENABLE CONSTRAINT Department_facId_fk;
```

para permitir de nuevo el fortalecimiento. Aunque no se recomienda, SQL permite escribir

```
DISABLE ALL CONSTRAINTS;
```

que suspende toda la verificación de integridad hasta encontrar un comando

```
ENABLE ALL CONSTRAINTS;
```

correspondiente. Estos enunciados se pueden usar tanto interactivamente como dentro de las aplicaciones.

6.5.2 Disparadores (triggers) SQL

Como las restricciones, los **disparadores** (triggers) permiten al DBMS monitorear la base de datos. Sin embargo, son más flexibles que las restricciones, se aplican a un rango más amplio de situaciones y permiten una mayor variedad de acciones. Un disparador consiste en tres partes:

- Un **evento**, que normalmente es algún cambio hecho a la base de datos
- Una **condición**, que es un predicado lógico que evalúa a verdadero o falso
- Una **acción**, que es algún procedimiento que se realiza cuando ocurre el evento y la condición evalúa a verdadero, también llamado disparar el disparador

Un disparador tiene acceso a los datos insertados, borrados o actualizados que causaron su disparo (es decir, a activarse o elevarse), y los valores de datos se pueden usar en el código tanto para la condición como para la acción. El prefijo :`OLD` se usa para hacer referencia a los valores en una tupla recién borrada o a los valores sustituidos en una actualización. El prefijo :`NEW` se usa para referirse a los valores en una tupla recién insertada o a los nuevos valores en una actualización. Los disparadores se pueden disparar o antes o después de la ejecución de la operación insertar, borrar o actualizar. También puede especificar si el disparador se dispara sólo una vez por cada enunciado de disparo, o por cada fila que cambie por el enunciado (recuerde que, por ejemplo, un enunciado de actualización puede cambiar muchas filas). En Oracle se especifica el comando SQL (INSERT, DELETE o UPDATE) que es el evento; la tabla involucrada; el nivel del disparador (ROW, fila, o STATEMENT, enunciado); la temporización (BEFORE, antes, o AFTER, después), y la acción a realizar, que se puede escribir como uno o más comandos SQL en PL/SQL. La sección 6.7 discute PL/SQL con más detalle. La sintaxis de disparador Oracle tiene la forma:

```
CREATE OR REPLACE TRIGGER nombre_disparador
  [BEFORE/AFTER] [INSERT/UPDATE/DELETE] ON nombre_tabla
  [FOR EACH ROW] [WHEN condición]
  BEGIN
      cuerpo del disparador
  END;
```

Por ejemplo, agregue a la tabla `Class` dos atributos adicionales, `currentEnroll`, que muestra el número de estudiantes actualmente inscritos en cada clase, y `maxEnroll`, que es el número máximo de estudiantes con permiso para inscribirse. La nueva tabla, `RevClass`, se muestra en la figura 6.5, junto con una nueva versión de la tabla `Enroll`, `RevEnroll`. Puesto que `currentEnroll` es un atributo derivado, dependiente de la tabla `RevEnroll`, su valor se debe actualizar cuando existan cambios relevantes hechos a `RevEnroll`. Los cambios que afectan a `currentEnroll` son:

1. Un estudiante que se inscribe en una clase
2. Un estudiante que da de baja una clase
3. Un estudiante que cambia de una clase a otra

En una base de datos activa debe haber disparadores para cada uno de estos cambios. Para el cambio (1) debe incrementar el valor de `currentEnroll` por uno. Se puede referir al `classNumber` del nuevo registro `RevEnroll` con el uso del prefijo :NEW. El disparador correspondiente se muestra en la figura 6.5(b). Note que no se usó WHEN porque siempre se quiere hacer este cambio después de insertar un nuevo registro de inscripción, así que no se necesitó condición alguna. Para el cambio (2) se necesita disminuir el valor de `currentEnroll` por uno, y se usa el prefijo: OLD para referirse al registro `RevEnroll` a borrar. El disparador se muestra en la figura 6.5(c). El cambio (3) se podría tratar como una baja seguida por una inscripción, pero en vez de ello se escribirá un disparador para una actualización a fin de demostrar una acción con dos partes, como se muestra en la figura 6.5(d).

Aunque estos disparadores son suficientes si hay espacio en una clase, también se necesita considerar lo que ocurriría si la clase ya tiene inscripción completa. Debió examinar el valor de `maxEnroll` antes de permitir a un estudiante su inscripción en la clase, así que necesita verificar dicho valor antes de hacer los cambios (1) o (3). Suponga que si un cambio causaría que una clase tuviera inscripción en exceso, se llamaría un procedimiento llamado

FIGURA 6.5(a)

Tablas para disparadores

RevClass

classNumber	facId	schedule	room	currentEnroll	maxEnroll
ART103A	F101	MWF9	H221	3	25
CSC201A	F105	TuThF10	M110	2	20
CSC203A	F105	MThF12	M110	0	20
HST205A	F115	MWF11	H221	1	35
MTH101B	F110	MTuTh9	H225	1	25
MTH103C	F110	MWF11	H225	2	25

RevEnroll

stuId	classNumber	grade
S1001	ART103A	A
S1001	HST205A	C
S1002	ART103A	D
S1002	CSC201A	F
S1002	MTH103C	B
S1010	ART103A	
S1010	MTH103C	
S1020	CSC201A	B
S1020	MTH101B	A

FIGURA 6.5(b)

Disparador para inscripción de estudiante en una clase

```
CREATE TRIGGER ADDENROLL
AFTER INSERT ON RevEnroll
FOR EACH ROW
BEGIN
    UPDATE RevClass
    SET currentEnroll = currentEnroll +1
    WHERE RevClass.classNumber = :NEW.classNumber;
END;
```

FIGURA 6.5(c)

Disparador para un estudiante que se da de baja en una clase

```
CREATE TRIGGER DROPENROLL
AFTER DELETE ON RevEnroll
FOR EACH ROW
BEGIN
    UPDATE RevClass
    SET currentEnroll = currentEnroll −1
    WHERE RevClass.classNumber = OLD.classNumber;
END;
```

```
CREATE TRIGGER SWITCHENROLL
AFTER UPDATE OF classNumnber ON RevEnroll
FOR EACH ROW
    BEGIN
        UPDATE RevClass
        SET currentEnroll = currentEnroll + 1
        WHERE RevClass.classNumber = :NEW.classNumber;
        UPDATE RevClass
        SET currentEnroll = currentEnroll –1
        WHERE RevClass.classNumber = :OLD.classNumber;
    END;
```

FIGURA 6.5(d)
Disparador para estudiante que cambia de clases

```
CREATE TRIGGER ENROLL_REQUEST
BEFORE INSERT OR UPDATE OF classNumber ON REVenroll
FOR EACH ROW
DECLARE
    numStu number;
    maxStu number;
BEGIN
    select   maxEnroll into maxStu
    from     RevClass
    where   RevClass.classNumber = :NEW.classNumber;

    select  currentEnroll +1 into numStu
    from     RevClass
    where   RevClass.classNumber = :NEW.classNumber;

    if numStu > maxStu
    RequestClosedCoursePermission(:NEW.stuId, :NEW.classNumber, RevClass.currentEnroll, RevClass.
   maxEnroll);
    end if;
END;
```

FIGURA 6.5(e)
Disparador para verificar la inscripción en exceso antes de inscribir a un estudiante

RequestClosedCoursePermission (solicitud de permiso de curso cerrada) que tomaría como parámetros la ID del estudiante, el número de clase, la inscripción actual y la inscripción máxima. La acción se debe tomar antes de hacer el cambio. El disparador se muestra en la figura 6.5(e).

Los disparadores se habilitan automáticamente cuando se crean. Se pueden deshabilitar con el enunciado:

ALTER TRIGGER *nombre_disparador* DISABLE;

Después de deshabilitar se pueden habilitar de nuevo mediante el enunciado:

ALTER TRIGGER *nombre_disparador* ENABLE;

Se pueden eliminar mediante el enunciado:

DROP TRIGGER *nombre_disparador*;

Oracle proporciona una forma INSTEAD OF (en lugar de) que es especialmente útil cuando un usuario intenta actualizar la base de datos a través de una vista. Esta forma especifica

una acción a realizar en lugar de la inserción, borrado o actualización que el usuario solicita. Se discutirá en la sección 6.8. Los disparadores también se pueden usar a fin de proporcionar una vía de auditoría para una tabla, registrar todos los cambios, el momento cuando se hicieron y la identidad del usuario que los hizo, una aplicación que se discutirá en la sección 9.6.

6.6 Uso de los enunciados COMMIT y ROLLBACK

Cualquier cambio hecho a una base de datos usando comandos SQL no es permanente hasta que el usuario escribe un enunciado COMMIT (compromiso). Como se discute en el capítulo 10, una transacción SQL termina cuando se encuentra o un enunciado COMMIT o un enunciado ROLLBACK (retrocesión). El COMMIT hace permanentes los cambios realizados desde el comienzo de la transacción actual, que es o el comienzo de la sesión o el momento desde el último COMMIT o ROLLBACK. El ROLLBACK deshace los cambios realizados por la transacción actual. Es aconsejable escribir COMMIT con frecuencia para guardar los cambios mientras trabaja.

6.7 Programación SQL

Para los enunciados SQL interactivos mostrados hasta el momento, se supone que había una interfaz de usuario que proporcionaba un entorno para aceptar, interpretar y ejecutar directamente los comandos SQL. Un ejemplo es la facilidad SQLPlus de Oracle. Aunque algunos usuarios pueden interactuar con la base de datos de esta forma, la mayoría de los accesos a la base de datos es a través de programas.

6.7.1 SQL incrustado (embedded)

Una forma en que se puede usar SQL es incrustarlo en programas escritos en un lenguaje de programación de propósito general como C, C++, Java, COBOL, Ada, Fortran, Pascal, PL/1 o M, a los que se les conoce como **lenguaje huésped**. Cualquier comando SQL interactivo como los discutidos, se puede usar en un programa de aplicación, como modificaciones menores. Los programadores escriben el código fuente usando tanto enunciados de lenguaje huésped, que proporcionan las estructuras de control, como enunciados SQL, que gestionan el acceso a la base de datos. Los enunciados SQL ejecutables están precedidos por un prefijo como la palabra clave EXEC SQL y terminan con un terminador como un punto y coma. Un enunciado SQL ejecutable puede aparecer en cualquier parte que pueda aparecer un enunciado en lenguaje huésped ejecutable. El DBMS proporciona un precompilador, que barre todo el programa y extrae los enunciados SQL, identificados por el prefijo. El SQL se compila por separado en algún tipo de módulo de acceso y los enunciados SQL se sustituyen, por lo general mediante llamadas simples de función en el lenguaje huésped. El programa resultante en lenguaje huésped puede entonces compilarse como siempre. La figura 6.6 ilustra este proceso.

El intercambio de datos entre el programa de aplicación y la base de datos se logra a través de las variables de programa en lenguaje huésped, para cuyos atributos de registros de base de datos se proporcionan valores o de los cuales reciben sus valores. Estas **variables compartidas** se declaran dentro del programa en una sección de declaración SQL como la siguiente:

```
EXEC SQL BEGIN DECLARE SECTION;
char stuNumber[5];
char stuLastName[15];
char stuFirstName[12];
```

```
char stuMajor[10];
int stuCredits;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
```

Las primeras cinco variables de programa declaradas aquí están diseñadas para coincidir con los atributos de la tabla `Student`. Se pueden usar para recibir valores de las tuplas `Student`, o suministrar valores a las tuplas. Puesto que los tipos de datos del lenguaje huésped pueden no coincidir exactamente con las de los atributos de la base de datos, se puede usar un tipo de operación cast (molde) para pasar valores apropiados entre variables de programa y atributos de base de datos. SQL:1999 proporciona vínculos de lenguaje que especifican la correspondencia entre tipos de datos SQL y los tipos de datos de cada lenguaje huésped.

La última variable declarada, SQLSTATE, es un arreglo carácter que se usa para comunicar condiciones de error al programa. Cuando se llama una función de biblioteca SQL se coloca un valor en SQLSTATE que indica cuál condición de error pudo haber ocurrido cuando se accedió a la base de datos. Un valor de '00000' indica no error, mientras que un valor '02000' indica que el enunciado SQL se ejecutó correctamente pero no se encontró tupla para una consulta. El valor de SQLSTATE se puede probar en enunciados de control de lenguaje huésped. Por ejemplo, se puede realizar algún procesamiento si el valor es '00000', o se puede escribir una iteración para leer un registro hasta que regrese el valor de '02000'.

Los enunciados SELECT de SQL incrustado que operan en una sola fila de una tabla son muy similares a los enunciados interactivos que se vieron anteriormente. Por ejemplo, un enunciado SELECT que recupera una sola tupla se modifica para el caso incrustado al identificar las variables compartidas en una línea INTO. Note que, cuando se hace referencia a variables compartidas dentro de un enunciado SQL, están precedidas por dos puntos para distinguirlas de los atributos, que pueden o no tener los mismos nombres. El siguiente ejemplo comienza con un enunciado en lenguaje huésped que asigna un valor a la variable huésped `stuNumber`. Luego usa un enunciado SQL SELECT para recuperar una tupla de

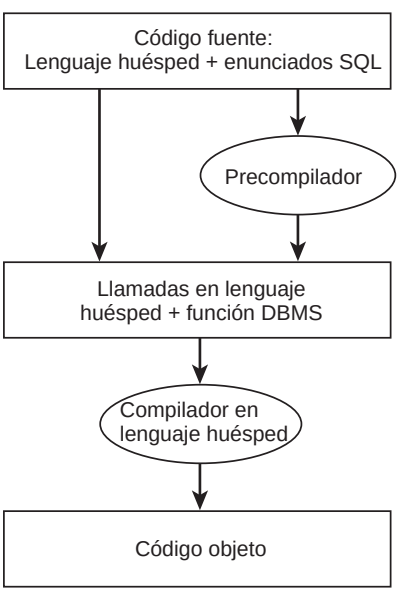


FIGURA 6.6
Procesamiento de programas SQL incrustados

la tabla `Student` cuyo `stuId` coincida con la variable compartida `stuNumber` (referida como :`stuNumber` en el enunciado SQL). Pone los valores de atributo de tupla en cuatro variables compartidas que se declararon previamente:

```
stuNumber = 'S1001';
EXEC SQL  SELECT  Student.lastName, Student.firstName, Student.major,
Student.credits
   INTO :stuLastName, :stuFirstName, :stuMajor, :stuCredits
   FROM Student
   WHERE Student.stuId = :stuNumber;
```

Este segmento se debe seguir mediante una verificación en lenguaje huésped del valor de SQLSTATE.

Para insertar una tupla de base de datos, puede asignar valores a variables compartidas en el lenguaje huésped y luego usar un enunciado SQL INSERT. Los valores de atributo se toman de las variables huésped. Por ejemplo, se podría escribir:

```
stuNumber = 'S1050';
stuLastName = 'Lee';
stuFirstName = 'Daphne';
stuMajor = 'English';
stuCredits = 0;
EXEC SQL     INSERT
           INTO Student (stuId, lastName, firstName, major, credits)
           VALUES(:stuNumber,:stuLastName, :stuFirstName,:stuMajor,
:stuCredits);
```

Puede borrar cualquier número de tuplas que pueda identificar usando el valor de una variable huésped:

```
stuNumber = 'S1015';
EXEC SQL    DELETE
        FROM Student
        WHERE stuId = :stuNumber;
```

También puede actualizar cualquier número de tuplas en una forma similar:

```
stuMajor = 'History';
EXEC SQL UPDATE Student
        SET CREDITS = CREDITS + 3
        WHERE major = :stuMajor;
```

En lugar de verificar el valor de SQLSTATE después de cada enunciado SQL, puede usar el enunciado manipulador de error WHENEVER (siempre que) que tiene la forma:

```
EXEC SQL WHENEVER [SQLERROR/NOT FOUND] [CONTINUE/GO TO enunciado];
```

Como se mencionó antes, una condición NOT FOUND significa que el valor de SQLSTATE es 02000, mientras que la condición SQLERROR es cualquier excepción. Un WHENEVER permanece en efecto durante todo el programa para la condición especificada, a menos que aparezca un nuevo WHENEVER para la misma condición, que pasa por encima de la primera.

Un problema especial llamado **desajuste de impedancia** ocurre cuando un enunciado SELECT regresa más de una tupla (un multiconjunto). Aunque el modelo relacional usa conjuntos, un lenguaje huésped generalmente es capaz de manipular sólo un registro a la vez en lugar de todo un conjunto de registros. Es necesario un dispositivo llamado **cursor**, un puntero simbólico que apunta a una fila de una tabla o multiconjunto a la vez, proporciona a la aplicación acceso a dicha fila, y luego se mueve hacia la siguiente fila. Para una consulta SQL que regresa un multiconjunto, el cursor se puede usar para avanzar a través de

los resultados de la consulta, lo que permite la dotación de valores al lenguaje huésped tupla por tupla. Un cursor se crea y coloca de modo que puede apuntar a una nueva fila en la tabla o en el multiconjunto a la vez. La forma para declarar un cursor es:

EXEC SQL DECLARE *nombre_cursor* [INSENSITIVE] [SCROLL] CURSOR FOR *consulta*
[FOR {READ ONLY | UPDATE OF *nombre_atributos*}];

Por ejemplo, para crear un cursor que más tarde se usará para ir a través de los registros de estudiantes CSC que sólo se planea recuperar, se escribiría:

```
EXEC SQL DECLARE CSCstuCursor CURSOR FOR
   SELECT stuId, lastName, firstName, major, credits
   FROM student
   WHERE major='CSC';
```

Note que ésta es una declaración, no un enunciado ejecutable. La consulta SQL no se ejecuta todavía. Después de declarar el cursor se escribe un enunciado para abrirlo. Éste ejecuta la consulta de modo que se crean los resultados multiconjunto. Abrir el cursor también lo coloca justo antes de la primera tupla del conjunto de resultados. Para este ejemplo, escriba:

```
EXEC SQL OPEN CSCstuCursor;
```

Para recuperar la primera fila de los resultados se usa entonces el comando FETCH (lectura de instrucción) que tiene la forma

```
EXEC SQL FETCH cursorname INTO hostvariables;
```

como en

```
EXEC SQL FETCH CSCstuCursor INTO :stuNumber,:stuLastName, :stuFirstName,:stuMajor,
   :stuCredits
```

El enunciado FETCH avanza el cursor y asigna los valores de los atributos mencionados en el enunciado SELECT a las correspondientes variables compartidas mencionadas en la línea INTO. Una iteración controlada por el valor de SQLSTATE (por ejemplo, WHILE (SQLSTATE=00000)) se debe crear en el lenguaje huésped de modo que se acceda a filas adicionales. La iteración también contiene enunciados en lenguaje huésped para hacer cualquier procesamiento que sea necesario. Después de recuperar todos los datos, se sale de la iteración y se cierra el cursor en un enunciado como:

```
EXEC SQL CLOSE CSCstuCursor;
```

Si planea actualizar múltiples filas usando el cursor, inicialmente debe declararlo como actualizable, con el uso de una declaración más completa como:

```
EXEC  SQL DECLARE stuCreditsCursor CURSOR FOR
      SELECT stuId, credits
      FROM Student
FOR UPDATE OF credits;
```

Se debe nombrar, tanto en el enunciado SELECT como en la lista de actualización de atributos, cualquier atributo que se planee actualizar usando el cursor. Una vez que el cursor se abra y active, puede actualizar la tupla donde se coloca el cursor, llamado el **actual** (current) del cursor, al escribir un comando como:

```
EXEC SQL UPDATE Student
SET credits = credits +3
WHERE CURRENT OF stuCreditsCursor;
```

De igual modo, la tupla actual se puede borrar mediante un enunciado como:

```
EXEC SQL DELETE FROM Student
WHERE CURRENT OF stuCreditCursor;
```

Es posible que los usuarios quieran poder especificar el tipo de acceso que necesitan en el tiempo de corrido en lugar de en una forma estática usando código compilado del tipo recién descrito. Por ejemplo, es posible que quiera un frente gráfico donde el usuario pueda ingresar una consulta que se pueda usar para generar enunciados SQL que se ejecuten dinámicamente, como el SQL interactivo descrito antes. Además de la versión de SQL apenas discutida, que se clasifica como estática, existe un **SQL dinámico**, que permite especificar el tipo de acceso a base de datos en el tiempo de corrida en lugar de en el momento de compilar. Por ejemplo, al usuario se le puede conminar a ingresar un comando SQL que luego se almacene como una cadena en lenguaje huésped. El comando SQL PREPARE (preparar) dice al sistema de gestión de la base de datos que analice sintéticamente (parse) y compile la cadena como un comando SQL y asigne el código ejecutable resultante a una variable SQL nominada. Luego se usa un comando EXECUTE para correr el código. Por ejemplo, en el siguiente segmento la variable en lenguaje huésped `userString` se asigna a un comando SQL de actualización, el código correspondiente se prepara y liga al identificador SQL `userCommand`, y entonces se ejecuta el código.

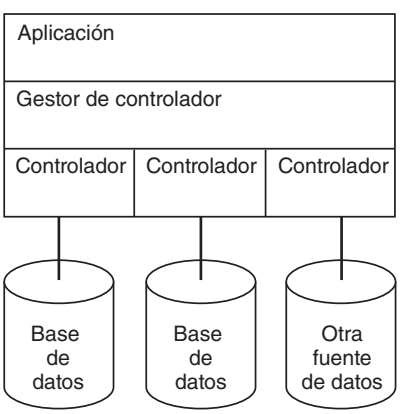
```
char userString[ ]='UPDATE Student SET credits = 36 WHERE stuId= S1050';
EXEC SQL PREPARE userCommand FROM :userString;
EXEC SQL EXECUTE userCommand;
```

6.7.2 API, ODBC y JDBC

Para SQL incrustado, un precompilador suministrado por el DBMS compila el código SQL, y lo sustituye con llamadas de función en el lenguaje huésped. Un enfoque más flexible es que el DBMS proporcione una **Interfaz de Programas de Aplicación, API,** que incluye un conjunto de funciones de biblioteca estándar para procesamiento de bases de datos. Entonces la aplicación puede llamar las funciones de biblioteca, que se escriben en un lenguaje de propósito general. Los programadores usan estas funciones de biblioteca esencialmente en la misma forma como usan la biblioteca del lenguaje en sí. Las funciones permiten a la aplicación realizar operaciones estándar como conectar a la base de datos, ejecutar comandos SQL, presentar tuplas una a la vez, etc. Sin embargo, la aplicación todavía tendría que usar el precompilador para un DBMS particular y ligarse a la librería API para dicho DBMS, de modo que el mismo programa podría no usarse con otro DBMS.

La **conectividad de base de datos abierta (ODBC)** y la **conectividad de base de datos JAVA (JDBC)** proporcionan formas estándar de integrar código SQL y lenguajes de propósito general al proporcionar una interfaz común. Esta estandarización permite que las aplicaciones accedan a múltiples bases de datos usando diferentes DBMS.

FIGURA 6.7
Arquitectura ODBC/JDBC



El estándar proporciona un alto grado de flexibilidad, lo que permite el desarrollo de aplicaciones cliente-servidor que funcionan con una variedad de DBMS, en lugar de estar limitadas a un proveedor API particular. La mayoría de los proveedores ofrecen controladores ODBC o JDBC que se conforman al estándar. Una aplicación que use una de estas interfaces estándar puede usar el mismo código para acceder a diferentes bases de datos sin recompilación. La arquitectura ODBC/JDBC requiere cuatro componentes: la aplicación, gestor de controlador, controlador y fuente de datos (normalmente una base de datos), como se ilustra en la figura 6.7. La aplicación inicia la conexión con la base de datos, envía los datos solicitados como enunciados SQL al DBMS, recupera los resultados, realiza procesamiento y termina la conexión, todo usando el API estándar. Un gestor de controlador carga y descarga controladores a petición de la aplicación, y pasa las llamadas ODBC o JDBC al controlador seleccionado. El controlador de base de datos vincula la aplicación a la fuente de datos, traduce las llamadas ODBC o JDBC a llamadas específicas de DBMS y manipula la traducción de datos necesaria debido a cualquier diferencia entre el lenguaje de datos DBMS y el estándar ODBC/JDBC, y las diferencias de manipulación de error que surgen entre la fuente de datos y el estándar. La fuente de datos es la base de datos (u otra fuente, como una hoja de cálculo) a la que se accede, junto con su entorno, que consiste en sus DBMS y plataforma. Existen distintos niveles de conformidad definidos para controladores ODBC y JDBC, dependiendo del tipo de relación entre la aplicación y la base de datos.

6.7.3 PSM SQL

La mayoría de los sistemas de gestión de bases de datos incluyen una extensión del mismo SQL, llamada **módulos de almacenamiento permanente (PSM)**, para permitir a los usuarios escribir procedimientos almacenados, llamados **rutinas internas**, dentro del espacio de proceso de la base de datos, en lugar de externamente. Estas facilidades se usan para escribir rutinas SQL que se puedan guardar con el esquema de la base de datos y se invoquen cuando se necesite (en contraste, los programas escritos en un lenguaje huésped se conocen como rutinas externas). El PL/SQL de Oracle, al que se puede acceder desde dentro del entorno SQLPlus, es este tipo de facilidad. El estándar SQL/PSM está diseñado para proporcionar facilidades de lenguaje de programación completo, incluidas declaraciones, estructuras de control y enunciados de asignación.

Los módulos SQL/PSM incluyen funciones, procedimientos y relaciones temporales. Para declarar un procedimiento, se escribe:

```
CREATE PROCEDURE nombre_procedimiento (lista_parámetros)
    declaraciones de variables locales
    código de procedimiento
```

Cada parámetro en la lista de parámetros tiene tres ítems: modo, nombre y tipo de datos. El modo puede ser IN, OUT o INOUT, dependiendo de si es un parámetro de entrada, un parámetro de salida o ambos. También se deben proporcionar el nombre y tipo de datos del parámetro. A continuación se tienen declaraciones de variables locales, si hay alguna, y el código para el procedimiento, que puede contener enunciados SQL del tipo visto anteriormente para SQL incrustado, más enunciados de asignación, enunciados de control y manipulación de error.

Una función tiene una declaración similar, del modo siguiente:

```
CREATE FUNCTION nombre_función (lista_parámetros)
    RETURNS tipo de datos SQL
declaraciones de variables locales
código de función (debe incluir un enunciado RETURN)
```

Las funciones sólo aceptan parámetros con modo IN, para evitar efectos colaterales. El único valor regresado debe ser el especificado en el enunciado RETURN.

Las declaraciones tienen esta forma:

```
DECLARE identificador de tipo de datos;
```

como en,

```
DECLARE     status                VARCHAR2;
DECLARE     number_of_courses     NUMBER;
```

El enunciado de asignación SQL, SET, permite que el valor de una constante o una expresión se asigne a una variable. Por ejemplo, podría escribir:

```
SET status = 'Freshman'; //However, Oracle uses := for assignment
SET number_of_courses = credits/3;
```

Las ramificaciones (branches) tienen la forma:

```
IF (condición) THEN enunciados;
   ELSEIF (condición) enunciados;
   . . .
   ELSEIF (condición) enunciados;
   ELSE enunciados;
END IF;
```

por ejemplo,

```
IF (Student.credits <30) THEN
   SET status = 'Freshman';
   ELSEIF (Student.credits <60) THEN
      SET status = 'Sophomore';
   ELSEIF (Student.credits <90) THEN
      SET status = 'Junior';
   ELSE SET status = 'Senior';
END IF;
```

El enunciado CASE se puede usar para selección con base en el valor de una variable o expresión:

```
CASE selector
   WHEN valor_1          THEN enunciados;
   WHEN valor_2          THEN enunciados;
   . . .
END CASE;
```

La repetición se controla mediante estructuras LOOP... ENDLOOP, WHILE... DO... END WHILE, REPEAT... UNTIL... END REPEAT, y FOR... DO... END FOR. Puede usar cursores como se hizo para SQL incrustado. Por ejemplo, podría escribir:

```
...
DECLARE CSCstuCursor CURSOR FOR
   SELECT stuId, lastName, firstName, major, credits
   FROM student
   WHERE major= 'CSC';
OPEN CSCstuCursor;
WHILE (SQLCODE = '00000') DO
   FETCH CSCstuCursor INTO stuNumber,stuLastName,stuFirstName,
   stuMajor,stuCredits;
      //statements to process these values
END WHILE;
CLOSE CSCstuCursor;
```

El lenguaje también proporciona manejadores de excepción predefinidos y también permite al usuario crear excepciones definidas por el usuario.

Una vez creado un procedimiento se puede ejecutar mediante este comando:

```
EXECUTE nombre_procedimiento(lista_parámetros_real);
```

Como con la mayoría de los lenguajes, la lista de parámetros real consiste en los valores o variables a pasar al procedimiento, en oposición a la lista de parámetros formal que aparece en la declaración del procedimiento.

Una función se invoca mediante su nombre, por lo general en un enunciado de asignación. Por ejemplo:

```
SET newVal = MyFunction (val1, val2);
```

6.8 Creación y uso de vistas

Las vistas son herramientas importantes a fin de proporcionar a los usuarios un entorno simple y personalizado, y para ocultar datos. Como se explicó en la sección 6.2, una vista relacional no corresponde exactamente con la vista externa general, sino que es una tabla virtual derivada de una o más tablas base subyacentes. No existe en almacenamiento en el sentido como lo hacen las tablas base, sino que se crean mediante selección de filas y columnas específicas de las tablas base, y posiblemente al realizar operaciones sobre ellas. La vista se produce dinámicamente conforme el usuario trabaja con ella. Para integrar una vista, el ABD decide a cuáles atributos necesita acceso el usuario, determina cuáles tablas base los contienen y construye una o más vistas para mostrar en forma de tabla los valores que el usuario debe ver. Las vistas permiten fácilmente la creación de un modelo externo dinámico. Las razones para proporcionar vistas en lugar de permitir a los usuarios trabajar con las tablas base son las siguientes:

- Las vistas permiten a diferentes usuarios ver los datos en distintas formas, y permiten un modelo externo que difiere del modelo lógico.
- El mecanismo de vista proporciona un dispositivo sencillo de control de autorización, creado fácilmente y reforzado en forma automática por el sistema. Los usuarios de vista no están al tanto de, y no pueden acceder, a ciertos ítems de datos.
- Las vistas pueden liberar a los usuarios de complicadas operaciones DML, especialmente en el caso donde las vistas involucran combinaciones. El usuario escribe un enunciado SELECT simple usando la vista como la tabla nominada y el sistema se encarga de los detalles de las correspondientes operaciones más complicadas sobre las tablas base para soportar la vista.
- Si la base de datos se reestructura en el nivel lógico, la vista se puede utilizar para mantener constante el modelo del usuario. Por ejemplo, si la tabla se divide mediante proyección y la clave primaria aparece en cada una de las nuevas tablas resultantes, la tabla original siempre se puede reconstruir cuando se necesite al definir una vista que es la combinación de las nuevas tablas.

La siguiente es la forma más común del comando usado para crear una vista:

```
CREATE VIEW nombre_vista
    [(nombre_vista [,nombre_vista] . . .)]
    AS  SELECT    nombre_columna [,nombre_columna] . . .
        FROM      nombre_tabla_base [,nombre_tabla_base] . . .
        WHERE     condición;
```

El nombre de la vista se elige usando las mismas reglas que el nombre de tabla y debe ser único dentro de la base de datos. Los nombres de columna en la vista pueden ser diferentes de los correspondientes nombres de columna en las tablas base, pero deben obedecer las mismas reglas de construcción. Si se elige hacerlos iguales es necesario no especificarlos dos veces, de modo que se deja la línea *nombre_col_vista*. En la línea AS SELECT se mencionan los nombres de las columnas de las tablas base subyacentes que se quieren incluir en la vista. El orden de estos nombres debe corresponder exactamente con los *nombre_col_vista*, si los mismos se especifican. Sin embargo, las columnas elegidas de las tablas base pueden reordenarse en cualquier forma deseada en la vista. Como en el usual SELECT... FROM... WHERE, la condición es un predicado lógico que expresa alguna restricción sobre los registros a incluir. Una forma más general de CREATE VIEW usa cualquier subconsulta válida en lugar del SELECT descrito.

- Ejemplo 1. Elección de un subconjunto vertical y horizontal de una tabla

  Suponga que un usuario necesita ver los ID y nombres de todos quienes tienen especialidad history. Puede crear una vista para este usuario del modo siguiente:

```
CREATE VIEW HISTMAJ (last, first, StudentId)
  AS SELECT    lastName, firstName,stuId
     FROM      Student
     WHERE     major = 'History';
```

Aquí se renombraron las columnas de la tabla base. El usuario de esta vista no necesita saber los nombres de columna reales.

- Ejemplo 2. Elección de un subconjunto vertical de una tabla

Si quisiera una tabla de todos los cursos con sus horarios y salones podría crearla del modo siguiente:

```
CREATE VIEW      ClassLoc
AS SELECT        classNumber, schedule, room
FROM             Class;
```

Note que aquí no se necesitó una condición, pues se querían estas partes de todos los registros `Class`. Esta vez se conservan los nombres de las columnas como aparecen en la tabla base.

- Ejemplo 3. Una vista usando dos tablas

Suponga que un usuario necesita una tabla que contenga las ID y nombres de todos los estudiantes en el curso CSC101. La tabla virtual se puede crear al elegir registros en `Enroll` que tengan `classNumber` de CSC101, coincidir el `stuId` de dichos registros con el `stuId` de los registros `Student` y tomar los correspondientes `lastName` y `firstName` de `Student`. Esto se podría expresar como una combinación o una subconsulta.

```
CREATE VIEW ClassList
   AS  SELECT    Student.stuId,lastName,
               firstName
       FROM      Enroll,Student
       WHERE     classNumber = 'CSC101'
               AND Enroll.stuId =
               Student.stuId;
```

- Ejemplo 4. Una vista de una vista

Es posible definir una vista derivada de una vista. Por ejemplo, puede pedir un subconjunto de la tabla (virtual) `ClassLoc` al escribir:

```
CREATE VIEW ClassLoc2
  AS SELECT     classNumber, room
     FROM       ClassLoc;
```

- Ejemplo 5. Una vista usando una función

  En el enunciado SELECT en la línea AS puede incluir funciones internas y opciones GROUP BY. Por ejemplo, si quiere una vista de `Enroll` que proporcione `classNumber` y el número de estudiantes inscritos en cada clase, escriba:

```
CREATE VIEW ClassCount (classNumber, TotCount)
  AS SELECT     classNumber, COUNT(*)
     FROM       Enroll
                GROUP BY classNumber;
```

Note que se debe proporcionar un nombre para la segunda columna de la vista, pues no hay alguno disponible de la tabla base.

- Ejemplo 6. Operaciones sobre vistas

Una vez creada una vista, el usuario puede escribir enunciados SELECT para recuperar datos a través de la vista. El sistema se encarga de mapear los nombres de usuario a los nombres de tabla base y nombres de columna subyacentes, y realiza cualquier función que se requiera para producir el resultado en la forma que el usuario espera. Los usuarios pueden escribir consultas SQL que se refieran a combinaciones, ordenamiento, agrupamiento, funciones internas, etc., de vistas tal como si estuviese operando sobre las tablas base. Dado que la operación SELECT no cambia las tablas base subyacentes, no hay restricción para usarlas con las vistas. El siguiente es un ejemplo de una operación SELECT sobre la vista `ClassLoc`:

```
SELECT     *
FROM       ClassLoc
WHERE      room LIKE 'H%';
```

INSERT, DELETE y UPDATE presentan ciertos problemas con las vistas. Por ejemplo, suponga que se tiene una vista de registros de estudiante como:

```
StudentVwl(lastName, firstName, major, credits)
```

Si tuviese permiso de ingresar registros, cualquier registro creado mediante esta vista en realidad serían registros `Student`, pero no contendrían `stuId`, que es la clave de la tabla `Student`. Dado que `stuId` tendría la restricción NOT NULL deberá rechazar cualquier registro sin este campo. Sin embargo, si tiene la siguiente vista:

```
StudentVw2(stuId,lastName, firstName,credits)
```

no debe tener problema para insertar registros, pues insertaría registros `Student` con un campo `major` nulo, lo que está permitido. Podría lograr esto al escribir:

```
INSERT
INTO    StudentVw2
VALUES  ('S1040' 'Levine', 'Adam', 30);
```

Sin embargo, el sistema en realidad insertaría el registro en la tabla `Student`. Puede usar un disparador INSTEAD OF para asegurar que esto ocurra.

```
CREATE TRIGGER InsertStuVw2
  INSTEAD OF INSERT ON StudentVw2
  FOR EACH ROW
   BEGIN
       INSERT
       INTO Student
       VALUES (:NEW.stuId, :NEW.lastName,
       :NEW.firstName, NEW. Credits);

   END;
```

Ahora considere insertar registros en la vista `ClassCount`, como se describió anteriormente en el ejemplo 5. Esta vista usó la función COUNT en grupos de registros en la tabla `Enroll`. Obviamente, esta tabla tenía la intención de ser un resumen dinámico de la tabla `Enroll`, en lugar de ser un subconjunto fila y columna de dicha tabla. No tendría sentido permitir la inserción de nuevos registros `ClassCount`, pues éstos no corresponden a filas o columnas individuales de una tabla base.

Los problemas identificados para INSERT se aplican con cambios menores también a UPDATE y DELETE. Como regla general, estas tres operaciones se pueden realizar sobre vistas que consisten en filas y columnas reales de tablas base subyacentes, siempre que la clave primaria se incluya en la vista y no se violen otras restricciones. Se pueden usar disparadores INSTEAD OF para asegurar que la base de datos actualiza las tablas subyacentes.

6.9 El catálogo del sistema

El **catálogo del sistema** o **diccionario de datos del sistema** se puede considerar como una base de datos de información acerca de las bases de datos. Contiene, en forma de tabla, un resumen de la estructura de cada base de datos como aparece en un momento dado. Siempre que una tabla base, vista, índice, restricción, módulo almacenado u otro ítem de un esquema de base de datos se crea, altera o elimina, el DBMS automáticamente actualiza sus entradas en el catálogo. El sistema también usa el catálogo para verificar autorizaciones y almacenar información para planes de acceso para aplicaciones. Los usuarios pueden consultar el diccionario de datos usando comandos SQL SELECT. Sin embargo, dado que el diccionario de datos se mantiene mediante el DBMS mismo, los comandos SQL UPDATE, INSERT y DELETE no se pueden usar en él.

El **diccionario de datos Oracle** contiene información acerca de todos los objetos de esquema, pero el acceso a él se proporciona mediante tres vistas diferentes, llamadas USER (usuario), ALL (todos) y DBA (administrador de base de datos). En Oracle, a cada usuario se le proporciona automáticamente acceso a todos los objetos que crea. La **vista USER** proporciona a un usuario información acerca de todos los objetos creados por dicho usuario. Los usuarios pueden tener acceso a objetos creados por otros. La **vista ALL** proporciona información acerca de dichos objetos además de los que creó el usuario. La **vista DBA**, que proporciona información acerca de todos los objetos de base de datos, está disponible al administrador de la base de datos. Cada una de las vistas se invoca mediante el uso del término apropiado como un prefijo para el objeto nombrado en la cláusula FROM en una consulta. Por ejemplo, si un usuario quiere una lista de los nombres de todas las tablas que creó, la consulta es:

```
SELECT TABLE_NAME
FROM USER_TABLES;
```

Las consultas se pueden escribir usando los prefijos adecuados (USER_, ALL_, DBA_) en la cláusula FROM, seguido por una de las categorías CATALOG, CONSTRAINTS, CONS_COLUMNS (columnas que tienen restricciones), DICTIONARY, IND_COLUMNS (columnas que tienen índices, INDEXES, OBJECTS, TAB_COLUMNS (columnas de tabla), TRIGGERS, ROLES, PROFILES, SEQUENCES, SOURCE (código fuente para un módulo), SYS_PRIVS, USERS, TABLES, TABLESPACES, VIEWS y otros objetos en el esquema. Para cada una de estas categorías, cada una de las tres vistas (USER, ALL, DBA) tiene varias columnas. Por lo general, puede suponer que cada categoría tiene una columna para el nombre del objeto, que puede usar con el fin de escribir una consulta como:

```
SELECT VIEW_NAME
FROM USER_VIEWS;
```

Para determinar cuáles son todas las columnas disponibles, puede usar el comodín (*) en la cláusula SELECT. Por ejemplo,

```
SELECT*
FROM USER_TAB_COLUMNS;
```

desplegará toda la información registrada acerca de las columnas en las tablas que usted creó. Luego puede usar los nombres de columna de las vistas (por ejemplo, COLUMN_NAME, DATA_TYPE) en una consulta más específica, como

```
SELECT COLUMN_NAME, DATA_TYPE
FROM USER_TAB_COLUMNS
WHERE TABLE_NAME = 'STUDENT';
```

Otra forma de aprender acerca de los objetos es usar el comando DESCRIBE. Una vez que conozca el nombre de un objeto (por ejemplo, una tabla, restricción, columna), que puede obtener mediante uno de los métodos recién ilustrados, puede solicitar una descripción de él. Por ejemplo, puede escribir

```
DESCRIBE STUDENT;
```

para ver lo que se conoce acerca de la tabla `Student`, o

```
DESCRIBE HISTMAJ;
```

para ver lo que se conoce acerca de la vista `HISTMAJ` que se creó en la sección 6.8. Si quiere aprender qué información está disponible acerca de las restricciones en la vista USER, escribiría:

```
DESCRIBE USER_CONSTRAINTS;
```

Entonces puede escribir una consulta con el uso de los nombres de las columnas que se desplegaron, como:

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME
FROM USER_CONSTRAINTS;
```

Para información acerca de los disparadores, el comando

```
SELECT TRIGGER_NAME, TRIGGERING_EVENT, TRIGGER_TYPE
FROM USER_TRIGGERS;
```

proporciona información útil acerca de ellos.

La **base de datos universal DB2** de IBM tiene un catálogo de sistema que también se mantiene en la forma de tablas en un esquema llamado SYSIBM, usualmente con acceso restringido. Dos vistas de las tablas, SYSCAT y SYSSTAT, están disponibles para los usuarios. El **esquema SYSCAT** tiene muchas tablas, todas ellas de sólo lectura. Algunas de las más importantes son las siguientes:

```
TABLES (TABSCHEMA, TABNAME, DEFINER, TYPE, STATUS, COLCOUNT, KEYCOLUMNS,
CHECKCOUNT, ...)
COLUMNS (TABSCHEMA, TABNAME, COLNAME, COLNO, TYPENAME, LENGTH, DEFAULT,
NULLS, ...)
INDEXES (INDSCHEMA, INDNAME, DEFINER, TABSCHEMA, TABNAME, COLNAMES, UNIQUERULE,
COLCOUNT, ...)
TRIGGERS (TRIGSCHEMA, TRIGNAME, DEFINER, TABSCHEMA, TABNAME, TRIGTIME,
TRIGEVENT, ...)
VIEWS (VIEWSCHEMA, VIEWNAME, DEFINER, TEXT ...)
```

Puede escribir consultas para estas tablas usando SQL, como en

```
SELECT TABSCHEMA, TABNAME
FROM TABLES
WHERE DEFINER = 'JONES';
```

que da los nombres de todas las tablas creadas por Jones.

La consulta

```
SELECT *
FROM COLUMNS
WHERE TABNAME = 'STUDENT'
GROUP BY COLNAME;
```

da toda la información disponible acerca de las columnas de la tabla `Student`.

6.10 Resumen del capítulo

Oracle, DB2 de IBM, MySQL, SQL Server y otros sistemas de gestión de bases de datos relacionales usan **SQL**, un DDL y DML relacional estándar. En el nivel lógico, cada relación se representa mediante una **tabla base**. El nivel externo consiste en **vistas**, que se crean a partir de subconjuntos, combinaciones u otras operaciones en las tablas base. Una tabla base puede tener **índices**, uno de los cuales puede ser un índice agrupado, definido en ella. La definición de base de datos dinámica permite que la estructura cambie en cualquier momento.

Los comandos **DDL SQL CREATE TABLE** y **CREATE INDEX** se usan para crear las tablas de datos y sus índices. Están disponibles muchos **tipos de datos** internos y los usuarios también pueden definir nuevos tipos. En el nivel de columna o tabla se pueden especificar **restricciones**. El comando **ALTER TABLE** permite cambios a tablas existentes, como agregar una nueva columna, eliminar una columna, cambiar tipos de datos o cambiar restricciones. El comando **RENAME TABLE** permite al usuario cambiar un nombre de tabla. **DROP TABLE** y **DROP INDEX** remueven tablas e índices, junto con todos los datos en ellos, de la base de datos.

Los comandos **DML** son **SELECT**, **UPDATE**, **INSERT** y **DELETE**. El comando SELECT tiene varias formas y realiza el equivalente de las operaciones SELECT, PROJECT y JOIN del álgebra relacional. Las opciones incluyen **GROUP BY**, **ORDER BY**, **GROUP BY**... **HAVING**, **LIKE** y funciones internas **COUNT**, **SUM**, **AVG**, **MAX** y **MIN**. El enunciado SELECT puede operar sobre combinaciones de tablas y puede manejar **subconsultas**, incluidas subconsultas correlacionadas. También son posibles expresiones y operaciones de conjuntos. El comando UPDATE se puede usar para actualizar uno o más campos en uno o más registros. El comando INSERT puede insertar uno o más registros, posiblemente con valores nulos para algunos campos. El operador DELETE borra registros, mientras deja intacta la estructura de la tabla.

Una base de datos activa es aquella donde el DBMS monitorea activamente los cambios para garantizar que sólo se crean instancias legales de la base de datos. Para crear una base de datos activa se puede usar una combinación de restricciones y **disparadores** (triggers).

El enunciado **COMMIT** hace permanentes todos los cambios que se han hecho mediante la transacción actual. El enunciado **ROLLBACK** deshace todos los cambios que se realizaron mediante la transacción actual. La transacción actual comienza inmediatamente después del último COMMIT o ROLLBACK, o, si ninguno de éstos ocurrió, entonces al comienzo de la actual sesión de usuario.

SQL se usa con frecuencia en un entorno de programación en lugar de interactivamente. Se puede **incrustar** (embedded) en un lenguaje de programación huésped y compilar por separado mediante un **precompilador**. Se puede usar con un API estándar a través de **ODBC** o **JDBC**. También se puede usar como un lenguaje completo usando sus propios **PSM SQL**. El PL/SQL de Oracle es un ejemplo de un entorno de programación completo para creación de PSM SQL.

El comando **CREATE VIEW** se usa para definir una tabla virtual, mediante la selección de campos de tablas base existentes o vistas definidas con antelación. La operación SELECT se puede usar en vistas, pero otros comandos DML están restringidos a ciertos tipos de vistas. Un **disparador INSTEAD OF** es útil para sustituir comandos DML del usuario escritos en una vista con comandos correspondientes en la(s) tabla(s) base usada(s) para la vista. Una definición de vista se puede destruir mediante un comando DROP VIEW.

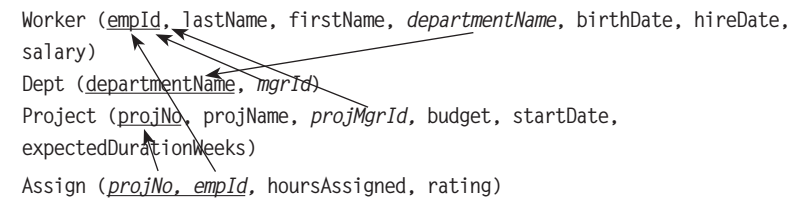
El **catálogo del sistema** o **diccionario de datos del sistema** es una base de datos que contiene información acerca de la base de datos del usuario. Sigue la pista de las tablas, columnas, índices y vistas que existen, así como información de autorización y otros datos. El sistema automáticamente actualiza el catálogo cuando se realizan cambios estructurales y otras modificaciones.

Ejercicios

6.1 Escriba los comandos necesarios para crear índices para las tablas `Student`, `Faculty`, `Class` y `Enroll` de este capítulo.

6.2 Para cada uno de los ejemplos de combinación (ejemplos 7-11) de la sección 6.4.2, sustituya la combinación mediante una subconsulta, si es posible. Si no es posible, explique por qué no.

Instrucciones para los ejercicios 6.3-6.25: Para el esquema que sigue, escriba los comandos indicados en SQL. La figura 6.8 muestra el DDL para crear estas tablas. Muestra que `departmentName` es una clave externa en la tabla `Worker`, que `mgrId` es una clave externa en la tabla `Dept`, que `projMgrId` es una clave externa en la tabla `Project`, y que `projNo` y `empId` son claves externas en la tabla `Assign`. Suponga que cada departamento tiene un supervisor (manager) y que cada proyecto tiene un supervisor, pero éstos no necesariamente están relacionados. (Nota: Se recomienda que realice el Ejercicio de laboratorio 1 en conjunto con estos ejercicios. Si Oracle no está disponible, puede usar otro DBMS relacional, incluido el freeware MySQL, que puede descargar. Dependiendo del producto es posible que necesite hacer algunos cambios al DDL. Si no planea realizar el Ejercicio de laboratorio 1, simplemente puede escribir los comandos.)



6.3 Obtenga los nombres de todos los trabajadores en el departamento accounting (contabilidad).

6.4 Obtenga una lista alfabética de nombres de todos los trabajadores asignados al proyecto 1001.

6.5 Obtenga el nombre del empleado (employee) en el departamento research (investigación) que tenga el salario más bajo (lowest).

6.6 Obtenga detalles del proyecto con el presupuesto (budget) más alto (highest).

6.7 Obtenga los nombres y departamentos de todos los trabajadores en el proyecto 1019.

6.8 Obtenga una lista alfabética de nombres y calificaciones (ratings) correspondientes de todos los trabajadores en cualquier proyecto que esté supervisado (managed) por Michael Burns.

FIGURA 6.8

DDL y enunciados Insert para el ejemplo Worker-Project-Assign

```
CREATE TABLE Dept (
departmentName VARCHAR2 (15),
mgrId Number(6),
CONSTRAINT Dept_departmentName_pk PRIMARY KEY (departmentName));
CREATE TABLE Worker (
empId NUMBER (6),
lastName VARCHAR2 (20) NOT NULL,
firstName VARCHAR2 (15) NOT NULL,
departmentName  VARCHAR2 (15),
birthDate DATE,
hireDate DATE,
salary NUMBER (8, 2),
CONSTRAINT Worker_empid_pk PRIMARY KEY (empid),
CONSTRAINT Worker_departmentNumber_fk FOREIGN KEY (departmentNumber) REFERENCES
Dept (departmentName));

ALTER TABLE Dept Add CONSTRAINT Dept_mgrId_fk FOREIGN KEY (mgrId) REFERENCES Worker (empId)) ON UPDATE

CREATE TABLE Project (
projNo NUMBER (6),
projName VARCHAR2 (20),
projMgrId VARCHAR2 (20),
budget NUMBER (8, 2),
startDate DATE,
expectedDurationWeeks NUMBER (4),
CONSTRAINT Project_projNo_pk PRIMARY KEY (projNo),
CONSTRAINT Project_projMgrId_fk FOREIGN KEY (projMgrid) REFERENCES WORKER (empId));

CREATE TABLE Assign (
projNo NUMBER (6),
empId NUMBER (6),
hoursAssigned NUMBER (3),
rating NUMBER (1),
CONSTRAINT Assign_projNo_empId_pk PRIMARY KEY (projNo, empId),
CONSTRAINT Assign_projNo_fk FOREIGN KEY (projNo) REFERENCES Project (projNo) ON UPDATE
CONSTRAINT Assign_empId_fk FOREIGN KEY (empId) REFERENCES Worker (empId) ON
ON DELETE CASCADE);

INSERT INTO Dept VALUES ('Accounting',);
INSERT INTO Dept VALUES ('Research',);

INSERT INTO Worker VALUES(101, 'Smith', 'Tom', 'Accounting', '01-Feb-1960', '06-Jun-1983', 50000);
INSERT INTO Worker VALUES(103, 'Jones', 'Mary' , 'Accounting', '15-Jun-1965', '20-Sep-1985', 48000);
INSERT INTO Worker VALUES(105, 'Burns', 'Jane', 'Accounting', '21-Sep-1970' , '12-Jun-1990', 39000);
INSERT INTO Worker VALUES(110, 'Burns', 'Michael', 'Research', '05-Apr-1967', '10-Sep-1990', 70000);
INSERT INTO Worker VALUES(115, 'Chin', 'Amanda', 'Research', '22-Sep-1965', '19-Jun-1985', 60000);

UPDATE Dept SET mgrId = 101 WHERE departmentName = 'Accounting';
UPDATE Dept SET mgrId = 101 WHERE departmentName = 'Research';

INSERT INTO Project VALUES (1001, 'Jupiter', 101, 300000, '01-Feb-2004', 50);
INSERT INTO Project VALUES (1005, 'Saturn', 101, 400000, '01-Jun-2004', 35);
INSERT INTO Project VALUES (1019, 'Mercury', 110, 350000, '15-Feb-2004', 40);
INSERT INTO Project VALUES (1025, 'Neptune', 110, 600000, '01-Feb-2005', 45);
INSERT INTO Project VALUES (1030, 'Pluto', 110, 380000, '15-Sept-2004', 50);
```

(*continúa*)

```
INSERT INTO Assign(projNo, empId, hoursAssigned) VALUES (1001, 101, 30);
INSERT INTO Assign VALUES (1001, 103, 20, 5);
INSERT INTO Assign(projNo, empId, hoursAssigned)VALUES (1005, 103, 20);
INSERT INTO Assign(projNo, empId, hoursAssigned)VALUES (1001, 105, 30);
INSERT INTO Assign VALUES (1001, 115, 20, 4);
INSERT INTO Assign VALUES (1019, 110, 20, 5);
INSERT INTO Assign VALUES (1019, 115, 10, 4);
INSERT INTO Assign(projNo, empId, hoursAssigned)VALUES (1025, 110, 10);
INSERT INTO Assign(projNo, empId, hoursAssigned)VALUES (1030, 110, 10);
```

FIGURA 6.8
Continuación

6.9 Cree una vista que tenga número de proyecto y nombre de cada proyecto, junto con las ID y nombres de todos los trabajadores asignados a él.

6.10 Con la vista creada en el ejercicio 6.9, encuentre el número de proyecto y nombre de proyecto de todos los proyectos a los que está asignado el empleado 110.

6.11 Agregue un nuevo trabajador llamado Jack Smith con ID de 1999 al departamento research.

6.12 Cambie las horas que tiene asignadas el empleado 110 al proyecto 1019, de 20 a 10.

6.13 Para todos los proyectos que comiencen (starting) después del 1 de mayo de 2004, encuentre el número de proyecto y las ID y nombres de todos los trabajadores asignados a ellos.

6.14 Para cada proyecto, haga una lista del número de proyecto y cuántos trabajadores se asignan a él.

6.15 Encuentre los nombres de empleado y nombres de supervisor de departamento de todos los trabajadores que no estén asignados a proyecto alguno.

6.16 Encuentre los detalles de cualquier proyecto con la palabra "urn" en cualquier parte en su nombre.

6.17 Obtenga una lista de números de proyecto y nombres y fechas de inicio de todos los proyectos que tienen la misma fecha de inicio.

6.18 Agregue un campo llamado `status` a la tabla `Project`. Los valores muestra para este campo son `active` (activo), `completed` (completado), `planned` (planificado), `cancelled` (cancelado). Luego escriba el comando para deshacer este cambio.

6.19 Obtenga la ID de empleado y número de proyecto de todos los empleados que no tengan calificaciones en dicho proyecto.

6.20 Si supone que `salary` ahora contiene salario anual, encuentre la ID, nombre y salario mensual de cada trabajador.

6.21 Agregue un campo llamado `numEmployeesAssigned` a la tabla `Project`. Use el comando UPDATE a fin de insertar valores en el campo para que corresponda con la información actual en la tabla `Assign`. Luego escriba un disparador que actualice el campo correctamente siempre que se realice, elimine o actualice una asignación. Escriba el comando para hacer permanentes estos cambios.

6.22 a. Escriba una consulta de diccionario de datos Oracle a fin de mostrar los nombres de todas las columnas en una tabla llamada `Customers` (clientes).

b. Escriba una consulta correspondiente para las tablas DB2 UDB SYSCAT para este ejemplo.

6.23 a. Escriba una consulta de diccionario de datos Oracle para encontrar toda la información acerca de todas las columnas llamadas PROJNO.

b. Escriba una consulta correspondiente para las tablas DB2 UDB SYSCAT para este ejemplo.

6.24 a. Escriba una consulta de diccionario de datos Oracle para obtener una lista de nombres de las personas que hayan creado tablas, junto con el número de tablas que creó cada una. Suponga que tiene privilegios ABD.

b. Escriba una consulta correspondiente para las tablas DB2 UDB SYSCAT para este ejemplo.

6.25 a. Escriba una consulta de diccionario de datos Oracle para encontrar los nombres de las tablas que tengan más de dos índices.

b. Escriba una consulta correspondiente para las tablas DB2 UDB SYSCAT para este ejemplo.

Ejercicios de laboratorio

Ejercicio de laboratorio 6.1. Exploración de la base de datos Oracle para el ejemplo Worker-Dept-Project-Assign

Un script para crear una base de datos Oracle para el ejemplo usado en los ejercicios 6.3-6.25 aparece en la figura 6.8 y en el website que acompaña a este libro. El script se escribió usando Notepad. Encuentre el script en el website, cópielo en su propio directorio y ábralo con Notepad. Abra la facilidad SQLPlus de Oracle. Cambiará de ida y vuelta entre Notepad y SQLPlus, porque el editor en SQLPlus es difícil de usar.

a. En Notepad, resalte y copie el comando para crear la primera tabla, luego cambie a SQLPlus y pegue dicho comando en la ventana SQLPlus y ejecute el comando. Debe ver el mensaje "Table created" (tabla creada). Si en vez de ello obtiene un mensaje de error, regrese al archivo Notepad y corrija el error.

b. Continúe para crear las tablas restantes una a la vez en la misma forma.

c. Corra los comandos INSERT para poblar las tablas. Explique por qué se usaron los dos enunciados UPDATE.

d. Con esta implementación, ejecute los enunciados SQL Oracle para los ejercicios 6.3-6.25.

Ejercicio de laboratorio 6.2. Creación y uso de una base de datos simple en Oracle

a. Escriba los comandos DDL para crear las tablas `Student`, `Faculty`, `Class` y `Enroll` para la base de datos University que se muestra en la figura 6.2.

b. Con el comando INSERT agregue los registros conforme aparezcan en la figura 6.3 a su nueva base de datos Oracle.

c. Escriba consultas SQL para las siguientes preguntas y ejecútelas.

i. Encuentre los nombres de todas las especialidades history.

ii. Encuentre el número de clase (class number), horario (schedule) y salón (room) para todas las clases que imparte Smith del departamento historia (history).

iii. Encuentre los nombres de todos los estudiantes que tienen menos que el número promedio de créditos.

iv. Encuentre los nombres de todos los profesores que tiene Ann Chin, junto con todas sus clases y calificaciones de mitad de semestre de cada una.

v. Para cada estudiante, encuentre el número de clases en las que está inscrito.

PROYECTO DE MUESTRA: CREACIÓN Y MANIPULACIÓN DE UNA BASE DE DATOS RELACIONAL PARA LA GALERÍA DE ARTE

En la sección del proyecto de muestra al final del capítulo 5 creó un modelo relacional normalizado para la base de datos de la Galería de Arte. Al renombrar ligeramente las tablas se concluyó que el siguiente modelo se debe implementar:

(1) Artist (artistId, firstName, lastName, interviewDate, interviewerName, areaCode, telephoneNumber, street, *zip*, salesLastYear, salesYearToDate, socialSecurityNumber, usualMedium, usualStyle, usualType)

(2) Zips (zip, city, state)

(3) PotentialCustomer (potentialCustomerId, firstname, lastName, areaCode, telephoneNumber, street, *zip*, dateFilledIn, *preferredArtistId*, preferredMedium, preferredStyle, preferredType)

(4) Artwork (artworkId, *artistId*, workTitle, askingPrice, dateListed, dateReturned, dateShown, status, workMedium, workSize, workStyle, workType, workYearCompleted, *collectorSocialSecurityNumber*)

(5) ShownIn (*artworkId,showTitle*)

(6) Collector (socialSecurityNumber, firstName, lastName, street, *zip*, interviewDate, interviewerName, areaCode, telephonenumber, salesLastYear, salesYearToDate, *collectionArtistId*, collectionMedium, collectionStyle, collectionType, SalesLastYear, SalesYearToDate)

(7) Show (showTitle, *showFeaturedArtistId*, showClosingDate, showTheme, showOpeningDate)

(8) Buyer (buyerId, firstName, lastName, street, *zip*, areaCode, telephoneNumber, purchasesLastYear, purchasesYearToDate)

(9) Sale (InvoiceNumber, *artworkId*, amountRemittedToOwner, saleDate, salePrice, saleTax, *buyerId*, *salespersonSocialSecurityNumber*)

(10) Salesperson (socialSecurityNumber, firstName, lastName, street, *zip*)

- Paso 6.1. Actualice el diccionario de datos y lista de suposiciones si se necesita. Para cada tabla escriba el nombre de la tabla y los nombres, tipos de datos y tamaños de todos los ítems de datos, identifique cualquier restricción y utilice las convenciones del DBMS que usará para la implementación.

A la lista de suposiciones no se hicieron cambios. No se necesitan más cambios al diccionario de datos. Para una base de datos Oracle, las tablas tendrán las siguientes estructuras:

TABLE Zips

Item	Datatype	Size	Constraints	Comments
Zip	CHAR	5	PRIMARY KEY	
city	VARCHAR2	15	NOT NULL	
state	CHAR	2	NOT NULL	

TABLE Artist

Item	Datatype	Size	Constraints	Comments
artistId	NUMBER	6	PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL; (firstName, lastName) UNIQUE	
lastName	VARCHAR2	20	NOT NULL; (firstName, lastName) UNIQUE	
interviewDate	DATE			
interviewerName	VARCHAR2	35		
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	
salesLastYear	NUMBER	8,2		
salesYearToDate	NUMBER	8,2		
socialSecurityNumber	CHAR	9	UNIQUE	
usualMedium	VARCHAR	15		
usualStyle	VARCHAR	15		
usualType	VARCHAR	20		

TABLE Collector

Item	Datatype	Size	Constraints	Comments
socialSecurityNumber	CHAR	9	PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL	
lastName	VARCHAR2	20	NOT NULL	
interviewDate	DATE			
interviewerName	VARCHAR2	35		
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY Ref Zips	
salesLastYear	NUMBER	8,2		
salesYearToDate	NUMBER	8,2		
collectionArtistId	NUMBER	6	FOREIGN KEY REF Artist	
collectionMedium	VARCHAR	15		
collectionStyle	VARCHAR	15		
collectionType	VARCHAR	20		

TABLE PotentialCustomer

Item	Datatype	Size	Constraints	Comments
potentialCustomerId	NUMBER	6	PRIMARY KEY	
firstname	VARCHAR2	15	NOT NULL	
lastName	VARCHAR2	20	NOT NULL	
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	
dateFilledIn	DATE			
preferredArtistId	NUMBER	6	FOREIGN KEY REF Artist	
preferredMedium	VARCHAR2	15		
preferredStyle	VARCHAR2	15		
preferredType	VARCHAR2	20		

TABLE Artwork

Item	Datatype	Size	Constraints	Comments
artworkId	NUMBER	6	PRIMARY KEY	
artistId	NUMBER	6	FOREIGN KEY REF Artist; NOT NULL; (artistId, workTitle) UNIQUE	
workTitle	VARCHAR2	50	NOT NULL; (artistId, workTitle) UNIQUE	
askingPrice	NUMBER	8,2		
dateListed	DATE			
dateReturned	DATE			
dateShown	DATE			
status	VARCHAR2	15		
workMedium	VARCHAR2	15		
workSize	VARCHAR2	15		
workStyle	VARCHAR2	15		
workType	VARCHAR2	20		
workYearCompleted	CHAR	4		
collectorSocialSecurityNumber	CHAR	9	FOREIGN KEY REF Collector	

TABLE Show

Item	Datatype	Size	Constraints	Comments
showTitle	VARCHAR2	50	PRIMARY KEY	
showFeaturedArtistId	NUMBER	6	FOREIGN KEY REF Arist	
showClosingDate	DATE			
showTheme	VARCHAR2	50		
showOpeningDate	DATE			

TABLE ShownIn

Item	Datatype	Size	Constraints	Comments
artworkId	NUMBER	6	PRIMARY KEY(artworkId, showTitle); FOREIGN KEY REF Artwork	
showTitle	VARCHAR2	50	PRIMARY KEY(artworkId, showTitle); FOREIGN KEY REF Show	

TABLE Buyer

Item	Datatype	Size	Constraints	Comments
buyerId	NUMBER	6	PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL	
lastName	VARCHAR2	20	NOT NULL	
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	
areaCode	CHAR	3		
telephoneNumber	CHAR	7		
purchasesLastYear	NUMBER	8,2		
purchasesYearToDate	NUMBER	8,2		

TABLE Salesperson				
Item	**Datatype**	**Size**	**Constraints**	**Comments**
socialSecurityNumber	CHAR	9	PRIMARY KEY	
firstName	VARCHAR2	15	NOT NULL; (firstName,lastName) UNIQUE	
lastName	VARCHAR2	20	NOT NULL; (firstName,lastName) UNIQUE	
street	VARCHAR2	50		
zip	CHAR	5	FOREIGN KEY REF Zips	

TABLE Sale				
Item	**Datatype**	**Size**	**Constraints**	**Comments**
invoiceNumber	NUMBER	6	PRIMARY KEY	
artworkId	NUMBER	6	NOT NULL; UNIQUE; FOREIGN KEY REF Artwork	
amountRemittedToOwner	NUMBER	8,2	DEFAULT 0.00	
saleDate	DATE			
salePrice	NUMBER	8,2		
saleTax	NUMBER	6,2		
buyerId	NUMBER	6	NOT NULL; FOREIGN KEY REF Buyer	
salespersonSocialSecurityNumber	CHAR	9		

- Paso 6.2. Escriba y ejecute enunciados SQL con el fin de crear todas las tablas necesarias para implementar el diseño.

Puesto que se quiere especificar claves externas conforme se creen las tablas, debe tener cuidado con el orden en el que se crean, porque la "tabla origen" tiene que existir antes de crear la tabla que contiene la clave externa. Por tanto, se usará el siguiente orden: Zips, Artist, Collector, Potential Customer, Artwork, Show, ShowIn, Buyer, Salesperson, Sale. Los enunciados DDL para crear las tablas se muestran en la figura 6.9. Se usa la sintaxis Oracle, pero los enunciados DDL deben funcionar, con modificaciones menores, para cualquier DBMS relacional.

- Paso 6.3. Cree índices para claves externas y cualquier otra columna que se usará con más frecuencia para las consultas.

Los enunciados DDL para crear los índices se muestran en la figura 6.10.

- Paso 6.4. Inserte aproximadamente cinco registros en cada tabla, que preserven todas las restricciones. Ponga suficientes datos para demostrar cómo funcionará la base de datos.

La figura 6.11 muestra el enunciado INSERT. Puesto que se quiere usar los valores generados por el sistema de Oracle para claves subrogadas, se crean secuencias para cada uno de `artistId`, `potentialCustomerId`, `artworkId` y `buyerId`, usando este comando:

```
CREATE SEQUENCE  nombre_secuencia
[START WITH valor_inicial]
[INCREMENT BY paso] . . .;
```

Se elige comenzar con uno e incrementar por uno, los defectos. Para generar cada nuevo valor, use el comando <*nombre_secuencia*>.NEXTVAL, como se muestra en los comandos INSERT. Se supone que `invoiceNumber` es un número preimpreso en la forma de factura, no generada por el sistema, así que no se necesita una secuencia para dicho número.

```
CREATE TABLE Zips (
        zip CHAR (5),
        city VARCHAR2 (15) NOT NULL,
        state CHAR (2) NOT NULL,
        CONSTRAINT Zips_zip_pk PRIMARY KEY (zip));

CREATE TABLE Artist (
        ArtistId NUMBER (6),
        firstName VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
        interviewDate DATE,
        interviewerName VARCHAR2 (35),
        areaCode  CHAR (3),
        telephoneNumber  CHAR (7),
        street VARCHAR2 (50),
        zip  CHAR (5),
        salesLastYear NUMBER (8, 2),
        salesYearToDate   NUMBER (8, 2),
        socialSecurityNumber CHAR (9),
        usualMedium VARCHAR (15),
        usualStyle VARCHAR (15),
        usualType VARCHAR (20),
        CONSTRAINT Artist_ArtistId_pk PRIMARY KEY (ArtistId),
        CONSTRAINT Artist_SSN_uk UNIQUE (socialSecurityNumber),
        CONSTRAINT Artist_fName_lName_uk UNIQUE (firstName,   lastName),
        CONSTRAINT Artist_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip));

CREATE TABLE Collector (
        socialSecurityNumber CHAR (9),
        firstName VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
        interviewDate DATE,
        interviewerName VARCHAR2 (35),
        areaCode CHAR (3),
        telephoneNumber   CHAR (7),
        street VARCHAR2 (50),
        zip CHAR (5),
        salesLastYear NUMBER (8, 2),
        salesYearToDate   NUMBER (8, 2),
        collectionArtistId NUMBER (6),
        collectionMedium VARCHAR (15),
        collectionStyle VARCHAR (15),
        collectionType VARCHAR (20),
        CONSTRAINT Collector_SSN_pk PRIMARY KEY
       (socialSecurityNumber).
        CONSTRAINT Collector_collArtistid_fk FOREIGN KEY (collectionArtistId)

REFERENCES Artist (artistId)
        CONSTRAINT Collector_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip));

CREATE TABLE PotentialCustomer (
        potentialCustomerId NUMBER (6),
        firstname VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
```

(*continúa*)

FIGURA 6.9

Enunciados DDL Oracle para las tablas de la Galería de Arte

FIGURA 6.9
Continuación

```
        areaCode CHAR (3),
        telephoneNumber CHAR (7),
        street VARCHAR2 (50),
        zip CHAR (5),
        dateFilledIn DATE,
        preferredArtistId NUMBER (6),
        preferredMedium VARCHAR2 (15),
        preferredStyle VARCHAR2 (15),
        preferredType VARCHAR2 (20),
        CONSTRAINT PotentialCustomer_potCusId_pk PRIMARY KEY
        (potentialCustomerId),
        CONSTRAINT PotentialCustomer_zip_fk FOREIGN KEY (zip)
        REFERENCES Zips (zip)
        CONSTRAINT Potential Customer_prefAId_fk FOREIGN KEY
        (preferredArtistId) REFERENCES Artist (artist_Id)) ;

CREATE TABLE Artwork (
        artworkId NUMBER (6),
        artistId NUMBER (6) NOT NULL,
        workTitle VARCHAR2 (50) NOT NULL,
        askingPrice NUMBER (8, 2),
        dateListed DATE,
        dateReturned DATE,
        dateShown DATE,
        status VARCHAR2 (15),
        workMedium VARCHAR2 (15),
        workSize VARCHAR2 (15),
        workStyle VARCHAR2 (15),
        workType VARCHAR2 (20),
        workYearCompeted CHAR (4),
        collectorSocialSecurityNumber CHAR (9),
        CONSTRAINT Artwork_artworkId_pk PRIMARY KEY (artworkId).
        CONSTRAINT Artwork_artId_wTitle_uk UNIQUE (artistId, workTitle),
        CONSTRAINT Artwork_artId_fk FOREIGN KEY (artistId) REFERENCES Artist (artistId)
        CONSTRAINT Artwork_collSSN_fk FOREIGN KEY
(collectorSocialSecurityNumber)   REFERENCES Collector (socialSecurityNumber);

CREATE TABLE Show (
        showTitle VARCHAR2 (50),
        showFeaturedArtistId NUMBER (6),
        showClosingDate DATE,
        showTheme VARCHAR2 (50),
        showOpeningDate DATE,
        CONSTRAINT Show_showTitle_pk PRIMARY KEY (showTitle),
        CONSTRAINT Show_showFeaturedArtId_fk FOREIGN KEY (showFeaturedArtistId)

REFERENCES Artist (artistId);
```

(*continúa*)

FIGURA 6.9
Continuación

```
CREATE TABLE ShownIn (
        artworkId NUMBER (6),
        showTitle VARCHAR2 (50),
        CONSTRAINT ShownIn_artId_showTitle_pk PRIMARY KEY (artworkId, showTitle),
        CONSTRAINT ShownIn_artId_fk FOREIGN KEY (artworkId) REFERENCES Artwork
        (artworkId)
        CONSTRAINT ShownIn_showTitle_fk FOREIGN KEY (showTitle) REFERENCES Show
        (showTitle));

CREATE TABLE Buyer (
        buyerId NUMBER (6),
        firstName VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
        street VARCHAR2 (50),
        zip CHAR (5),
        areaCode CHAR (3),
        telephoneNumber CHAR (7),
        purchasesLastYear NUMBER (8, 2),
        purchasesYearToDate NUMBER (8, 2),
        CONSTRAINT Buyer_buyerId_pk PRIMARY KEY (buyerId),
        CONSTRAINT Buyer_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip));

CREATE TABLE Salesperson (
        socialSecurityNumber CHAR (9),
        firstName VARCHAR2 (15) NOT NULL,
        lastName VARCHAR2 (20) NOT NULL,
        street VARCHAR2 (50),
        zip CHAR (5),
        CONSTRAINT Salesperson_SSN_pk PRIMARY KEY
        socialSecurityNumber),
        CONSTRAINT Salesperson_fName_lName_uk UNIQUE (firstName, lastName),
        CONSTRAINT Salesperson_zip_fk FOREIGN KEY (zip) REFERENCES Zips (zip));

CREATE TABLE Sale (
        invoiceNumber NUMBER (6),
        artworkId NUMBER (6) NOT NULL,
        amountRemittedToOwner NUMBER (8, 2) DEFAULT 0.00,
        saleDate DATE,
        salePrice NUMBER (8, 2),
        saleTax NUMBER (6, 2),
        buyerId NUMBER (6) NOT NULL,
        salespersonSocialSecurityNumber CHAR (9),
        CONSTRAINT Sale_invoiceNumber_pk PRIMARY KEY (invoiceNumber),
        CONSTRAINT Sale_artworkId_uk UNIQUE (artworkId),
        CONSTRAINT Sale_artworkId_fk FOREIGN KEY (artworkId) REFERENCES Artwork (artworkId),
        CONSTRAINT Sale_buyerId_fk FOREIGN KEY (buyerId) REFERENCES Buyer (buyerId));
```

FIGURA 6.10

Enunciados DDL Oracle para los índices de la Galería de Arte

```
CREATE UNIQUE INDEX Artist_lastName_firstName ON Artist(lastName, firstName);
CREATE INDEX Artist_zip ON Artist(zip);

CREATE INDEX Collector_collectionArtistId On Collector(collectionArtistId);
CREATE INDEX Collector_zip ON Collector(zip);
CREATE INDEX Collector_lastName_firstName ON Collector(lastName, firstName);

CREATE INDEX PotentialCustomer_zip ON PotentialCustomer(zip);
CREATE INDEX PotentialCustomer_lastName_firstName ON PotentialCustomer(lastName, firstName);

CREATE UNIQUE INDEX Artwork_artistId_workTitle ON Artwork (artistId, workTitle);
CREATE INDEX Artwork_artistId ON Artwork(artistId);
CREATE INDEX Artwork_collectorSocialSecurityNumber ON Artwork
(collectorSocialSecurityNumber);

CREATE INDEX Show_showFeaturedArtistId On Show (showFeaturedArtistId);

CREATE INDEX Shownin_artworkId ON Shownin (artworkId);
CREATE INDEX Shownin_show Title ON ShownIn (showTitle);

CREATE INDEX Buyer_zip ON Buyer(zip);
CREATE INDEX Buyer_lastName_firstName ON Buyer (lastName, firstName);

CREATE UNIQUE INDEX Salesperson_lastName_firstName ON Salesperson (lastName, firstName);
CREATE INDEX Salesperson_zip ON Salespeson (zip);

CREATE INDEX Sale_buyerId ON Sale (buyerId);
```

FIGURA 6.11

Enunciados INSERT para poblar las tablas de la Galería de Arte

```
INSERT INTO Zips VALUES ('10101', 'New York', 'NY');
INSERT INTO Zips VALUES ('10801', 'New Rochelle', 'NY');
INSERT INTO Zips VALUES ('92101', 'San Diego', 'CA');
INSERT INTO Zips VALUES ('33010', 'Miami', 'FL');
INSERT INTO Zips VALUES ('60601', 'Chicago', 'IL');

CREATE SEQUENCE artistId_sequence;
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL, 'Leonardo', 'Vincenti', '10-Oct-1999', 'Hughes,''212', '5559999', '10 Main
Street', '10101', 9000, 4500, '099999876', 'oil', 'realism', 'painting');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL, 'Vincent', 'Gogh', '15-Jun-2004', 'Hughes', '914,''5551234,''55
West 18 Street', '10801', 9500, 5500, '099999877', 'oil', 'impressionism', 'painting');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL, 'Winslow', 'Homes', '05-Jan-2004', 'Hughes', '619', '1234567', '100
Water Street', '92101', 14000, 4000, '083999876', 'watercolor', 'realism', 'painting');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL, 'Alexander', 'Calderone', '10-Feb-1999', 'Hughes', '212', '5559999',
'10 Main Street', '10101', 20000, 20000, '123999876', 'steel', 'cubism', 'sculpture');
INSERT INTO Artist VALUES(artistId_sequence.NEXTVAL, 'Georgia', 'Keefe', '05-Oct-2004', 'Hughes', '305', '1239999', '5
Chestnut Street', '33010', 19000, 14500, '987999876', 'oil', 'realism', 'painting');

INSERT INTO Collector VALUES('102345678', 'John', 'Jackson', '01-Feb-2004', 'Hughes', '917', '7771234', '24 Pine Avenue',
'10101', 4000, 3000, 1, 'oil', 'realism', 'collage');
INSERT INTO Collector VALUES ('987654321', 'Mary', 'Lee', '01-Mar-2003', 'Jones', '305', '5551234', '10 Ash Street', 33010,
'2000', 3000, 2, 'watercolor', 'realism', 'painting');
INSERT INTO Collector VALUES('034345678', 'Ramon', 'Perez', '15-Apr-2003', 'Hughes', '619', '8881234', '15 Poplar Avenue',
'92101', 4500, 3500, 3, 'oil', 'realism', 'painting');
INSERT INTO Collector VALUES('888881234', 'Rick', 'Lee', '20-Jun-2004', 'Hughes', '212', '9991234', '24 Pine Avenue', '10101',
4000, 3000, 3, 'oil', 'realism', 'sculpture');
```

(*continúa*)

FIGURA 6.11
Continuación

```
INSERT INTO Collector VALUES('777345678', 'Samantha', 'Torno', '05-May-2004', 'Jones', '305', '5551234', '10 Ash Street',
'33010', 40000, 30000, 1, 'acrylic', 'realism', 'painting');

CREATE SEQUENCE potentialCustomerId-sequence;
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL, 'Adam', 'Burns', '917', '3456789', '1
Spruce Street', '10101', '12-Dec-2003', 1, 'watercolor', 'impressionism', 'painting');
INSERT INTO PotentialCustomer VALUES(potentialCustomerId sequence.NEXTVAL, 'Carole', 'Burns', '917', '3456789', '1 Spruce
Street', '10101', '12-Dec-2003', 2, 'watercolor', 'realism', sculpture');
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL, 'David', 'Engel', '914', '7777777', '715
North Avenue', '10801', '08-Aug-2003', 3, 'watercolor', 'realism', 'painting');
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL, 'Frances', 'Hughes', '619', '3216789', '10
Pacific Avenue', '92101', '05-Jan-2004', 2, 'oil', 'impressionism', 'painting');
INSERT INTO PotentialCustomer VALUES(potentialCustomerId_sequence.NEXTVAL, 'Irene', 'Jacobs', '312', '1239876', '1
Windswept Place', '60601', '21-Sep-2003', 5, 'watercolor', 'abstract expressionism', 'painting');

CREATE SEQUENCE artworkId_sequence;
INSERT INTO Artwork VALUES(artworkid_sequence.NEXTVAL, 1, 'Flight', 15000.00, '08-Sep-2003',NULL ,NULL, 'for
sale',oil', '36 in X 48 in', 'realism', 'painting',2001',NULL );
INSERT INTO Artwork VALUES(artworkid_sequence.NEXTVAL, 3, 'Bermuda Sunset', 8000.00, '15-Mar-2004',NULL
,01-Apr-2004' , 'sold','watercolor', '22 in X 28 in', 'realism', 'painting', 2003',NULL );
INSERT INTO Artwork VALUES(artworkid_sequence.NEXTVAL, 3, 'Mediterranean Coast', 4000.00, '18-Oct-2003',NULL
,01-Apr-2004', 'for sale',watercolor', '22 in X 28 in', 'realism', 'painting','2000',102345678');
INSERT INTO Artwork VALUES(artworkid_sequence.NEXTVAL, 5, 'Ghost orchid', 18000.00',
05-Jun-2003',NULL ,NULL , 'sold',oil', '36 in X 48 in', 'realism', 'painting','2001',034345678' );
INSERT INTO Artwork VALUES(artworkid_sequence.NEXTVAL, 4, 'Five Planes', 15000.00,
'10-Jan-2004',NULL ,'10-Mar-2004' , 'for sale',steel', '36inX30inX60in', 'cubism''sculpture' '2003',034345678' );

INSERT INTO Show VALUES('The Sea in Watercolor',3, '30-Apr-2004', 'seascapes', '01-Apr-2004');
INSERT INTO Show VALUES('Calderone: Mastery of Space',4,'20-Mar-2004',mobiles', '10-Mar-2004');

INSERT INTO ShownIn VALUES(2,'The Sea in Watercolor');
INSERT INTO ShownIn VALUES(3, 'The Sea in Watercolor');
INSERT INTO ShownIn VALUES(5, 'Calderone: Mastery of Space');

CREATE SEQUENCE buyerId_sequence;
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL, 'Valerie', 'Smiley', '15 Hudson Street', '10101', '718','5551234',
5000, 7500);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL, 'Winston', 'Lee', '20 Liffey Avenue', '60601', '312','7654321',
3000, 0);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL, 'Samantha', 'Babson', '25 Thames Lane', '92101', '619','4329876',
15000, 0);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL, 'John', 'Flagg', '22 Amazon Street', '10101', '212','7659876',
3000, 0);
INSERT INTO Buyer VALUES (BuyerId_sequence.NEXTVAL, 'Terrence', 'Smallshaw', '5 Nile Street', '33010', '305','2323456',
15000, 17000);

INSERT INTO Salesperson VALUES('102445566', 'John','Smith', '10 Sapphire Row', '10801');
INSERT INTO Salesperson VALUES('121344321', 'Alan','Hughes', '10 Diamond Street', '10101');
INSERT INTO Salesperson VALUES('101889988', 'Mary','Brady', '10 Pearl Avenue', '10801');
INSERT INTO Salesperson VALUES('111223344', 'Jill','Fleming', '10 Ruby Row', '10101');
INSERT INTO Salesperson VALUES('123123123', 'Terrence','DeSimone', '10 Emerald Lane', '10101');

INSERT INTO Sale VALUES(1234, 2,NULL ,'05-Apr-2004',7500,600, 1, '102445566');
INSERT INTO Sale VALUES(1235, 6,NULL ,'06-Apr-2004',17000,1360, 5, '121344321');
```

- Paso 6.5. Escriba enunciados SQL que procesarán cinco solicitudes no rutinarias para información de la base de datos recién creada. Para cada una, escriba la solicitud en inglés, seguida por el correspondiente comando SQL.

1. Encuentre los nombres de todos los artistas que se entrevistaron después del 1 de enero de 2004, pero que no tengan obras de arte en lista.

```
SELECT firstName, lastName
FROM Artist
WHERE interviewDate > '01-Jan-2004' AND NOT EXISTS
  (SELECT *
  FROM Artwork
  WHERE artistId =Artist.artistId);
```

2. Encuentre la comisión total para el vendedor John Smith obtenida entre las fechas 1 de abril de 2004 y 15 de abril de 2004. Recuerde que la galería carga 10% de comisión y el vendedor recibe la mitad de eso, que es 5% del precio de venta.

```
SELECT .05 * SUM(salePrice)
FROM Sale
WHERE saleDate > = '01-Apr-2004' AND
  saleDate < = '15-Apr-2004' AND
  salespersonSocialSecurityNumber = (SELECT socialSecurityNumber
                                     FROM Salesperson
                                     WHERE firstName= 'John' AND lastName
                                       = 'Smith');
```

3. Encuentre los nombres de coleccionista, nombres de artista y títulos de todas las obras de arte que sean propiedad de coleccionistas (collectors), no de los propios artistas, en orden del apellido del coleccionista.

```
SELECT Collector.firstName, Collector.lastName, Artist.firstName, Artist.
lastName, workTitle
FROM Artist, Artwork, Collector
WHERE   Artist.artistId = Artwork.artistId AND
        Artwork.collectorSocialSecurityNumber =
        Collector.socialSecurityNumber AND
        collectorSocialSecurityNumber IS NOT NULL
ORDER BY Collector.lastName, Collector.firstName;
```

4. Para cada comprador potencial, encuentre información acerca de las exposiciones que presenten a su artista preferido.

```
SELECT firstName, lastName, showTitle, showOpeningDate, showClosingDate,
potentialCustomerId
FROM Show, PotentialCustomer
WHERE showFeaturedArtistId = PotentialCustomer.preferredArtistId
ORDER BY potentialCustomerId;
```

5. Encuentre el precio de venta promedio de las obras de la artista Georgia Keefe.

```
SELECT AVG(salePrice)
FROM Sale
WHERE artworkId IN (SELECT artworkId
  FROM Artwork
  WHERE artistId = (SELECT ArtistId
                    FROM Artist
                    WHERE lastName = 'Keefe' AND firstName ='Georgia'));
```

- Paso 6.6. Cree al menos un disparador y escriba el código para él. Este disparador actualizará la cantidad de compras del comprador del año a la fecha siempre que una venta se complete.

```
CREATE TRIGGER UPDATEBUYERYTD
AFTER INSERT ON Sale
FOR EACH ROW
BEGIN
  UPDATE Buyer
  SET purchasesYearToDate = purchasesYearToDate + :NEW.salePrice
  WHERE Buyer.buyerId = :NEW.buyerId;
END;
```

PROYECTOS ESTUDIANTILES: CREACIÓN Y USO DE UNA BASE DE DATOS RELACIONAL PARA LOS PROYECTOS ESTUDIANTILES

Para las tablas normalizadas que desarrolló al final del capítulo 5 para el proyecto que eligió, realice los siguientes pasos para implementar el diseño usando un sistema de gestión de base de datos relacional como Oracle, SQLServer o MySQL.

- Paso 6.1. Actualice el diccionario de datos y lista de suposiciones según se requiera. Para cada tabla, escriba el nombre de tabla y los nombres, tipos de datos y tamaños de todos los ítems de datos, e identifique cualquier restricción, usando las convenciones del DBMS que usará para la implementación.
- Paso 6.2. Escriba y ejecute enunciados SQL para crear todas las tablas necesarias para implementar el diseño. El website para este libro tiene instrucciones para usar SQL con el fin de crear una base de datos Access.
- Paso 6.3. Cree índices para claves externas y para cualquier otra columna que necesite.
- Paso 6.4. Inserte al menos cinco registros en cada tabla, que preserve todas las restricciones. Ponga suficientes datos para demostrar cómo funcionará la base de datos.
- Paso 6.5. Escriba enunciados SQL que procesarán cinco solicitudes no rutinarias para información de la base de datos recién creada. Para cada una, escriba la solicitud en inglés, seguido por el correspondiente comando SQL.
- Paso 6.6. Cree al menos un disparador y escriba el código para él.