El lenguaje SQL

Carme Martín Escofet

PID_00201455

Universitat Oberta de Catalunya

Índice

Introducción

SQL es el lenguaje estándar ANSI/ISO de definición, manipulación y control de bases de datos (BD) relacionales. Es un lenguaje declarativo; sólo se tiene qué decir qué se quiere hacer. En cambio, en los lenguajes procedimentales hay que especificar cómo se tiene que hacer cualquier cosa sobre la BD. SQL es un lenguaje muy parecido al lenguaje natural, concretamente se parece al inglés, y es muy expresivo. Por estas razones, y como lenguaje estándar, SQL es un lenguaje con el que se puede acceder a todos los sistemas de gestión de bases de datos (SGBD) relacionales comerciales.

Empecemos con una breve explicación sobre el modo en que SQL ha llegado a ser el lenguaje estándar de las BD relacionales:

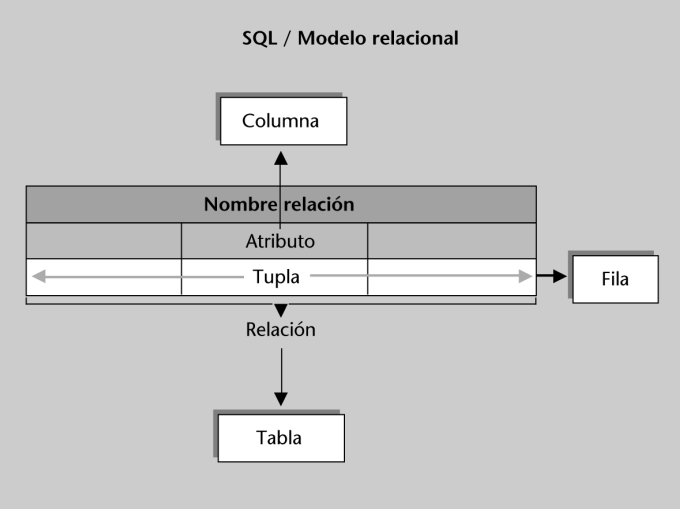
1) A principios de los años setenta, los laboratorios de investigación Santa Teresa de IBM empezaron a trabajar en el proyecto System R. El objetivo de este proyecto era implementar un prototipo de SGBD relacional y, por lo tanto, necesitaban también investigar en el campo de los lenguajes de BD relacionales. A mediados de los años setenta, el proyecto de IBM dio un primer lenguaje llamado *SEQUEL* (*Structured English QUEry Language*), que por razones legales más adelante se denominó *SQL* (*Structured Query Language*). A finales de los setenta y principios de los ochenta, una vez finalizado el proyecto System R, IBM y otras empresas empezaron a utilizar SQL en sus SGBD relacionales, y así este lenguaje adquirió una gran popularidad.

2) En 1982, el ANSI (American National Standards Institute) encargó a uno de sus comités (X3H2) la definición de un lenguaje de BD relacionales. Este comité, después de evaluar diferentes lenguajes, y ante la aceptación comercial de SQL, escogió como lenguaje estándar un lenguaje basado en SQL casi en su totalidad. SQL se convirtió oficialmente en el lenguaje estándar de ANSI en 1986, y de ISO (International Standards Organization) en 1987. También ha sido adoptado como lenguaje estándar por FIPS (Federal Information Processing Standard), Unix X/Open y SAA (Systems Application Architecture) de IBM.

3) En 1989, el estándar fue objeto de una revisión y una ampliación que dieron lugar al lenguaje que se conoce con el nombre de SQL1 o SQL:1989. En 1992, el estándar volvió a ser revisado y ampliado considerablemente para cubrir carencias de la versión anterior. Esta nueva versión de SQL se conoce con el nombre de SQL2 o SQL:1992. En 1999, apareció la esperada versión SLQ3 o SQL:1999, en la que, entre otras cosas, se definía formalmente uno de los componentes lógicos de control más útiles: los disparadores.

En este módulo didáctico, aunque aparezca sólo la sigla SQL, siempre nos referiremos a la **última versión**, ya que tiene como subconjunto todas las anteriores y, por lo tanto, todo lo que era válido en la anterior lo continuará siendo en la siguiente. Sólo especificaremos el año de una versión de SQL cuando, por ejemplo, queramos enfatizar que fue concretamente en ésta en la que se hizo una aportación determinada.

El modelo relacional tiene como estructura de almacenamiento de los datos las relaciones. La intensión o esquema de una relación consiste en el nombre que hemos dado a la relación y a un conjunto de atributos. La extensión de una relación es un conjunto de tuplas. Al trabajar con SQL esta nomenclatura cambia, como podemos ver en la figura siguiente:



- Hablaremos de **tablas** en lugar de relaciones.
- Hablaremos de **columnas** en lugar de atributos.
- Hablaremos de **filas** en lugar de tuplas.

Sin embargo, aunque la nomenclatura utilizada sea diferente, los conceptos son los mismos.

Con SQL se puede definir y manipular una BD relacional. Acto seguido veremos, aunque sólo de manera introductoria, cómo se podría hacer tal cosa:

1. Habría que crear una tabla que contuviera los datos de los productos de nuestra empresa:

```
CREATE TABLE productos
  (codigo_producto INTEGER,
  nombre_producto CHAR(20),
  tipo CHAR(20),
  descripcion CHAR(50),
  precio REAL,
  PRIMARY KEY (codigo_producto));
```

Notación

Aunque no es una convención de SQL estándar, la mayoría de sistemas relacionales co-merciales requieren que una sentencia finalice con ;.

2. Insertar un producto en la tabla creada anteriormente:

```
INSERT INTO productos
VALUES 1250,"LENA","Tabla","Diseño Joan Pi. Año1920.",2500.0);;
```

Valores de la fila

3. Consultar qué productos de nuestra empresa son sillas:

```
                Columnas seleccionadas
SELECT  codigo_producto,nombre_producto
FROM productos
WHERE tipo = "silla";
      Filas seleccionadas
```

Y muchas más cosas que iremos viendo, punto por punto, en los siguientes apartados.

Fijémonos en la estructura de todo lo que hemos hecho hasta ahora con SQL. Las operaciones de SQL se llaman **sentencias** y están formadas por diferentes partes que denominamos **cláusulas**, tal como podemos ver en el ejemplo siguiente:

Sentencia:

```
SELECT codigo_producto, num_producto, tipo   — Cláusula
FROM productos                               — Cláusula
WHERE precio > 1000.0;                       — Cláusula
```

Esta consulta muestra el código, el nombre y el tipo de los productos que cuestan más de 1.000 euros.

Los dos primeros apartados de este módulo tratan de un tipo de SQL llamado **SQL interactivo**, que permite acceder directamente a una BD relacional. En el primer apartado, definiremos las llamadas **sentencias de definición**, en las que crearemos la BD, las tablas que la compondrán, los dominios y las aserciones que queramos. En el segundo apartado, aprenderemos a **manipular** la BD, introduciendo valores en las filas, modificándolos o borrándolos, o haciendo consultas.

Pero muchas veces querremos acceder a la BD desde una aplicación hecha en un lenguaje de programación cualquiera, que nos ofrezca mucha más potencia fuera del entorno de las BD. Para utilizar SQL desde un lenguaje de programación, necesitaremos sentencias especiales que nos permitan distinguir entre

las instrucciones del lenguaje de programación y las sentencias de SQL. La idea es que, trabajando básicamente con un lenguaje de programación anfitrión, se puede acoger a SQL como si fuera un huésped. Por este motivo, este tipo de SQL se conoce con el nombre de **SQL hospedado**. Para trabajar con SQL hospedado, necesitamos un precompilador que separe las sentencias del lenguaje de programación de las del lenguaje de BD. Una alternativa a esta manera de trabajar son las rutinas **SQL/CLI*** (SQL/*Call-Level Interface*), que resuelve también el problema de acceder a SQL desde un lenguaje de programación y no necesita precompilador.

*** Las rutinas SQL/CLI se añadieron al estándar SQL:1992 en 1995.**

Antes de empezar a conocer el lenguaje, conviene añadir un último comentario. Aunque SQL sea el lenguaje estándar para BD relacionales y haya sido sobradamente aceptado por los sistemas relacionales comerciales, no es capaz de reflejar toda la teoría del modelo relacional establecida por E. F. Codd, como iremos viendo a medida que profundicemos en el lenguaje.

Los sistemas relacionales comerciales y los investigadores de BD son una referencia muy importante para mantener el estándar actualizado. En estos momentos la última versión de la que se dispone es SQL: 2008. Pero con anterioridad habían aparecido, respectivamente, SQL: 2003 y SQL: 2006. Las principales novedades de estos últimos estándares tienen que ver con como utilizar SQL en conjunción con XML. Y está claro que continuarán apareciendo versiones nuevas a medida que se vayan incorporando mejoras sobre el lenguaje SQL. Por eso, en informática en general, y particularmente en BD, hay que estar siempre al día, y es muy importante mantener el hábito de leer publicaciones periódicas que nos informen y nos mantengan al corriente de las novedades.

Objetivos

Una vez finalizado el estudio de los materiales didácticos de este módulo, tendréis las herramientas indispensables para alcanzar los siguientes objetivos:

1. Conocer los conceptos básicos del lenguaje estándar SQL.

2. Definir una BD relacional, incluyendo dominios.

3. Saber introducir, borrar y modificar datos.

4. Ser capaz de plantear cualquier tipo de consulta a la BD.

5. Saber utilizar sentencias de control.

6. Conocer los principios básicos de la utilización de SQL desde un lenguaje de programación.

1. Sentencias de definición de datos

Para poder trabajar con BD relacionales, lo primero que tenemos que hacer es definirlas. Veremos las sentencias de SQL estándar para crear y borrar una BD relacional y para crear, borrar y modificar las diferentes tablas que la componen. En este apartado, también veremos cómo se definen los dominios y las aserciones (restricciones). 

La sencillez y la homogeneidad de SQL hacen que:

1. Para crear BD, tablas, dominios y aserciones se utilice la **sentencia `CREATE`**.
2. Para modificar tablas y dominios se utilice la **sentencia `ALTER`**.
3. Para borrar BD, tablas, dominios y aserciones se utilice la **sentencia `DROP`**.

La adecuación de estas sentencias en cada caso nos dará diferencias que iremos perfilando al hacer la descripción individual de cada una.

Para ilustrar la aplicación de las sentencias de SQL que iremos viendo, utilizaremos **una BD de ejemplo** muy sencilla de una pequeña empresa con sede en Barcelona, Gerona, Lérida y Tarragona, que se encarga de desarrollar proyectos informáticos. La información que nos interesará almacenar de esta empresa, que llamaremos *BDUOC*, será la siguiente:

1. De los empleados que trabajan en la empresa, querremos saber el código de empleado, el nombre y apellido, el sueldo, el nombre y la ciudad de su departamento y el número de proyecto al cual están asignados.

2. De los diferentes departamentos en que está estructurada la empresa, nos interesa conocer el nombre, la ciudad donde se encuentran y el teléfono. Habrá que tener en cuenta que un departamento con el mismo nombre puede estar en ciudades diferentes y que en una misma ciudad puede haber departamentos con nombres diferentes.

3. De los proyectos informáticos que se desarrollen, querremos saber el código, el nombre, el precio, la fecha de inicio, la fecha prevista de finalización, la fecha real de finalización y el código de cliente para quien se desarrolla.

4. De los clientes para quien trabaja la empresa, querremos saber el código de cliente, el nombre, el NIF, la dirección, la ciudad y el teléfono.

1.1. Creación y borrado de una BD relacional

SQL estándar no dispone de ninguna sentencia de creación de BD. La idea es que una BD no es nada más que un conjunto de tablas y, por lo tanto, las sentencias que SQL nos ofrece se concentran en la creación, la modificación y el borrado de estas tablas.

La instrucción `CREATE DATABASE`

Muchos de los sistemas rela-cionales comerciales (como es el caso de Informix, DB2, SQL Server y otros) han incorporado sentencias de creación de BD con la siguiente sintaxis:

```
CREATE DATABASE
<nombre_BD>;
```

En cambio, disponemos de una sentencia más potente que la de creación de BD: la **sentencia de creación de esquemas** llamada **`CREATE SCHEMA`**.

Con la creación de esquemas podemos agrupar un conjunto de elementos de la BD que son propiedad de un usuario. Esta sentencia se ofrece desde la primera versión del estándar, pero no con la sintaxis que veremos en este módulo. La sintaxis de SQL:1992 de esta sentencia es la que tenéis a continuación:

```
CREATE SCHEMA {<nombre_esquema>|AUTHORIZATION <usuario>}
[<lista_elementos_esquema>];
```

La **nomenclatura utilizada en esta sentencia** y, de aquí en adelante, es la siguiente:

- Las palabras en negrita son palabras reservadas del lenguaje.
- La notación [...] quiere decir que lo que hay entre los corchetes se podría poner o no.
- La notación {A|...|B} quiere decir que tenemos que escoger entre todas las opciones que hay entre las llaves. Pero tenemos que poner una obligatoriamente.
- La notación <A> quiere decir que A es un elemento pendiente de definición.

La sentencia de creación de esquemas permite que un conjunto de tablas (`lista_elementos_esquema`) se pueda agrupar bajo un mismo nombre (`nombre_esquema`) y que tengan un propietario (`usuario`). Aunque los parámetros de la sentencia `CREATE SCHEMA` sean opcionales, como mínimo se tiene que dar o bien el nombre del esquema, o bien el nombre del usuario propietario de la BD.

Si sólo especificamos al usuario, éste será el escogido como nombre del esquema.

La creación de esquemas puede hacer mucho más que agrupar tablas, porque `lista_elementos_esquema` puede ser, además de tablas y dominios, otros componentes.

Para borrar una BD encontramos el mismo problema que para crearla. SQL nos ofrece sólo la **sentencia de borrado de esquemas** **`DROP SCHEMA`**, que tiene la siguiente sintaxis:

```
DROP SCHEMA <nombre_esquema> {RESTRICT|CASCADE};
```

La sentencia `DROP DATABASE`

Muchos de los sistemas relacionales comerciales (como Informix, DB2, SQL Server y otros) han incorporado sentencias de borrado de BD con la siguiente sintaxis:

```
DROP DATABASE
<nombre_BD>;
```

En que tenemos lo siguiente:

- La opción de borrado de esquemas **`RESTRICT`** hace que el esquema sólo se pueda borrar si no contiene ningún elemento.
- La opción **`CASCADE`** borra el esquema aunque no esté completamente vacío.

1.2. Creación de tablas

Como ya hemos visto, la estructura de almacenamiento de los datos del modelo relacional son las tablas. Para **crear una tabla** hay que utilizar la **sentencia** **`CREATE TABLE`**. Veamos el formato:

```
CREATE TABLE <nombre_tabla>
   ( <definición_columna>
   [, <definición_columna>...]
   [, <restricciones_tabla>]
   );
```

Donde **`definición_columna`** es:

```
<nombre_columna> {<tipo_datos>|<dominio>} [<def_defecto>] [<restricciones_columna>]
```

El proceso que hay que seguir para crear una tabla es el siguiente:

1) Decidir el nombre de la tabla (`nombre_tabla`).

2) Dar nombre a cada uno de los atributos que formarán las columnas de la tabla (`nombre_columna`).

3) Asignar a cada una de las columnas un tipo de datos predefinido o bien un dominio definido por el usuario. También se puede dar definiciones por defecto y restricciones de columna.

4) Una vez definidas las columnas, sólo habrá que dar las restricciones de tabla.

1.2.1. Tipos de datos

Para cada columna tenemos que escoger entre algún dominio definido por el usuario o alguno de los tipos de datos predefinidos que se describen a continuación:

Tipos de datos predefinidos	
Tipo de datos	**Descripción**
`CHARACTER (longitud)`	Cadenas de caracteres de longitud fija
`CHARACTER VARYING (longitud)`	Cadenas de caracteres de longitud variable
`BIT (longitud)`	Cadenas de bits de longitud fija
`BIT VARYING (longitud)`	Cadenas de bits de longitud variable
`NUMERIC (precisión, escala)`	Números decimales con tantos dígitos como indique la precisión y tantos decimales como indique la escala
`DECIMAL (precisión, escala)`	Números decimales con tantos dígitos como indique la precisión y tantos decimales como indique la escala
`INTEGER`	Números enteros
`SMALLINT`	Números enteros pequeños
`REAL`	Números con coma flotante con precisión predefinida
`FLOAT (precisión)`	Números con coma flotante con la precisión especificada
`DOUBLE PRECISION`	Números con coma flotante con más precisión predefinida que la del tipo `REAL`
`DATE`	Fechas. Se componen de: `YEAR año, MONTH mes, DAY día`
`TIME`	Horas. Se componen de: : `HOUR hora, MINUT minutos, SECOND segundos`
`TIMESTAMP`	Fechas y horas. Se componen de: `YEAR año, MONTH mes, DAY día, HOUR hora, MINUT minutos, SECOND segundos`

Los tipos de datos `NUMERICO` y `DECIMAL`

`NUMERICO` y `DECIMAL` se describen igual y se pueden utilizar tanto el uno como el otro para definir números decimales.

El tratamiento del tiempo

SQL define la siguiente nomenclatura para trabajar con el tiempo:

`YEAR`	(0001..9999)
`MONTH`	(01..12)
`DAY`	(01..31)
`HOUR`	(00..23)
`MINUT`	(00..59)
`SECOND`	(00..59.precisión)

De todas maneras, los SGBD comerciales disponen de diferentes formatos, entre los cuales podemos escoger cuándo tenemos que trabajar con columnas temporales.

Ejemplos de asignaciones de columnas

Veamos algunos ejemplos de asignaciones de columnas en los tipos de datos predefinidos `DATE`, `TIME` y `TIMESTAMP`:

- La columna `fecha_nacimiento` podría ser de tipo `DATE` y podría tener como valor '1978-12-25'.
- La columna `inicio_partido` podría ser de tipo `TIME` y podría tener como valor '17:15:00.000000'.
- La columna `entrada_trabajo` podría ser de tipo `TIMESTAMP` y podría tener como valor '1998-7-8 9:30:05'.

Dominios definidos por el usuario

La sentencia `CREATE DOMAIN` de SQL:1992 hay pocos SGBD comerciales que permitan usarla. De todos modos, SQL:1999 ofrece los `USER DEFINED TYPES` como una nueva posibilidad para la creación de dominios definidos para el usuario más potente que la propuesta por SQL:1992.

1.2.2. Creación, modificación y borrado de dominios

Además de los dominios dados por los tipos de datos predefinidos, SQL nos ofrece la posibilidad de trabajar con dominios definidos por el usuario.

Para **crear un dominio** hay que utilizar la sentencia **CREATE DOMAIN**:

```
CREATE DOMAIN <nombre_dominio> [AS] <tipo_datos>
            [<def_defecto>] [<restricciones_dominio>];
```

donde **restricciones_dominio** tiene el siguiente formato:

```
CONSTRAINT [<nombre_restricción>] CHECK (<condiciones>)
```

Explicaremos la construcción de condiciones más adelante, en el subapartado 2.5, cuando hablemos de cómo se hacen consultas en una BD. Veremos `def_defecto` en el subapartado 1.2.3 de este módulo. Aunque dar nombre a la restricción sea opcional, normalmente se le da para poderla referenciar posteriormente.

Creación de un dominio en BDUOC

Si quisiéramos definir un dominio para las ciudades donde se encuentran los departamentos de la empresa BDUOC, haríamos:

```
CREATE DOMAIN dom_ciudades AS CHAR(20)
  CONSTRAINT ciudades_validas
  CHECK (VALUE IN ("Barcelona", "Tarragona", "Lerida", "Gerona"));
```

De esta manera, cuando definimos la columna `ciudad` dentro de la tabla `departamentos` no se tendrá que decir que es de tipo `CHAR(20)`, sino de tipo `dom_ciudades`. Eso nos tendría que asegurar, según el modelo relacional, sólo hacer operaciones sobre la columna `ciudad` con otras columnas que tengan este mismo dominio definido por el usuario, pero SQL no nos ofrece herramientas para asegurar que las comparaciones que hacemos sean entre los mismos dominios definidos por el usuario.

Por ejemplo, si tenemos una columna con los nombres de los empleados definida sobre el tipo de datos `CHAR(20)`, SQL nos permite compararla con la columna `ciudad`, aunque semánticamente no tenga sentido. En cambio, según el modelo relacional, esta comparación no se tendría que haber permitido.

Para **borrar un dominio definido por el usuario**, hay que utilizar la **sentencia DROP DOMAIN**, que tiene este formato:

```
DROP DOMAIN <nombre_dominio> {RESTRICT|CASCADE};
```

En este caso, tenemos que:

- La opción de borrado de dominios **RESTRICT** hace que el dominio sólo se pueda borrar si no se utiliza en ningún sitio.
- La opción **CASCADE** borra el dominio aunque esté referenciado y pone el tipo de datos del dominio allí donde se utilizaba.

Borrar un dominio de BDUOC

Si quisiéramos borrar el dominio que hemos creado antes para las ciudades donde se encuentran los departamentos de la empresa BDUOC, haríamos:

```
DROP DOMAIN dom_ciudades RESTRICT;
```

En este caso, nos aseguraríamos de que ninguna columna está definida sobre `dom_ciudades` antes de borrar el dominio.

Para **modificar un dominio**, se utiliza la **sentencia `ALTER DOMAIN`**. Veamos el formato:

```
ALTER DOMAIN <nombre_dominio> {<acción_modificar_dominio>|
                      <acción_modificar_restricción_dominio>};
```

Donde tenemos lo siguiente:

- **`acción_modificar_dominio`** puede ser:

```
SET {<def_defecto>|DROP DEFAULT}
```

- **`acción_modificar_restricción_dominio`** puede ser:

```
ADD {<restricción_dominio>|DROP CONSTRAINT <nombre_restricción>}
```

Modificar un dominio en BDUOC

Si quisiéramos añadir una ciudad nueva (Mataró) en el dominio que hemos creado antes para las ciudades donde se encuentran los departamentos de la empresa BDUOC, haríamos:

```
ALTER DOMAIN dom_ciudades DROP CONSTRAINT ciudades_validas;
```

Con ello, hemos eliminado la restricción de dominio antigua. Y ahora tenemos que introducir la nueva restricción:

```
ALTER DOMAIN dom_ciudades ADD CONSTRAINT ciudades_validas
  CHECK (VALUE IN ("Barcelona","Tarragona", "Lerida",
                "Gerona", "Mataro"));
```

1.2.3. Definiciones por defecto

Ya hemos visto en otros módulos la importancia de los valores nulos y su inevitable aparición como valores de las BD.

La opción **`def_defecto`** nos permite especificar qué nomenclatura queremos dar a nuestros valores por omisión.

Ejemplo

Para un empleado del cual todavía no se ha decidido cuánto ganará, se puede escoger que, de momento, tenga un sueldo de 0 euros (`DEFAULT 0.0`), o bien que tenga un sueldo con un valor nulo (`DEFAULT NULL`).

Hay que tener en cuenta, sin embargo, que si escogemos la opción **DEFAULT NULL** la columna para la cual daremos la definición por defecto de valor nulo tendría que admitir valores nulos.

La opción **DEFAULT** tiene el siguiente formato:

```
DEFAULT {<literal>|<función>|NULL}
```

La posibilidad más utilizada y la opción por defecto si no especificamos nada es la palabra reservada **NULL**. Pero también podemos definir nuestro propio literal, o bien recurrir a una de las funciones que aparecen en la siguiente tabla:

Función	Descripción
{USER|CURRENT_USER}	Identificador del usuario actual
SESSION_USER	Identificador del usuario de esta sesión
SYSTEM_USER	Identificador del usuario del sistema operativo
CURRENT_DATE	Fecha actual
CURRENT_TIME	Hora actual
CURRENT_TIMESTAMP	Fecha y hora actuales

1.2.4. Restricciones de columna

En cada una de las columnas de la tabla, una vez les hemos dado un nombre y hemos definido el dominio, podemos imponer ciertas restricciones que siempre se tendrán que cumplir. Las restricciones que se pueden dar son las que aparecen en la tabla que tenéis a continuación:

Restricciones de columna	
Restricción	**Descripción**
NOT NULL	La columna no puede tener valores nulos
UNIQUE	La columna no puede tener valores repetidos. Es una clave alternativa
PRIMARY KEY	La columna no puede tener valores repetidos ni nulos. Es la clave primaria
REFERENCES <nombre_tabla> [(<nombre_columna>)]	La columna es la clave foránea de la columna de la tabla especificada
CONSTRAINT [<nombre_restricción>] CHECK (<condiciones>)	La columna tiene que cumplir las condiciones especificadas

1.2.5. Restricciones de tabla

Una vez hemos dado un nombre, hemos definido un dominio y hemos impuesto ciertas restricciones para cada una de las columnas, podemos aplicar

restricciones sobre toda la tabla, las cuales siempre se tendrán que cumplir. Las restricciones que se pueden dar son las siguientes:

Restricciones de tabla	
Restricción	**Descripción**
`UNIQUE (<nombre_columna> [, <nombre_columna>...])`	El conjunto de las columnas especificadas no puede tener valores repetidos. Es una clave alternativa.
`PRIMARY KEY (<nombre_columna> [, <nombre_columna>...])`	El conjunto de las columnas especificadas no puede tener valores nulos ni repetidos. Es una clave primaria.
`FOREIGN KEY (<nombre_columna> [, <nombre_columna>...]) REFERENCES <nombre_tabla> [(<nombre_columna2> [, <nombre_columna2>...])]`	El conjunto de las columnas especificadas es una clave foránea que referencia la clave primaria formada por el conjunto de las columnas2 de la tabla dada. Si las columnas y las columnas2 se llaman exactamente igual, entonces no habría que poner columnas2.
`CONSTRAINT [<nombre_restricción>] CHECK (<condiciones>)`	La tabla tiene que cumplir las condiciones especificadas.

1.2.6. Modificación y borrado de claves primarias con claves foráneas que hacen referencia a ellas

Hemos visto tres políticas aplicables a los casos de borrado y modificación de filas que tienen una clave primaria referenciada por claves foráneas. Estas políticas eran la restricción, la actualización en cascada y la anulación..

SQL nos ofrece la posibilidad de especificar, definiendo una clave foránea, qué política queremos seguir. Veamos el formato:

```
CREATE TABLE <nombre_tabla>
  ( <definición_columna>
    [, <definición_columna>...]
    [, <restricciones_tabla>]
  );
```

En el cual, una de las restricciones de tabla era la definición de claves foráneas, que tiene el siguiente formato:

```
FOREIGN KEY <clave_foránea> REFERENCES <nombre_tabla> [(<clave_primaria>)]
[ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
[ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
```

Aquí, `NO ACTION` corresponde a la política de restricción; `CASCADE`, a la actualización en cascada, y `SET NULL` sería la anulación. `SET DEFAULT` se podría considerar una variante de `SET NULL`, donde en lugar de valores nulos se puede poner el valor especificado por defecto.

1.2.7. Aserciones

Una aserción es una restricción general que hace referencia a una o más columnas de una o más tablas. Para **definir una aserción** se usa la sentencia **`CREATE ASSERTION`** y tiene el siguiente formato:

```
CREATE ASSERTION <nombre_aserción> CHECK (<condiciones>);
```

Crear una aserción en BDUOC

Creamos una aserción sobre la BD BDUOC que nos asegure que no hay ningún empleado con un `sueldo` superior a `80000` asignado al proyecto `SALSA`:

```
CREATE ASSERTION restriccion1 CHECK (NOT EXISTS (SELECT *
                        FROM proyectos p, empleados e
                        WHERE p.codigo_proy = e.num_proy
                              and e.sueldo > 80000.0
                              and p.nombre_proy = "SALSA"));
```

Para **borrar una aserción**, hay que utilizar la sentencia **`DROP ASSERTION`**, que presenta este formato:

```
DROP ASSERTION <nombre_aserción>;
```

Borrar un aserción en BDUOC

Para borrar la aserción `restriccion1`, utilizaríamos la sentencia `DROP ASSERTION` de la siguiente manera:

```
DROP ASSERTION restriccion1;
```

Aserciones

Aunque SQL ofrezca la sentencia `CREATE ASSERTION`, no existen productos comerciales que permitan usarla. La alternativa a las aserciones son los disparadores (*triggers*). Los disparadores fueron introducidos con SQL:1999 y, a diferencia de las aserciones, la gran mayoría de productos comerciales los ofrecen.

1.3. Modificación y borrado de tablas

Para **modificar una tabla** hay que utilizar la sentencia **`ALTER TABLE`**. Veamos el formato:

```
ALTER TABLE <nombre_tabla> {<acción_modificar_columna>|
                        <acción_modificar_restricción_tabla>};
```

En este caso, tenemos que:

- **`acción_modificar_columna`** puede ser:

```
{ADD [COLUMN] <nombre_columna> <def_columna> |
  ALTER [COLUMN] <nombre_columna> {SET <def_defecto>|DROP DEFAULT}|
  DROP [COLUMN] <nombre_columna> {RESTRICT|CASCADE}}
```

- **acción_modificar_restricción_tabla** puede ser:

```
{ADD <def_restricción>|
  DROP CONSTRAINT <nombre_restricción> {RESTRICT|CASCADE}}
```

Si queremos modificar una tabla, es que queremos hacer una de las sigiuientes operaciones:

1. Añadir una columna (`ADD <nombre_columna>`).
2. Modificar las definiciones por defecto de la columna (`ALTER <nombre_columna>`).
3. Borrar la columna (`DROP <nombre_columna>`).
4. Añadir alguna nueva restricción de tabla (`ADD <def_restricción>`).
5. Borrar alguna restricción de tabla (`DROP CONSTRAINT <nombre_restricción>`).

Para **borrar una tabla**, hay que utilizar la sentencia **DROP TABLE**:

```
DROP TABLE <nombre_tabla> {RESTRICT|CASCADE};
```

En este caso, tenemos que:

- Si utilizamos la opción **RESTRICT** la tabla no se borrará si está referenciada, por ejemplo, desde alguna otra tabla o vista.
- Si usamos la opción **CASCADE**, todo lo que referencie en la tabla se borrará con ella.

1.4. Creación y borrado de vistas

La arquitectura ANSI/SPARC distingue tres niveles, que se describen en el esquema conceptual, el esquema interno y los esquemas externos. Hasta ahora, mientras creábamos las tablas de la BD, íbamos describiendo el esquema conceptual. Para describir los diferentes esquemas externos utilizamos el concepto de vista de SQL.

Para **crear una vista** es necesario utilizar la **sentencia CREATE VIEW**. Veamos su formato:

```
CREATE VIEW <nombre_vista> [(lista_columnas)] AS (consulta)
            [WITH CHECK OPTION];
```

Lo primero que tenemos que hacer para crear una vista es decidir qué nombre le queremos poner (`nombre_vista`). Si queremos cambiar el nombre de las

columnas, o bien poner nombre a alguna que en principio no tenía, lo podemos hacer en `lista_columnas`. Y ya sólo nos quedará definir la consulta que formará nuestra vista.

Las vistas no existen realmente como un conjunto de valores almacenados en la BD, sino que son tablas ficticias, denominadas *derivadas* (no materializadas). Se construyen a partir de tablas reales (materializadas) almacenadas en la BD, y conocidas con el nombre de *tablas básicas* (o tablas de base). La no-existencia real de las vistas hace que puedan ser actualizables o no.

Creación de una vista en BDUOC

Creamos una vista sobre la BD BDUOC que nos dé para cada cliente el número de proyectos que tiene encargados el cliente en cuestión.

```
CREATE VIEW proyectos_por_cliente (codigo_cli, numero_proyectos) AS
(SELECT c.codigo_cli, COUNT(*)
 FROM proyectos p, clientes c
 WHERE p.codigo_cliente = c.codigo_cli
 GROUP BY c.codigo_cli);
```

Si tuviésemos las siguientes extensiones:

- Tabla `clientes`:

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Gerona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lérida	973.23.45.67

- Tabla `proyectos`:

proyectos						
codigo_proy	**nombre_proy**	**precio**	**fecha_inicio**	**fecha_prev_fin**	**fecha_fin**	**codigo_cliente**
1	GESCOM	1,0E+6	1-1-98	1-1-99	NULL	10
2	PESCI	2,0E+6	1-10-96	31-3-98	1-5-98	10
3	SALSA	1,0E+6	10-2-98	1-2-99	NULL	20
4	TINELL	4,0E+6	1-1-97	1-12-99	NULL	30

Y mirásemos la extensión de la vista `proyectos_por_clientes`, veríamos lo que encontramos en el margen.

proyectos_por_clientes	
codigo_cli	**numero_proyectos**
10	2
20	1
30	1

En las vistas, además de hacer consultas, podemos insertar, modificar y borrar filas.

Actualización de vistas en BDUOC

Si alguien insertase en la vista `proyectos_por_cliente`, los valores para un nuevo cliente 60 con tres proyectos encargados, encontraríamos que estos tres proyectos tendrían que figurar realmente en la tabla `proyectos` y, por lo tanto, el SGBD los debería insertar con la

información que tenemos, que es prácticamente inexistente. Veamos gráficamente cómo quedarían las tablas después de esta hipotética actualización, que no llegaremos a hacer nunca, ya que iría en contra de la teoría del modelo relacional:

- Tabla `clientes`

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Gerona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lérida	973.23.45.67
60	NULL	NULL	NULL	NULL	NULL

- Tabla proyectos:

proyectos						
codigo_proy	**nombre_proy**	**precio**	**fecha_inicio**	**fecha_prev_fin**	**fecha_fin**	**codigo_cliente**
1	GESCOM	1,0E+6	1-1-98	1-1-99	NULL	10
2	PESCI	2,0E+6	1-10-96	31-3-98	1-5-98	10
3	SALSA	1,0E+6	10-2-98	1-2-99	NULL	20
NULL	NULL	NULL	NULL	NULL	NULL	60
NULL	NULL	NULL	NULL	NULL	NULL	60
NULL	NULL	NULL	NULL	NULL	NULL	60

El SGBD no puede actualizar la tabla básica `clientes` si sólo sabe la clave primaria, y todavía menos la tabla básica `proyectos` sin la clave primaria; por lo tanto, esta vista no sería actualizable.

En cambio, si definimos una vista para saber los clientes que tenemos en Barcelona o en Gerona, haríamos:

```
CREATE VIEW clientes_Barcelona_Gerona AS
(SELECT *
 FROM clientes
 WHERE ciudad IN ('Barcelona', 'Gerona'))
WHITH CHECK OPTION;
```

Si queremos asegurarnos de que se cumpla la condición de la cláusula `WHERE`, debemos poner la opción `WHITH CHECK OPTION`. Si no lo hiciésemos, podría ocurrir que alguien incluyese en la vista `clientes_Barcelona_Gerona` a un cliente nuevo con el código 70, de nombre JMB, con el NIF 36.788.224-C, la dirección en NULL, la ciudad Lérida y el teléfono NULL.

Si consultásemos la extensión de la vista `clientes_Barcelona_Gerona`, veríamos:

clientes_Barcelona_Gerona					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Gerona	972.23.57.21

Esta vista sí podría ser actualizable. Podríamos insertar un nuevo cliente con código 50, de nombre CEA, con el NIF 38.226.777-D, con la dirección París 44, la ciudad Barcelona y el te-

léfono 93.422.60.77. Después de esta actualización, en la tabla básica `clientes` encontraríamos, efectivamente:

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragón 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Gerona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lérida	973.23.45.67
50	CEA	38.226.777-D	París, 44	Barcelona	93.442.60.77

Para borrar una vista es preciso utilizar la **sentencia `DROP VIEW`**, que presenta el formato:

```
DROP VIEW <nombre_vista> (RESTRICT|CASCADE);
```

Si utilizamos la opción **`RESTRICT`**, la vista no se borrará si está referenciada, por ejemplo, por otra vista. En cambio, si ponemos la opción **`CASCADE`**, todo lo que referencie a la vista se borrará con ésta.

Borrar una vista en BDUOC

Para borrar la vista `clientes_Barcelona_Gerona`, haríamos lo siguiente:

```
DROP VIEW clientes_Barcelona_Gerona RESTRICT;
```

1.5. Definición de la BD relacional BDUOC

Orden de creación

Antes de crear una tabla con una o más claves foráneas, se han de haber creado las tablas que tienen como clave primaria las referenciadas por las foráneas.

Veamos cómo se crearían las tablas de la BD BDUOC:

```
CREATE TABLE clientes (codigo_cli INTEGER,
  nombre_cli CHAR(30) NOT NULL,
  nif CHAR(12),
  direccion CHAR(30),
  ciudad CHAR(20),
  telefono CHAR(12) DEFAULT NULL,
  PRIMARY KEY(codigo_cli), UNIQUE(nif)
);

CREATE TABLE departamentos
  (nombre_dpt CHAR(20) PRIMARY KEY*,
  ciudad_dpt CHAR(20),
  telefono CHAR(12) DEFAULT NULL,
  PRIMARY KEY (nombre_dpt, ciudad_dpt)
);
```

La mayoría de productos comerciales no acepta ni la "ñ" ni los acentos.

*** Tenemos que escoger restricción de tabla porque la clave primaria está compuesta por más de un atributo.**

```
CREATE TABLE proyectos
  (codigo_proy INTEGER,
  nombre_proy CHAR(20),
  precio REAL,
  fecha_inicio DATE,
  fecha_prev_fin DATE,
  fecha_fin DATE DEFAULT NULL,
  codigo_cliente INTEGER,
  PRIMARY KEY (codigo_proy),
  FOREIGN KEY codigo_cliente REFERENCES clientes(codigo_cli),
  CHECK (fecha_inicio < fecha_prev_fin),
  CHECK (fecha_inicio < fecha_fin)
);

CREATE TABLE empleados
  (codigo_empl INTEGER,
  nombre_empl CHAR(20),
  apellido_empl CHAR(20),
  sueldo REAL CHECK(sueldo > 7000.0),
  nombre_dpt CHAR(20),
  ciudad_dpt CHAR(20),
  num_proy INTEGER,
  PRIMARY KEY (codigo_empl),
  FOREIGN KEY (nombre_dpt, ciudad_dpt)
    REFERENCES departamentos(num_dpt, ciudad_dpt),
  FOREIGN KEY (num_proy) REFERENCES proyectos(codigo_proy)
);
```

Al crear una tabla, vemos que muchas restricciones se pueden imponer de dos maneras: como restricciones de columna o como restricciones de tabla. Por ejemplo, cuando queremos decir cuál es la clave primaria de una tabla tenemos ambas posibilidades. Eso es debido a la flexibilidad de SQL:

- En caso de que la restricción haga referencia a un solo atributo, podemos escoger la posibilidad que nos guste más.

- En el caso de la tabla departamentos, tenemos que escoger forzosamente la opción de restricciones de tabla, porque la clave primaria está compuesta por más de un atributo.

En general, por homogeneidad, lo pondremos todo como restricciones de tabla, excepto `NOT NULL` y `CHECK` cuando haga referencia a una sola columna.

2. Sentencias de manipulación de datos

Una vez creada la BD con sus tablas, habrá que poder insertar, modificar y borrar los valores de las filas de las tablas. Para poder hacer eso, SQL nos ofrece las siguientes sentencias: **INSERT**, para insertar; **UPDATE**, para modificar, y **DELETE**, para borrar. Una vez hemos insertado valores en las tablas, tenemos que poder consultarlos. La sentencia para hacer consultas a una BD con SQL es **SELECT FROM**. Veamos, acto seguido, estas sentencias.

2.1. Inserción de filas en una tabla

Antes de poder consultar los datos de una BD, hay que introducir los datos con la sentencia **INSERT INTO VALUES**, que tiene el siguiente formato:

```
INSERT INTO <nombre_tabla> [(<columnas>)]
  {VALUES ({<v1>|DEFAULT|NULL}, ..., {<vn>|DEFAULT|NULL}) |<consulta>};
```

Inserción de múltiples filas

Para insertar más de una fila con una sola sentencia tenemos que obtener los valores como resultado de una consulta hecha en una o más tablas.

Los valores `v1`, `v2`..., `vn` se tienen que corresponder exactamente con las columnas que hemos dicho que tendríamos con el `CREATE TABLE`, y tienen que estar en el mismo orden, a no ser que las volvamos a poner a continuación del nombre de la tabla. En este último caso, los valores se deben disponer de manera coherente con el nuevo orden que hemos impuesto. Podría ser que quisiéramos que algunos valores a insertar fueran valores por omisión, definidos previamente con la opción `DEFAULT`. Entonces pondríamos la palabra reservada `DEFAULT`. Si se trata de introducir valores nulos también podemos usar la palabra reservada `NULL`.

Inserción de una fila en BDUOC

La manera de insertar un cliente en la tabla clientes de la BD BDUOC es:

```
INSERT INTO clientes
VALUES (10, "ECIGSA", "37.248.573-C", "ARAGON 242", "Barcelo-
na", DEFAULT);
```

o bien:

```
INSERT INTO clientes(nif, nombre_cli, codigo_cli, telefono,
direccion, ciudad)
VALUES("37.248.573-C", "ECIGSA", 10, DEFAULT,
       "ARAGON 242", "Barcelona");
```

2.2. Borrado de filas de una tabla

Para **borrar valores de algunas filas de una tabla**, podemos utilizar la **sentencia `DELETE FROM WHERE`**. Su formato es el siguiente:

```
DELETE FROM <nombre_tabla>
[WHERE <condiciones>];
```

En cambio, si lo que quisiéramos conseguir fuera **borrar todas las filas de una tabla**, entonces sólo tendríamos que poner la sentencia **`DELETE FROM`**, sin `WHERE`.

Borrar todas las filas de una tabla en BDUOC

Podemos dejar la tabla proyectos sin ninguna fila:

```
DELETE FROM proyectos;
```

En nuestra BD, borrar los proyectos del cliente con `codigo_cliente = 2` se haría de la manera que mostramos a continuación:

```
DELETE FROM proyectos WHERE codigo_cliente = 2;
```

Borrado de filas múltiples

Fijémonos que el cliente con `codigo_cliente = 2` podría tener más de un proyecto contratado y, por lo tanto, se borraría más de una fila con una sola sentencia.

2.3. Modificación de filas de una tabla

Si quisiéramos **modificar los valores de algunas filas de una tabla** tendríamos que utilizar la sentencia **`UPDATE SET WHERE`**. A continuación, presentamos el formato:

```
UPDATE <nombre_tabla>
SET <nombre_columna> = {<expresión>|DEFAULT|NULL}
    [, <nombre_columna> = {<expresión>|DEFAULT|NULL} ...]
WHERE <condiciones>;
```

Modificación de los valores de algunas filas en BDUOC

Supongamos que los empleados del proyecto con `num_proy = 2` pasan a ganar un sueldo más alto. La modificación de esta situación sería:

```
UPDATE empleados
SET sueldo = sueldo + 1000.0
WHERE num_proy = 2;
```

Modificación de múltiples filas

Fijémonos que el proyecto con `num_proy = 2` podría tener más de un empleado asignado y, por lo tanto, se modificaría la columna sueldo de más de una fila con una sola sentencia.

2.4. Introducción de filas en la BD relacional BDUOC

Antes de empezar a hacer consultas a la BD BDUOC, habremos introducido unas cuantas filas en sus tablas con la sentencia `INSERT INTO`. De esta mane-

ra, podremos ver reflejado el resultado de las consultas que iremos haciendo, a partir de este momento. A continuación, presentamos la extensión de las diferentes tablas:

- Tabla `departamentos`:

departamentos		
nombre_dpt	**ciudad_dpt**	**telefono**
DIR	Barcelona	93.422.60.70
DIR	Gerona	972.23.89.70
DIS	Lerida	973.23.50.40
DIS	Barcelona	93.224.85.23
PROG	Tarragona	977.33.38.52
PROG	Gerona	972.23.50.91

- Tabla `clientes`:

clientes					
codigo_cli	**nombre_cli**	**nif**	**direccion**	**ciudad**	**telefono**
10	ECIGSA	38.567.893-C	Aragon 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Gerona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lerida	973.23.45.67
40	JGM	38.782.345-B	Rossellon 44	Tarragona	977.33.71.43

- Tabla `empleados`:

empleados						
codigo_empl	**nombre_empl**	**apellido_empl**	**sueldo**	**nombre_dpt**	**ciudad_dpt**	**num_proy**
1	Maria	Puig	100000.0	DIR	Gerona	1
2	Pedro	Mas	90000.0	DIR	Barcelona	4
3	Ana	Ros	70000.0	DIS	Lerida	3
4	Jorge	Roca	70000.0	DIS	Barcelona	4
5	Clara	Blanc	40000.0	PROG	Tarragona	1
6	Laura	Tort	30000.0	PROG	Tarragona	3
7	Roger	Salt	40000.0	NULL	NULL	4
8	Sergio	Grau	30000.0	PROG	Tarragona	NULL

- Tabla `proyectos`:

proyectos						
codigo_proy	**nombre_proy**	**precio**	**fecha_inicio**	**fecha_prev_fin**	**fecha_fin**	**codigo_cliente**
1	GESCOM	1000000.0	1-1-98	1-1-99	NULL	10
2	PESCI	2000000.0	1-10-96	31-3-98	1-5-98	10
3	SALSA	1000000.0	10-2-98	1-2-99	NULL	20
4	TINELL	4000000.0	1-1-97	1-12-99	NULL	30

2.5. Consultas a una BD relacional

Para hacer **consultas** sobre una tabla con SQL, hay que utilizar la **sentencia** **`SELECT FROM`**, que tiene el siguiente formato:

```
SELECT <nombre_columna_seleccionar> [[AS] <col_renombrada>]
[,<nombre_columna_seleccionar> [[AS] <col_renombrada>]...]
FROM <tabla_consultar> [[AS] <tabla_renombrada>];
```

La opción **`AS`** nos permite renombrar las columnas que queremos seleccionar o las tablas que queremos consultar, en este caso sólo una. Dicho de otra manera, nos permite la definición de alias. Fijémonos que la palabra clave `AS` es opcional, y es bastante habitual, al definir alias en la cláusula `FROM`, poner sólo un espacio en blanco en lugar de toda la palabra.

Consultas en BDUOC

A continuación, presentamos un ejemplo de consulta con la BD BDUOC para conocer todos los datos que hay en la tabla `clientes`

```
SELECT * FROM clientes;
```

El asterisco (*) después de `SELECT` indica que queremos ver todos los atributos que hay en la tabla.

La respuesta a esta consulta sería:

codigo_cli	nombre_cli	nif	direccion	ciudad	telefono
10	ECIGSA	38.567.893-C	Aragon 11	Barcelona	NULL
20	CME	38.123.898-E	Valencia 22	Gerona	972.23.57.21
30	ACME	36.432.127-A	Mallorca 33	Lerida	973.23.45.67
40	JGM	38.782.345-B	Rossellon 44	Tarragona	977.33.71.43

Si hubiéramos querido ver sólo el código, el nombre, la dirección y la ciudad, habríamos escrito lo siguiente:

```
SELECT codigo_cli, nombre_cli, direccion, ciudad
FROM clientes;
```

El resultado de la consulta anterior sería el siguiente:

codigo_cli	nombre_cli	direccion	ciudad
10	ECIGSA	Aragon 11	Barcelona
20	CME	Valencia 22	Gerona
30	ACME	Mallorca 33	Lerida
40	JGM	Rossellon 44	Tarragona

Con la sentencia `SELECT FROM` podemos seleccionar columnas de una tabla, pero para seleccionar filas de una tabla hay que añadir la cláusula `WHERE`. El formato es el siguiente:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
WHERE <condiciones>;
```

La cláusula `WHERE` nos permite obtener las filas que cumplen la condición especificada en la consulta.

Consulta en BDUOC seleccionando filas

Veamos un ejemplo en el que pedimos "los códigos de los empleados que trabajan en el proyecto número 4":

```
SELECT codigo_empl
FROM empleados
WHERE num_proy = 4;
```

La respuesta a esta consulta sería la siguiente:

codigo_empl
2
4
7

Para definir las condiciones de la cláusula `WHERE`, podemos utilizar alguno de los operadores de que dispone SQL, que son los siguientes:

Operadores de comparación	
=	Igual
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual
<>	Diferente

Operadores lógicos	
NOT	Para la negación de condiciones
AND	Para la conjunción de condiciones
OR	Para la disyunción de condiciones

Si queremos que en una consulta nos aparezcan las filas resultantes sin repeticiones, hay que poner la palabra clave **`DISTINCT`** inmediatamente después

de SELECT. También podríamos explicitar que lo queremos todo, incluso con repeticiones, poniendo ALL (opción por defecto) en lugar de DISTINCT. El formato de DISTINCT es el siguiente:

```
SELECT DISTINCT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
[WHERE <condiciones>];
```

Consulta en BDUOC seleccionando filas sin repeticiones

Si quisiéramos ver qué sueldos se pagan en nuestra empresa, podríamos hacer:

```
SELECT DISTINCT sueldo
FROM empleados;
```

La respuesta a esta consulta, sin repeticiones, sería la siguiente:

sueldo
30000.0
40000.0
70000.0
90000.0
100000.0

2.5.1. Funciones de agregación

SQL ofrece las siguientes **funciones de agregación** para efectuar diferentes operaciones con los datos de una BD:

Funciones de agregación	
Función	**Descripción**
COUNT	Nos da el número total de filas seleccionadas
SUM	Suma los valores de una columna
MIN	Nos da el valor mínimo de una columna
MAX	Nos da el valor máximo de una columna
AVG	Calcula la media de una columna

En general, las funciones de agregación se aplican a una columna, excepto la función de agregación COUNT, que normalmente se aplica a todas las columnas de la tabla o tablas seleccionadas: COUNT(*) contaría las filas de la tabla o tablas que cumplan las condiciones, COUNT(DISTINCT <nombre_columna>) sólo contaría los valores que no fueran nulos ni repetidos, y COUNT(<nombre_columna>) sólo contaría los valores que no fueran nulos.

Ejemplo de utilización de la función `COUNT(*)`

Veamos un ejemplo de uso de la función `COUNT`, que aparece en la cláusula `SELECT`, para hacer la consulta, "¿Cuántos departamentos están ubicados en la ciudad de Lérida?":

```
SELECT COUNT(*) AS numero_dpt FROM departamentos
WHERE ciudad_dpt = "Lerida";
```

La respuesta a esta consulta es la siguiente:

numero_dept
1

Veremos ejemplos de las otras funciones de agregación en los siguientes apartados.

2.5.2. Subconsultas

Una subconsulta es una consulta incluida dentro de una cláusula `WHERE` o `HAVING` de otra consulta. A veces, para expresar ciertas condiciones no hay otro remedio que obtener el valor que buscamos como resultado de una consulta.

Veremos la cláusula `HAVING` en el subapartado 2.5.5 de este módulo didáctico.

Subconsulta en BDUOC

Si quisiéramos saber los códigos y los nombres de los proyectos de precio más alto, en primer lugar tendríamos que encontrar los proyectos que tienen el precio más alto. Lo haríamos de la siguiente manera:

```
SELECT codigo_proy, nombre_proy FROM proyectos
WHERE precio = (SELECT MAX(precio)
                FROM proyectos);
```

El resultado de la consulta anterior sería el siguiente:

codigo_proy	nombre_proy
4	TINELL

Los proyectos de precio más bajo

Si en lugar de los códigos y los nombres de los proyectos de precio más alto hubiéramos querido saber los de precio más bajo, habríamos aplicado la función de agregación `MIN`.

2.5.3. Otros predicados

1. Predicado `BETWEEN`

Para expresar una condición que quiere encontrar un valor entre unos límites concretos podemos utilizar `BETWEEN`:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
WHERE <columna> BETWEEN <límite1> AND <límite2>;
```

Ejemplo de uso del predicado `BETWEEN`

Se piden "Los códigos de los empleados que ganan entre 20.000 y 50.000 euros anuales".

```
SELECT codigo_empl
FROM empleados
WHERE sueldo BETWEEN 20000.0 and 50000.0;
```

La respuesta a esta consulta sería la siguiente:

codigo_empl
5
6
7
8

2. Predicado `IN`

Para ver si un valor coincide con los elementos de una lista utilizaremos `IN`, y para ver si no coincide, `NOT IN`:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
WHERE <columna> [NOT] IN (<valor1>, ..., <valorN>);
```

Ejemplo de uso del predicado `IN`

"Queremos saber el nombre de todos los departamentos que se encuentran en las ciudades de Lérida o Tarragona."

```
SELECT nombre_dpt, ciudad_dpt
FROM departamentos
WHERE ciudad_dpt IN ("Lerida", "Tarragona");
```

La respuesta sería la siguiente:

nombre_dpt	ciudad_dpt
DIS	Lerida
PROG	Tarragona

3. Predicado `LIKE`

Para ver si una columna de tipo carácter cumple alguna característica determinada podemos utilizar `LIKE`:

```
SELECT <nombre_columnas_seleccionar> FROM <tabla_consultar>
WHERE <columna> LIKE <característica>;
```

Los **patrones de SQL para expresar características** son los siguientes:

Otros patrones

Aunque _ y % son los caracteres escogidos por el estándar, cada SGBD comercial ofrece distintas variantes.

a) Se pone un carácter _ para cada carácter individual que se quiera considerar.

b) Se pone un carácter % para expresar una secuencia de caracteres, que puede ser ninguna.

Ejemplo de uso del predicado `LIKE`

Atributos añadidos

Aunque en la consulta se pida sólo el nombre de los empleados, añadimos el código para poder diferenciar dos empleados con el mismo nombre.

Se buscan los nombres de los empleados que empiezan con la letra J y los proyectos que empiezan por S y tienen cinco letras:

a) Nombres de empleados que empiezan con la letra J:

```
SELECT codigo_empl, nombre_empl
FROM empleados
WHERE nombre_empl LIKE "J%";
```

La respuesta a esta consulta sería la siguiente:

codigo_empl	nombre_empl
4	Jorge

b) Proyectos que empiezan por S y tienen cinco letras:

```
SELECT codigo_proy
FROM proyectos
WHERE nombre_proy LIKE "S_ _ _ _";
```

Y la respuesta a esta otra consulta sería la siguiente:

codigo_proy
3

4. Predicado `IS NULL`

Para ver si un valor es nulo utilizaremos `IS NULL`, y para ver si no lo es, `IS NOT NULL`. El formato es:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
WHERE <columna> IS [NOT] NULL;
```

Ejemplo de uso del predicado `IS NULL`

"Queremos saber el código y el nombre de todos los empleados que no están asignados a ningún proyecto."

```
SELECT codigo_empl, nombre_empl FROM empleados
WHERE num_proy IS NULL;
```

Obtendríamos la siguiente respuesta:

codigo_empl	nombre_empl
8	Sergio

5. Predicados ANY/SOME i ALL

Para ver si una columna cumple que todas sus filas (ALL) o alguna de sus filas (ANY/SOME) satisfagan una condición, podemos hacer:

Los predicados ANY/SOME

Podemos escoger cualquiera de los dos predicados para pedir alguna fila que satisfaga una condición.

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
WHERE <nombre_columna> <operador_comparación> {ALL|ANY|SOME}
      <subconsulta>;
```

Ejemplo de uso de los predicados ALL y ANY/SOME

a) Un ejemplo de utilización de ALL para encontrar los códigos y los nombres de los proyectos en que los sueldos de todos los empleados asignados son más pequeños que el precio del proyecto.

```
SELECT codigo_proy, nombre_proy FROM proyectos
WHERE precio > ALL(SELECT sueldo
                   FROM empleados
                   WHERE codigo_proy = num_proy);
```

Fijémonos en la condición de WHERE de la subconsulta, que nos asegura que los sueldos que miramos son los de los empleados asignados al proyecto de la consulta. La respuesta a esta consulta sería la siguiente:

codigo_proy	nombre_proy
1	GESCOM
2	PESCI
3	SALSA
4	TINELL

b) Un ejemplo de utilización de ANY/SOME para encontrar los códigos y los nombres de los proyectos que tienen algún empleado que gana un sueldo más elevado que el precio del proyecto en el cual trabaja.

```
SELECT codigo_proy, nombre_proy FROM proyectos
WHERE precio < ANY(SELECT sueldo
                   FROM empleados
                   WHERE codigo_proy = num_proy);
```

La respuesta a esta consulta está vacía, como se ve a continuación:

codigo_proy	nombre_proy

6. Predicado EXISTS

Para ver si una subconsulta produce alguna fila de resultados, podemos utilizar la sentencia llamada *test de existencia*: EXISTS. Para comprobar si una subconsulta no produce ninguna fila de resultados, podemos utilizar NOT EXISTS.

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
WHERE [NOT] EXISTS <subconsulta>;
```

Ejemplo de uso del predicado EXISTS

Se buscan los códigos y los nombres de los empleados que están asignados a algún proyecto.

```
SELECT codigo_empl, nombre_empl FROM empleados
WHERE EXISTS (SELECT *
              FROM proyectos
              WHERE codigo_proy = num_proy);
```

La respuesta a esta consulta sería la siguiente:

codigo_empl	nombre_empl
1	Maria
2	Pedro
3	Ana
4	Jorge
5	Clara
6	Laura
7	Roger

2.5.4. Ordenación de los datos obtenidos en respuestas a consultas

Si se quiere que, al hacer una consulta, los datos aparezcan en un orden determinado, hay que utilizar la **cláusula ORDER BY** en la sentencia SELECT, que tiene el siguiente formato:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
[WHERE <condiciones>]
ORDER BY <nombre_columna_según_la_que_ordenar> [DESC]
        [, <nombre_columna_según_la_que_ordenar> [DESC]...];
```

Consulta con respuesta ordenada en BDUOC

Se quiere consultar los nombres de los empleados ordenados según el sueldo que ganan, y si ganan el mismo sueldo, ordenados alfabéticamente por el nombre:

```
SELECT codigo_empl, nombre_empl, apellido_empl, sueldo
FROM empleados
ORDER BY sueldo, nombre_empl;
```

Esta consulta daría la siguiente respuesta:

codigo_empl	nombre_empl	apellido_empl	sueldo
6	Laura	Tort	30000.0
8	Sergio	Grau	30000.0
5	Clara	Blanc	40000.0
7	Roger	Salt	40000.0
3	Ana	Ros	70000.0
4	Jorge	Roca	70000.0
2	Pedro	Mas	90000.0
1	Maria	Puig	10000.0

Si no se especifica nada más, el orden que se seguirá será ascendente, pero si se quiere seguir un orden descendente hay que añadir **DESC** detrás de cada factor de ordenación expresado en la cláusula ORDER BY:

```
ORDER BY <nombre_columna> [DESC][, <nombre_columna> [DESC]...];
```

También se puede explicitar un orden ascendente poniendo la palabra clave **ASC** (opción por defecto).

2.5.5. Consultas con agrupación de filas de una tabla

Las siguientes cláusulas, añadidas a la sentencia SELECT FROM, permiten organizar las filas por grupos:

a) La cláusula **GROUP BY** nos sirve para agrupar filas según las columnas que indique esta cláusula.

b) La cláusula **HAVING** especifica condiciones de búsqueda para grupos de filas; lleva a cabo la misma función que antes hacía la cláusula WHERE para las filas de toda la tabla, pero ahora las condiciones se aplican a los grupos obtenidos.

Presenta el siguiente formato:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla_consultar>
[WHERE <condiciones>]
```

```
GROUP BY <columnas_según_las_que_agrupar>
[HAVING <condiciones_por_grupos>]
[ORDER BY <nombre_columna> [DESC][, <nombre_columna> [DESC]...]];
```

Fijémonos que en las sentencias SQL se van añadiendo cláusulas a medida que la dificultad o la exigencia de la consulta lo requiere.

Consulta con agrupación de filas en BDUOC

Se quiere saber el sueldo medio que ganan los empleados de cada departamento.

```
SELECT nombre_dpt, ciudad_dpt, AVG(sueldo) AS sueldo_medio
FROM empleados
GROUP BY nombre_dpt, ciudad_dpt;
```

Factores de agrupación

Los factores de agrupación de la cláusula `GROUP BY` tienen que ser, como mínimo, las columnas que figuran en el `SELECT`, exceptuando las columnas afectadas por las funciones de agregación.

La cláusula `GROUP BY` podríamos decir que empaqueta a los empleados de la tabla `empleados` según el departamento en que están asignados. En la siguiente figura, podemos ver los grupos que se formarían después de agrupar:

empleados

codigo_empl	nombre_empl	apellido_empl	sueldo	nombre_dpt	ciudad_dpt	num_proy
1	Maria	Puig	100000.0	DIR	Gerona	1
2	Pedro	Mas	90000.0	DIR	Barcelona	4
3	Ana	Ros	70000.0	DIS	Lerida	3
4	Jorge	Roca	70000.0	DIS	Barcelona	4
5	Clara	Blanc	40000.0	PROG	Tarragona	1
6	Laura	Tort	30000.0	PROG	Tarragona	3
8	Sergio	Grau	30000.0	PROG	Tarragona	NULL
7	Roger	Salt	40000.0	NULL	NULL	4

Y así, con la agrupación anterior, es sencillo obtener el resultado de esta consulta:

nombre_dpt	ciudad_dpt	sueldo_medio
DIR	Gerona	100000.0
DIR	Barcelona	90000.0
DIS	Lerida	70000.0
DIS	Barcelona	70000.0
PROG	Tarragona	33000.0
NULL	NULL	40000.0

Ejemplo de uso de la función de agregación `SUM`

Veamos un ejemplo de uso de una función de agregación `SUM` de SQL que aparece en la cláusula `HAVING` de `GROUP BY`: "Queremos saber los códigos de los proyectos en los cuales la suma de los sueldos de los empleados es mayor de 180.000 euros":

```
SELECT num_proy
FROM empleados
GROUP BY num_proy
HAVING SUM(sueldo)>180000.0;
```

DISTINCT y GROUP BY

En este ejemplo, no hay que poner `DISTINCT`, aunque la columna `num_proy` no es atributo identificador. Fijémonos que en la tabla `empleados` hemos puesto todos los proyectos que tienen el mismo número juntos en un mismo grupo y no puede ser que salgan repetidos.

Nuevamente, antes de mostrar el resultado de esta consulta, veamos gráficamente qué grupos se formarían en este caso:

empleados						
codigo_empl	nombre_empl	apellido_empl	sueldo	nombre_dpt	ciudad_dpt	num_proy
1	Maria	Puig	100000.0	DIR	Gerona	1
5	Clara	Blanc	40000.0	PROG	Tarragona	1
3	Ana	Ros	70000.0	DIS	Lerida	3
6	Laura	Tort	30000.0	PROG	Tarragona	3
2	Pedro	Mas	90000.0	DIR	Barcelona	4
4	Jorge	Roca	70000.0	DIS	Barcelona	4
7	Roger	Salt	40000.0	NULL	NULL	4
8	Sergio	Grau	30000.0	PROG	Tarragona	NULL

El resultado de la consulta sería el siguiente:

num_proy
4

2.5.6. Consultas en más de una tabla

Muchas veces queremos consultar datos de más de una tabla haciendo combinaciones de columnas de tablas diferentes. En SQL es posible listar más de una tabla especificándolo en la cláusula `FROM`.

1. Combinación

La combinación consigue crear una sola tabla a partir de las tablas especificadas en la cláusula `FROM`, haciendo coincidir los valores de las columnas relacionadas de estas tablas.

Ejemplo de combinación en BDUOC

Se quiere saber el NIF del cliente y el código y el precio del proyecto que se desarrolla para el cliente número 20:

```
SELECT proyectos.codigo_proy, proyectos.precio, clientes.nif
FROM clientes, proyectos
WHERE clientes.codigo_cli = proyectos.codigo_cliente AND
      clientes.codigo_cli = 20;
```

El resultado sería el siguiente:

proyectos.codigo_proy	proyectos.precio	clientes.nif
3	1000000.0	38.123.898-E

Si trabajamos con más de una tabla, puede pasar que la tabla resultante tenga dos columnas con el mismo nombre. Por eso, es obligatorio especificar a qué tabla corresponden las columnas a las que nos referimos, nombrando a la tabla a la que

pertenecen, antes de ponerlas (por ejemplo, `clientes.codigo_cli`). Para simplificarlo se utilizan los alias, que, en este caso, se definen en la cláusula `FROM`.

Ejemplo de alias en BDUOC

c podría ser el alias de la tabla clientes. De esta manera, para indicar a qué tabla pertenece `codigo_cli` sólo habría que poner `c.codigo_cli`.

Veamos cómo quedaría la consulta anterior expresada mediante alias, aunque en este ejemplo no serían necesarios porque las columnas de las dos tablas tienen nombres diferentes. Pediremos, además, las columnas `c.codigo_cli` y `p.codigo_cliente`.

```
SELECT p.codigo_proy, p.precio, c.nif, p.codigo_cliente, c.codigo_cli
FROM clientes c, proyectos p
WHERE c.codigo_cli = p.codigo_cliente AND c.codigo_cli = 20;
```

El resultado sería el siguiente:

p.codigo_proy	p.precio	c.nif	p.codigo_cliente	c.codigo_cli
3	1000000.0	38.123.898-E	20	20

Advertimos que en el `WHERE` necesitamos expresar el vínculo que hay entre las dos tablas, en este caso `codigo_cli` de clientes y `codigo_cliente` de proyectos. Expresado en operaciones del álgebra relacional, esto quiere decir que hacemos una combinación en lugar de un producto cartesiano.

Fijémonos en que, igual que en el álgebra relacional, la operación que acabamos de hacer es una equicombinación (*equi-join*) y, por lo tanto, nos aparecen dos columnas idénticas: `c.codigo_cli` y `p.codigo_cliente`.

La manera de expresar la combinación que acabamos de ver pertenece al SQL:1989. Una manera alternativa de hacer la equicombinación de antes, utilizando SQL:1992, sería la siguiente:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla1> JOIN <tabla2>
       {ON <condiciones>|
       USING (<nombre_columna> [,<nombre_columna>...])}
[WHERE <condiciones>];
```

Ejemplo anterior con SQL:1992

El ejemplo que hemos hecho antes utilizando SQL:1992 sería:

```
SELECT p.codigo_proy, p.precio, c.nif, p.codigo_cliente, c.codigo_cli
FROM clientes c JOIN proyectos p ON c.codigo_cli = p.codigo_cliente
WHERE c.codigo_cli = 20;
```

Con esta operación, se obtendría el mismo resultado que con el método anterior.

La opción `ON`, además de expresar condiciones con la igualdad, en caso de que las columnas que queremos relacionar tengan nombres diferentes, nos ofrece la posibilidad de expresar condiciones con los otros operadores de compara-

ción que no sean el de igualdad. Sería lo equivalente a la operación que, en álgebra relacional, hemos llamado *θ-combinación* (*θ-join*).

También podemos utilizar una misma tabla dos veces con alias diferentes, para poder distinguirlas.

Dos alias para una misma tabla en BDUOC

Se piden los códigos y los apellidos de los empleados que ganan más que el empleado que tiene por código el número 5.

```
SELECT e1.codigo_empl, e1.apellido_empl
FROM empleados e1 JOIN empleados e2 ON e1.sueldo > e2.sueldo
WHERE e2.codigo_empl = 5;
```

Se ha tomado la tabla `e2` para fijar la fila del empleado con código número 5, de manera que se pueda comparar el sueldo de la tabla `e1`, que contiene a todos los empleados, con el sueldo de la tabla `e2`, que contiene sólo al empleado 5.

La respuesta a esta consulta sería la siguiente:

e1.codigo_empl	e1.apellido_empl
1	Puig
2	Mas
3	Ros
4	Roca

2. Combinación natural

La combinación natural (*natural join*) de dos tablas consiste, básicamente, igual que en el álgebra relacional, en hacer una equicombinación entre columnas del mismo nombre y eliminar las columnas repetidas. La combinación natural, utilizando SQL:1992, se haría de la siguiente manera:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla1> NATURAL JOIN <tabla2>
[WHERE <condiciones>];
```

Combinación natural en BDUOC

Se quiere saber el código y el nombre de los empleados que están asignados al departamento que tiene por teléfono 977.333.852.

```
SELECT codigo_empl, nombre_empl
FROM empleados NATURAL JOIN departamentos
WHERE telefono = "977.333.852";
```

La combinación natural también se podría hacer con la cláusula `USING`, sólo aplicando la palabra reservada `JOIN`:

```
SELECT codigo_empl, nombre_empl
FROM empleados JOIN departamentos USING (nombre_dpt,ciudad_dpt)
WHERE telefono = "977.333.852";
```

La respuesta a esta consulta sería la siguiente:

codigo_empl	nombre_empl
5	Clara
6	Laura
8	Sergio

3. Combinación interna y externa

Cualquier combinación puede ser interna o externa.

La **combinación interna** (*inner join*) sólo se queda con las filas que tienen valores idénticos en las columnas de las tablas que compara. Esto puede hacer que se pierda alguna fila interesante de alguna de las dos tablas, por ejemplo, porque se encuentra `NULL` en el momento de hacer la combinación. Su formato es el siguiente:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla1> [NATURAL] [INNER] JOIN <tabla2>
       {ON <condiciones>|
       USING (<nombre_columna> [,<nombre_columna>...])}
[WHERE <condiciones>];
```

Si no se quiere perder ninguna fila de una tabla determinada, se puede hacer una **combinación externa** (*outer join*), que permite obtener todos los valores de la tabla que hemos puesto a la derecha, o los valores de la que hemos puesto a la izquierda, o todos los valores de ambas tablas. Su formato es el siguiente:

```
SELECT <nombre_columnas_seleccionar>
FROM <tabla1> [NATURAL] {LEFT|RIGHT|FULL} [OUTER] JOIN <tabla2>
     {ON <condiciones>|
     USING (<nombre_columna> [,<nombre_columna>...])}
[WHERE <condiciones>];
```

Combinación interna

Aunque en el ejemplo hacemos una combinación natural interna, no hay que poner la palabra `INNER`, ya que es la opción por defecto.

Combinación natural interna en BDUOC

Si se quisiera relacionar con una combinación natural interna las tablas `empleados` y `departamentos` para saber el código y el nombre de todos los empleados y el nombre, la ciudad y el teléfono de todos los departamentos, se tendría que hacer lo siguiente:

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dpt, e.ciudad_dpt, d.telefono
FROM empleados e NATURAL JOIN departamentos d;
```

Con esta combinación se obtendría el resultado siguiente:

e.codigo_empl	e.nombre_empl	e.nombre_dpt	e.ciudad_dpt	d.telefono
1	Maria	DIR	Gerona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lerida	973.23.50.40
4	Jorge	DIS	Barcelona	93.224.85.23
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
8	Sergio	PROG	Tarragona	977.33.38.52

En el resultado, no sale el empleado número 7, que no está asignado a ningún departamento, ni el departamento de programación de Gerona, que no tiene ningún empleado asignado.

Combinación natural externa en BDUOC

En los siguuientes ejemplos, veremos cómo varían los resultados que iremos obteniendo según los tipos de combinación externa:

a) Combinación externa izquierda

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dpt, e.ciudad_dpt,
       d.telefono
FROM empleados e NATURAL LEFT OUTER JOIN departamentos d;
```

Combinación externa izquierda

Figura el empleado 7.

Con esta combinación, se obtendría el resultado siguiente:

e.codigo_empl	e.nombre_empl	e.nombre_dpt	e.ciudad_dpt	d.telefono
1	Maria	DIR	Gerona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lerida	973.23.50.40
4	Jorge	DIS	Barcelona	93.224.85.23
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
7	Roger	NULL	NULL	NULL
8	Sergio	PROG	Tarragona	977.33.38.52

b) Combinació externa derecha

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dpt, e.ciudad_dpt,
       d.telefono
FROM empleados e NATURAL RIGHT OUTER JOIN departamentos d;
```

Combinación externa derecha

Figura el departamento de programación de Gerona.

Con esta combinación, se obtendría el resultado siguiente:

e.codigo_empl	e.nombre_empl	e.nombre_dpt	e.ciudad_dpt	d.telefono
1	Maria	DIR	Gerona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lerida	973.23.50.40
4	Jorge	DIS	Barcelona	93.224.85.23

e.codigo_empl	e.nombre_empl	e.nombre_dpt	e.ciudad_dpt	d.telefono
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
8	Sergio	PROG	Tarragona	977.33.38.52
NULL	NULL	PROG	Gerona	972.23.50.91

c) Combinación externa plena

```
SELECT e.codigo_empl, e.nombre_empl, e.nombre_dpt, e.ciudad_dpt,
       d.telefono
FROM empleados e NATURAL FULL OUTER JOIN departamentos d;
```

Combinación externa plena

Figura el empleado 7 y el departamento de programación de Gerona.

Con esta combinación, se obtendría el resultado siguiente:

e.codigo_empl	e.nombre_empl	e.nombre_dpt	e.ciudad_dpt	d.telefono
1	Maria	DIR	Gerona	972.23.89.70
2	Pedro	DIR	Barcelona	93.422.60.70
3	Ana	DIS	Lerida	973.23.50.40
4	Jorge	DIS	Barcelona	93.224.85.23
5	Clara	PROG	Tarragona	977.33.38.52
6	Laura	PROG	Tarragona	977.33.38.52
7	Roger	NULL	NULL	NULL
8	Sergio	PROG	Tarragona	977.33.38.52
NULL	NULL	PROG	Gerona	972.23.50.91

4. Combinaciones con más de dos tablas

Si se quieren combinar tres tablas o más con SQL:1989 sólo hay que añadir todas las tablas en el `FROM` y los vínculos necesarios en el `WHERE`. Si se vuelven a combinar con SQL:1992 hay que ir haciendo combinaciones por parejas de tablas y la tabla resultante se convertirá en la primera pareja de la siguiente.

Combinaciones con más de dos tablas en BDUOC

Veamos ejemplos de los dos casos, suponiendo que se quieran combinar las tablas `empleados`, `proyectos` y `clientes`:

```
SELECT *
FROM empleados, proyectos, clientes
WHERE num_proy = codigo_proy AND codigo_cliente = codigo_cli;
```

o bien:

```
SELECT *
FROM (empleados JOIN proyectos ON num_proy = codigo_proy)
JOIN clientes ON codigo_cliente = codigo_cli;
```

2.5.7. La unión

La cláusula **UNION** permite unir consultas de dos o más sentencias SELECT FROM. Su formato es el siguiente:

```
SELECT <nombre_columnas>
FROM <tabla>
[WHERE <condiciones>]
UNION [ALL]
SELECT <nombre_columnas>
FROM <tabla>
[WHERE <condiciones>];
```

Si se utiliza la opción ALL, aparecen todas las filas obtenidas al hacer la unión. No se escribirá esta opción si se quieren eliminar las filas repetidas. Lo más importante de la unión es que somos nosotros los que tenemos que vigilar que se haga entre columnas definidas sobre dominios compatibles; es decir, que tengan la misma interpretación semántica. Como ya se ha dicho, SQL no ofrece herramientas para asegurar la compatibilidad semántica entre columnas.

Utilización de la unión en BDUOC

Si se quieren saber todas las ciudades que hay en la BD se puede hacer lo siguiente:

```
SELECT ciudad
FROM clientes
UNION
SELECT ciudad_dpt
FROM departamentos;
```

El resultado de esta consulta sería el siguiente:

ciudad
Barcelona
Gerona
Lerida
Tarragona

2.5.8. La intersección

Para hacer la intersección entre dos o más sentencias SELECT FROM, se puede utilizar la cláusula **INTERSECT**, cuyo formato es el siguiente:

```
SELECT <nombre_columnas>
FROM <tabla>
[WHERE <condiciones>]
INTERSECT [ALL]
SELECT <nombre_columnas>
FROM <tabla>
[WHERE <condiciones>];
```

Si se utiliza la opción ALL aparecen todas las filas obtenidas al hacer la intersección. No se escribirá esta opción si se quieren eliminar las filas repetidas. Lo más importante de la intersección es que somos nosotros los que tenemos que vigilar que se haga entre columnas definidas sobre dominios compatibles; es decir, que tengan la misma interpretación semántica.

Utilización de la intersección en BDUOC

Se quieren saber todas las ciudades donde hay departamentos en los cuales se puede encontrar algún cliente.

```
SELECT ciudad
FROM clientes
INTERSECT
SELECT ciudad_dpt
FROM departamentos;
```

El resultado de esta consulta sería el siguiente:

ciudad
Barcelona
Gerona
Lerida
Tarragona

La intersección es una de las operaciones de SQL que se puede hacer de mayor número de maneras diferentes:

- Intersección utilizando IN:

```
SELECT <nombre_columnas>
FROM <tabla>
WHERE <nombre_columna> IN (SELECT <nombre_columna>
                       FROM <tabla>
                       [WHERE <condiciones>]);
```

- Intersección utilizando EXISTS:

```
SELECT <nombre_columnas>
FROM <tabla>
WHERE EXISTS (SELECT *
              FROM <tabla>
              WHERE <condiciones>);
```

Ejemplo anterior expresado con IN y con EXISTS

El ejemplo anterior se podría expresar con IN:

```
SELECT ciudad
FROM clientes
WHERE ciudad IN (SELECT ciudad_dpt
                 FROM departamentos);
```

o también con EXISTS:

```
SELECT c.ciudad
FROM clientes c
WHERE EXISTS (SELECT *
              FROM departamentos d
              WHERE c.ciudad = d.ciudad_dpt);
```

2.5.9. La diferencia

Para encontrar la diferencia entre dos o más sentencias SELECT FROM podemos utilizar la cláusula **EXCEPT**, que tiene este formato:

```
SELECT <nombre_columnas>
FROM <tabla>
[WHERE <condiciones>]
EXCEPT [ALL]
SELECT <nombre_columnas>
FROM <tabla>
[WHERE <condiciones>];
```

Si se utiliza la opción ALL, aparecen todas las filas obtenidas al hacer la **diferencia**. No se escribirá esta opción si se quieren eliminar las filas repetidas. Lo más importante de la diferencia es que somos nosotros los que tenemos que vigilar que se haga entre columnas definidas sobre dominios compatibles.

Utilización de la diferencia en BDUOC

Se quieren saber los clientes que no han contratado ningún proyecto.

```
SELECT codigo_cli FROM clientes EXCEPT
SELECT codigo_cliente FROM proyectos;
```

El resultado de esta consulta sería el siguiente:

codigo_cli
40

La diferencia es, junto con la intersección, una de las operaciones de SQL que se puede hacer de más maneras diferentes:

- Diferencia utilizando NOT IN:

```
SELECT <nombre_columnas>
FROM <tabla>
WHERE <nombre_columna> NOT IN (SELECT <nombre_columna>
                               FROM <tabla>
                               [WHERE <condiciones>]);
```

- Diferencia utilizando `NOT EXISTS`:

```
SELECT <nombre_columnas>
FROM <tabla>
WHERE NOT EXISTS (SELECT *
                  FROM <tabla>
                  WHERE <condiciones>);
```

Ejemplo anterior expresado con `NOT IN` y con `NOT EXISTS`

El ejemplo anterior no se podría expresar con `NOT IN`:

```
SELECT codigo_cli
FROM clientes
WHERE codigo_cli NOT IN (SELECT codigo_cliente FROM proyectos);
```

o también con `NOT EXISTS`:

```
SELECT c.codigo_cli
FROM clientes c
WHERE NOT EXISTS (SELECT *
                  FROM proyectos p
                  WHERE c.codigo_cli = p.codigo_cliente);
```

3. Sentencias de control

Además de definir y manipular una BD relacional, es importante que se establezcan **mecanismos de control** para resolver problemas de concurrencia de usuarios y garantizar la seguridad de los datos. Para la concurrencia de usuarios utilizaremos el concepto de *transacción*, y para la seguridad veremos cómo se puede autorizar y desautorizar a usuarios a acceder a la BD.

3.1. Las transacciones

Una transacción es una unidad lógica de trabajo. O informalmente, y trabajando con SQL, un conjunto de sentencias que se ejecutan como si fuesen una sola. En general, las sentencias que forman parte de una transacción se interrelacionan entre sí, y no tiene sentido que se ejecute una sin que se ejecuten las demás.

La mayoría de las transacciones se inician de forma implícita al utilizar alguna sentencia que empieza con `CREATE`, `ALTER`, `DROP`, `SET`, `DECLARE`, `GRANT` o `REVOKE`, aunque existe la sentencia SQL para **iniciar transacciones**, que es la siguiente:

```
SET TRANSACTION {READ ONLY|READ WRITE};
```

Si queremos actualizar la BD utilizaremos la opción `READ WRITE`, y si no la queremos actualizar, elegiremos la opción `READ ONLY`.

Sin embargo, en cambio, una transacción siempre debe acabar explícitamente con alguna de las siguientes sentencias:

```
{COMMIT|ROLLBACK} [WORK];
```

La diferencia entre `COMMIT` y `ROLLBACK` es que mientras la sentencia `COMMIT` confirma todos los cambios producidos contra la BD durante la ejecución de la transacción, la sentencia `ROLLBACK` deshace todos los cambios que se hayan producido en la BD y la deja como estaba antes del inicio de nuestra transacción.

La palabra reservada `WORK` sólo sirve para aclarar lo que hace la sentencia, y es totalmente opcional.

Ejemplo de transacción

A continuación proponemos un ejemplo de transacción en el que se quiere disminuir el sueldo de los empleados que han trabajado en el proyecto 3 en 1.000 euros. y aumentar el sueldo de los empleados que han trabajado en el proyecto 1 también en 1.000 euros.

```
SET TRANSACTION READ WRITE;
UPDATE empleados SET sueldo = sueldo - 1000 WHERE num_proy = 3;
UPDATE empleados SET sueldo = sueldo + 1000 WHERE num_proy = 1;
COMMIT;
```

3.2. Las autorizaciones y desautorizaciones

Todos los privilegios sobre la BD los tiene su propietario, pero no es el único que accede a ésta. Por este motivo, SQL nos ofrece sentencias para autorizar y desautorizar a otros usuarios.

1) Autorizaciones

Para autorizar, SQL dispone de la siguiente sentencia:

```
GRANT <privilegios> ON <objeto> TO <usuarios>
[WITH GRANT OPTION];
```

Donde tenemos que:

a) **`privilegios`** puede ser:

- `ALL PRIVILEGES`: todos los privilegios sobre el objeto especificado.
- `USAGE`: utilización del objeto especificado; en este caso el dominio.
- `SELECT` [(`columnas`)]: consultas. Se puede concretar de qué `columnas`.
- `INSERT`: inserciones.
- `UPDATE` [(`columnas`)]: modificaciones. Se puede concretar de qué `columnas`.
- `DELETE`: borrados.
- `REFERENCES` [(`columna`)]: referencia del objeto en restricciones de integridad. Se puede concretar de qué `columnas`.

b) **`Objeto`** debe ser:

- `DOMAIN`: dominio

- `TABLE`: tabla.

- `Vista`.

c) **Usuarios** puede ser todo el mundo: `PUBLIC`, o bien una lista de los identificadores de los usuarios que queremos autorizar.

d) La opción **`WITH GRANT OPTION`** permite que el usuario que autoricemos pueda, a su vez, autorizar a otros usuarios a acceder al objeto con los mismos privilegios con los que ha sido autorizado.

2) Desautorizaciones

Para desautorizar, SQL dispone de la siguiente sentencia:

```
REVOKE [GRANT OPTION FOR] <privilegios> ON <objeto> FROM
<usuarios> [RESTRICT|CASCADE];
```

Donde tenemos que:

a) `privilegios`, `objeto` y `usuarios` son los mismos que para la sentencia `GRANT`.

b) La opción **`GRANT OPTION FOR`** se utilizaría en el caso de que quisiéramos eliminar el derecho a autorizar (`WITH GRANT OPTION`).

c) Si un usuario al que hemos autorizado ha autorizado a su vez a otros, que al mismo tiempo pueden haber hecho más autorizaciones, la **opción `CASCADE`** hace que queden desautorizados todos a la vez.

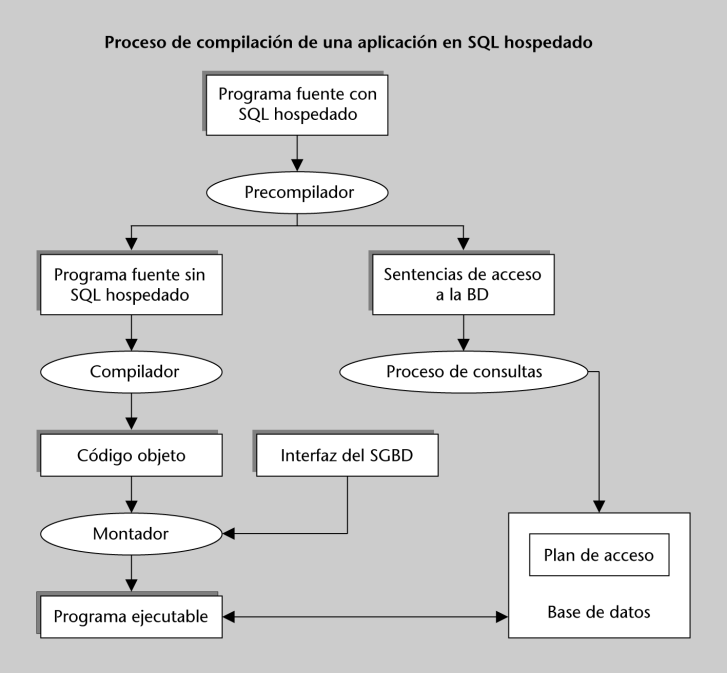
d) La opción **`RESTRICT`** no nos permite desautorizar a un usuario si éste ha autorizado a otros.

4. Sublenguajes especializados

Muchas veces querremos acceder a la BD desde una aplicación hecha en un lenguaje de programación cualquiera. Para utilizar SQL desde un lenguaje de programación, podemos utilizar **SQL hospedado**, y para trabajar con SQL hospedado necesitamos un precompilador que separe las sentencias del lenguaje de programación de las del lenguaje de BD. Una alternativa muy interesante a esta manera de trabajar son las **rutinas SQL/CLI**.

El objetivo de este apartado no es explicar en profundidad SQL hospedado, y aun menos, las rutinas SQL/CLI. Sólo introduciremos las ideas básicas del funcionamiento de ambos y presentaremos un ejemplo de combinación de SQL con el lenguaje de programación Java.

4.1. SQL hospedado



Para crear y manipular una BD relacional necesitamos SQL. Además, si el trabajo que queremos hacer requiere el poder de procesamiento de un lenguaje de programación como Java, C++, Cobol, Fortran, Pascal, etc., podemos utilizar SQL hospedado en el lenguaje de programación escogido, que llamaremos *anfitrión*. Para poder compilar esta mezcla de llamadas de SQL y sentencias del propio lenguaje de programación, antes tenemos que utilizar un precompilador. Un precompilador es una herramienta que separa las sentencias de SQL y las sentencias de programación. Allí donde en el programa fuente haya una sentencia de acceso a la BD, marcada según el estándar con la cláusula `EXEC SQL`*, se tiene que insertar una llamada a la interfaz del SGBD. El programa fuente resultante de la

*** Puede haber pequeñas diferencias dependiendo del lenguaje de programación concreto que consideremos. Por ejemplo, si utilizamos SQLJ (SQL hospedado en Java) la palabra reservada es `#sql`.**

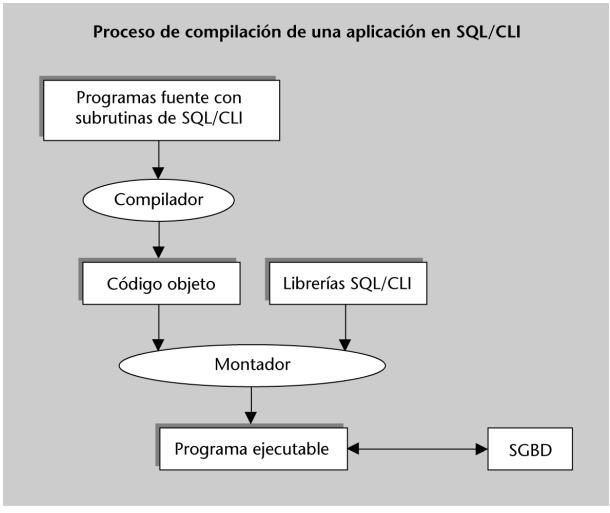
precompilación sólo incluye sentencias del lenguaje de programación, preparado para ser compilado, montado y ejecutado.

Todas las sentencias de definición y manipulación que hemos visto para SQL se pueden utilizar en SQL hospedado, teniendo presente ahora que necesitaremos un nuevo tipo de variables llamadas **variables puente** que servirán para que el programa pueda recoger y utilizar los valores de una fila de la BD. Otra cosa que cabe tener en cuenta es que, cuando el resultado de una sentencia SQL obtenga o haga referencia a más de una fila, entonces tendremos que trabajar con el **concepto de cursor**.

Un cursor se tiene que haber declarado antes de utilizarlo (`EXEC SQL DECLARE <nombre_cursor> CURSOR FOR`). Para usarlo, hay que abrir (`EXEC SQL OPEN <nombre_cursor>`), ir cogiendo los datos de uno en uno, (`EXEC SQL FETCH <nombre_cursor> INTO`), tratarlos y, finalmente, cerrarlo (`EXEC SQL CLOSE <nombre_cursor>`).

4.2. Las SQL/CLI

Las SQL/CLI permiten que aplicaciones desarrolladas en un cierto lenguaje de programación (con sólo las herramientas disponibles para este lenguaje y sin el uso de un precompilador) puedan incluir sentencias SQL mediante llamadas a rutinas predefinidas disponibles en librerías. Estas sentencias SQL se tienen que interpretar en tiempo de ejecución. A diferencia de SQL hospedado, que requería el uso de un precompilador, las SQL/CLI no lo necesitan. Pero, al generar el plan de acceso de las sentencias SQL en tiempo de ejecución, se produce una disminución global del rendimiento con respecto al SQL hospedado que encuentra el plan de acceso de las consultas en tiempo de compilación. Si se utilizan las SQL/CLI, lo que hay que hacer es, simplemente, montar la aplicación junto con la librería de rutinas tal como se muestra en la siguiente figura:



La **interfaz ODBC** (*Open Database Connectivity*) define una librería de funciones que permite a las aplicaciones acceder al SGBD utilizando SQL. Las SQL/CLI se basan fuertemente en las características de la interfaz ODBC y, gracias

al trabajo desarrollado por SAG-X/Open (SQL Access Group-X/Open), fueron añadidas al estándar SQL:1992 en 1995.

Las SQL/CLI son rutinas que llaman al SGBD para interpretar las sentencias SQL que pide la aplicación. Desde el punto de vista del SGBD, las SQL/CLI se pueden ver, simplemente, como otras aplicaciones. Esto hace que el código de las aplicaciones sólo contenga instrucciones del lenguaje de programación. Todas las sentencias SQL quedan escondidas dentro de las rutinas de las SQL/CLI. Además, las aplicaciones que se desarrollan no necesitan conocer las sentencias SQL antes de ser ejecutadas y pueden ser independientes de un SGBD concreto.

El nombre de las rutinas SQL/CLI consta de dos partes: un prefijo (`SQL`) y el nombre del trabajo que hace la subrutina, por ejemplo: `Fetch`. Así, esta subrutina se llamará `SQLFetch`.

4.3. SQL y Java

Para ilustrar las dos técnicas anteriores de SQL programado, hemos escogido el lenguaje Java. En el caso de SQL hospedado en Java, **SQLJ** sigue bastan-te fielmente lo que hemos explicado para esta técnica. En cambio, las SQL/CLI de Java, llamadas **JDBC**, son una solución basada en la filosofía de la técnica que hemos presentado, pero que se desliga de las fuertes características de la interfaz ODBC. Por este motivo, las presentaremos en primer lugar.

4.3.1. JDBC

JDBC es un API (*Application Programming Interface*) estándar de Java que permite tener interacción con los datos de la BD desde un programa escrito en Java. Se basa en la técnica SQL/CLI y consiste en una serie de clases e interfaces escritas en Java que puede utilizar un programador para acceder a la BD.

Ejemplo

Ejecución de una consulta con su correspondiente tratamiento de errores sobre la tabla `empleados` para obtener el nombre del empleado con código 1.

```
try
{
  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT nombre_empl FROM empleados
  WHERE codigo_empl=1");
  rs.next();
  String qNombre = rs.getString("nombre_empl");
  System.out.println(qNombre);
  rs.close();
  stmt.close();
}
catch (SQLException e)
{
  if (e.getSQLState().equals("02000"))
  {
     System.out.println("No existe ninguna tupla que cumpla la
                         condición");
  }
}
```

Nota

Antes de ejecutar esta consulta, haría falta haber localizado y cargado el *driver* y haberse conectado a la BD. Después de la ejecución, hay que hacer la desconexión de la BD.

Analicemos la consulta:

- Se solicita la creación de objeto `Statement` (`stmt`) asociado a un objeto de conexión activo (`conn`). Como el objeto de la clase `Statement` del JDBC se crea a partir de una conexión activa, también dispone de su propio entorno de ejecución.
- Como lo que se quiere hacer es una consulta, el método que se tiene que utilizar en el objeto `Statement` es `executeQuery` que devuelve un objeto JDBC llamado `ResultSet`.
- Para poder recuperar el resultado de la consulta (en este caso se trata de un único valor), hay que ejecutar el método `next`.
- Para visualizar el resultado que interese del `ResultSet` se puede usar el método `get<Tipo>`, en el que `<Tipo>` se corresponderá con el tipo de datos de la columna que se quiera consultar.
- Si ya no se tiene que utilizar más, se puede cerrar el `ResultSet` explícitamente mediante el método `close`. El objeto de la clase `Statement` también se puede cerrar explícitamente mediante el método `close`.

Además, se ha hecho un sencillo tratamiento de excepciones utilizando un objeto `SQLException` del JDBC que ofrece métodos para obtener información de los errores. En este caso, para obtener un código estándar de error se ha utilizado `getSQLState`. El código estándar de error `02000` quiere decir que la consulta no ha devuelto ningún dato.

4.3.2. SQLJ

SQLJ es una extensión/abstracción del API JDBC que permite introducir sentencias SQL directamente en el código de Java. Las instrucciones de SQLJ dentro de un programa Java se denotan, a diferencia de lo que dice el estándar, con la palabra reservada **`#sql`**.

Ejemplo

Veamos la ejecución de la misma consulta del apartado anterior sobre la tabla `empleados` con su tratamiento de errores correspondiente.

```
try
{
  String qNombre = null;
  #sql {SELECT nombre_empl INTO qNombre FROM empleados WHERE
  codigo_empl=1};
  System.out.println(qNombre);

}
catch (SQLException e)
{
  if (e.getSQLState().equals("02000"))
  {
      System.out.println("No existe ninguna tupla que cumpla la
                          condición");
  }
}
```

A diferencia del ejemplo anterior, en el que la consulta era un parámetro de un método, en el caso de SQLJ se pide directamente la consulta con la palabra reservada `#sql`. Por lo tanto, para recoger el resultado de la consulta hay que utilizar una variable puente, la variable `qNombre`.

Resumen

En este módulo, hemos presentado las sentencias más utilizadas del lenguaje estándar SQL de definición y manipulación de BD relacionales. Como ya hemos comentado en la introducción, SQL es un lenguaje muy potente y eso hace que haya más sentencias y opciones de las que hemos explicado en este módulo. También es cierto, sin embargo, que hemos visto más sentencias de las que algunos sistemas relacionales comerciales ofrecen actualmente. Hemos intentado seguir con la mayor fidelidad el estándar, incluyendo comentarios sólo cuando en la mayoría de SGBD comerciales alguna operación se hacía de manera diferente.

Si se conoce SQL se puede trabajar con cualquier SGBD comercial; sólo habrá que dedicar unas cuantas horas a identificar las variaciones respecto del estándar.

Recordemos cómo será **la creación de una BD con SQL**:

1. Dar nombre a la BD, con la sentencia `CREATE DATABASE`, si hay, o con `CREATE SCHEMA`.

2. Definir las tablas, los dominios, las aserciones y las vistas que formarán la BD.

3. Rellenar las tablas con la sentencia `INSERT INTO`.

Cuando la BD tenga un conjunto de filas, se podrá **manipular**; es decir, actualizar las filas o hacer consultas directamente con SQL interactivo o SQL programado.

Además, podemos utilizar las sentencias de control que hemos explicado. También hemos planteado el problema de la **actualización de las vistas**.

Actividad

1. A buen seguro que siempre habéis querido saber dónde teníais aquellos apuntes que nunca encontrabais. Os proponemos crear una BD para organizar los apuntes y localizarlos rápidamente cuando os apetezca utilizarlos. Tendréis que crear la BD y las tablas; decidir las claves primarias y las foráneas, e insertar filas.

Para almacenar los apuntes tendremos que crear las siguientes tablas:

- Los apuntes: querremos saber su código, la estantería donde se encuentran, el estante y la carpeta, suponiendo que en un estante quepa más de una carpeta. El código de los apuntes lo tendremos que escribir en el lomo de cada carpeta física.

- Los temas: querremos saber su código y nombre. El código de los temas lo tendremos que escribir en cada carpeta física para distinguir temas que tienen el mismo nombre.

- Los autores: sólo querremos saber su código, nombre y apellidos. El código de los autores, que inventaremos nosotros, permitirá distinguir los autores que se llaman igual.

- Temas que hay en cada carpeta de apuntes: en esta tabla, introduciremos el código de los apuntes y el código del tema. En unos apuntes puede haber más de un tema, y un tema puede aparecer repetido en más de una carpeta de apuntes, y se tiene que tener en cuenta este hecho a la hora de escoger la clave primaria.

- Autores de los temas: en esta tabla introduciremos el código del tema y el código del autor. En un tema, puede haber más de un autor, y un autor puede hacer más de un tema, y eso se tiene que tener presente cuando se escoja la clave primaria.

Esperamos que, además de practicar sentencias de definición y manipulación de SQL, esta actividad os resulte útil.

Ejercicios de autoevaluación

Con la actividad hemos insertado filas y, si nos hubiéramos equivocado, también habríamos borrado y modificado alguna fila. Con los ejercicios de autoevaluación practicaremos la parte de sentencias de manipulación que todavía no hemos tratado: las consultas. Los ejercicios que proponemos se harán sobre la BD relacional BDUOC que ha ido saliendo en todo este módulo.

1. Obtened los códigos y los nombres y apellidos de los empleados ordenados alfabéticamente de manera descendente por apellido y, en caso de repeticiones, por nombre.

2. Consultad el código y el nombre de los proyectos de los clientes que son de Barcelona.

3. Obtened los nombres y las ciudades de los departamentos que trabajan en los proyectos número 3 y 4.

4. De todos los empleados que perciben un sueldo de entre 50.000 y 80.000 euros, buscad los códigos de empleado y los nombres de los proyectos que tienen asignados.

5. Buscad el nombre, la ciudad y el teléfono de los departamentos donde trabajan los empleados del proyecto GESCOM.

6. Obtened los códigos y los nombres y apellidos de los empleados que trabajan en los proyectos de precio más alto.

7. Averiguad cuál es el sueldo más alto de cada departamento. Concretamente, hay que dar el nombre y la ciudad del departamento y el sueldo mayor.

8. Obtened los códigos y los nombres de los clientes que tienen más de un proyecto contratado.

9. Averiguad los códigos y los nombres de los proyectos en que todos los empleados que están asignados tienen un sueldo superior a 30.000 euros.

10. Buscad los nombres y las ciudades de los departamentos que no tienen ningún empleado asignado.

Solucionario

1.

```
SELECT apellido_empl, nombre_empl, codigo_empl
FROM empleados
ORDER BY apellido_empl DESC, nombre_empl DESC;
```

2. Con SQL:1989, la solución sería:

```
SELECT p.codigo_proy, p.nombre_proy
FROM proyectos p, clientes c
WHERE c.ciudad = "Barcelona" and c.codigo_cli = p.codigo_cliente;
```

Con SQL:1992, la solución sería:

```
SELECT p.codigo_proy, p.nombre_proy
FROM proyectos p JOIN clientes c ON c.codigo_cli = p.codigo_cliente
WHERE c.ciudad = "Barcelona";
```

3.

```
SELECT DISTINCT e.nombre_dpt, e.ciudad_dpt
FROM empleados e
WHERE e.num_proy IN (3,4);
```

4. Con SQL:1989, la solución sería:

```
SELECT e.codigo_empl, p.nombre_proy
FROM empleados e, proyectos p
WHERE e.sueldo BETWEEN 50000.0 AND 80000.0 and e.num_proy = p.codigo_proy;
```

Con SQL:1992, la solución sería:

```
SELECT e.codigo_empl, p.nombre_proy
FROM empleados e JOIN proyectos p ON e.num_proy = p.codigo_proy
WHERE e.sueldo BETWEEN 50000.0 AND 80000.0;
```

5. Con SQL:1989, la solución sería:

```
SELECT DISTINCT d.*
FROM departamentos d, empleados e, proyectos p
WHERE p.nombre_proy = "GESCOM" and d.nombre_dpt = e.nombre_dpt AND
d.ciudad_dpt = e.ciudad_dpt and e.num_proy = p.codigo_proy;
```

Con SQL:1992, la solución sería:

```
SELECT DISTINCT d.nombre_dpt, d.ciudad_dpt, d.telefono
FROM (departamentos d NATURAL JOIN empleados e) JOIN proyectos p
ON e.num_proy = p.codigo_proy
WHERE p.nombre_proy = "GESCOM";
```

6. Con SQL:1989, la solución sería:

```
SELECT e.codigo_empl, e.nombre_empl, e.apellido_empl
FROM proyectos p, empleados e
WHERE e.num_proy = p.codigo_proy and p.precio = (SELECT MAX(p1.precio)
                                                 FROM proyectos p1);
```

Con SQL:1992, la solución sería:

```
SELECT e.codigo_empl, e.nombre_empl, e.apellido_empl
FROM empleados e JOIN proyectos p ON e.num_proy = p.codigo_proy
WHERE p.precio = (SELECT MAX(p1.precio)
         FROM proyectos p1);
```

7.

```
SELECT nombre_dpt, ciudad_dpt, MAX(sueldo) AS sueldo_maximo
FROM empleados
GROUP BY nombre_dpt, ciudad_dpt;
```

8. Con SQL:1989, la solución sería:

```
SELECT c.codigo_cli, c.nombre_cli
FROM proyectos p, clientes c
WHERE c.codigo_cli = p.codigo_cliente
GROUP BY c.codigo_cli, c.nombre_cli
HAVING COUNT(*) > 1;
```

Con SQL:1992, la solución sería:

```
SELECT c.codigo_cli, c.nombre_cli
FROM proyectos p JOIN clientes c ON c.codigo_cli = p.codigo_cliente
GROUP BY c.codigo_cli, c.nombre_cli
HAVING COUNT(*) > 1;
```

9. Con SQL:1989, la solución sería:

```
SELECT p.codigo_proy, p.nombre_proy FROM proyectos p, empleados e
WHERE e.num_proy = p.codigo_proy
GROUP BY p.codigo_proy, p.nombre_proy
HAVING MIN(e.sueldo) > 30000.0;
```

Con SQL:1992, la solución sería:

```
SELECT p.codigo_proy, p.nombre_proy
FROM empleados e JOIN proyectos p ON e.num_proy = p.codigo_proy
GROUP BY p.codigo_proy, p.nombre_proy
HAVING MIN(e.sueldo)>30000.0;
```

10.

```
SELECT d.nombre_dpt, d.ciudad_dpt FROM departamentos d
WHERE NOT EXISTS (SELECT *
```

```
                FROM empleados e
                WHERE e.nombre_dpt = d.nombre_dpt AND e.ciudad_dpt =
                      d.ciudad_dpt);
```

o bien:

```
SELECT nombre_dpt, ciudad_dpt
FROM departamentos
EXCEPT
SELECT nombre_dpt, ciudad_dpt
FROM empleados;
```

Bibliografia

El SLQ:1992 se define, según si lo buscáis en ISO o en ANSI, en cualquiera de los siguientes documentos:

- Database Language SQL (1992). Documento ISO/IEC 9075:1992. International Organization for Standardization (ISO).

- Database Language SQL (1992). Documento ANSI/X3.135-1992. American National Standards Institute (ANSI).

Date, C. J. (2001). *Introducción a los sistemas de bases de datos* (7.ª ed.). Madrid: Prentice-Hall.

Tenéis incluso una versión más resumida de uno de los mismos autores del libro anterior en el capítulo 4 de este libro. Además, en el apéndice B podéis encontrar una panorámica de SQL:1999.

Date, C. J.; Darwen, H. (1997). *A Guide to the SQL Standard* (4.ª ed.). Reading (Massachusetts): Addison-Wesley.

Los libros que contienen la descripción del estándar ANSI/ISO SQL: 1992 son bastante gruesos y pesados de leer. Este libro es un resumen del oficial.

Otros libros traducidos al castellano de SQL: 1992 que os recomendamos son los siguientes:

Groff, J. R.; Weinberg, P. N. (1998). *LAN Times. Guía de SQL*. Osborne: McGraw-Hill.

Os recomendamos su consulta por su claridad y por los comentarios sobre la manera en que se utiliza el estándar en los diferentes sistemas relacionales comerciales.

Silberschatz, A.; Korth, H. F.; Sudarshan, S. (1998). *Fundamentos de bases de datos*. (3.ª ed.). Madrid: McGraw-Hill.

Podéis encontrar una lectura rápida, resumida, pero bastante completa de SQL en el capítulo 4 de este libro.

Para profundizar en el estudio de SQL: 1999, recomendamos el libro siguiente:

Melton, J.; Simon, A. R. (2002). *SQL: 1999. Understanding Relational Language Components*. San Francisco: Morgan Kaufmann.

Para manteneros informados de las publicaciones de nuevas versiones del estándar, podéis consultar la dirección web de ANSI:

http://www.ansi.org/ y de ISO: http://www.iso.org/

Anexos

Anexo 1. Sentencias de definición de datos

1. Creación de esquemas:

```
CREATE SCHEMA {<nombre_esquema>|AUTHORIZATION <usuario>}
              [<lista_elementos_esquema>];
```

2. Borrado de esquemas:

```
DROP SCHEMA <nombre_esquema> {RESTRICT|CASCADE};
```

3. Creación de BD (esta sentencia no forma parte de SQL estándar):

```
CREATE DATABASE <nombre_BD>;
```

4. Borrado de BD (esta sentencia no forma parte de SQL estándar):

```
DROP DATABASE <nombre_BD>;
```

5. Creación de tablas:

```
CREATE TABLE <nombre_tabla>
   ( <definición_columna>
   [, <definición_columna>...]
   [, <restricciones_tabla>]
);
```

Donde tenemos lo siguiente:

- **definición_columna** es:

```
<nombre_columna> {<tipo_datos>|<dominio>} [<def_defecto>]
               [<restricciones_columna>]
```

- Una de las restricciones de tabla era la definición de claves foráneas:

```
FOREIGN KEY <clave_foránea> REFERENCES <tabla> [(<clave_primaria>)]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
```

6. Modificación de una tabla:

```
ALTER TABLE <nombre_tabla> {<acción_modificar_columna>|
                        <acción_modificar_restricción_tabla>}
```

Donde tenemos lo siguiente:

- **acción_modificar_columna** puede ser:

```
{ADD [COLUMN] <nombre_columna> <def_columna> |
ALTER [COLUMN] <nombre_columna> {SET <def_defecto>|DROP DEFAULT}|
DROP [COLUMN] <nombre_columna> {RESTRICT|CASCADE}}
```

- **acción_modificar_restricción_tabla** puede ser:

```
ADD {<def_restricción>|
DROP CONSTRAINT <nombre_restricción> {RESTRICT|CASCADE}}
```

7. Borrado de tablas:

```
DROP TABLE <nombre_tabla> {RESTRICT|CASCADE};
```

8. Creación de dominios:

```
CREATE DOMAIN <nombre_dominio> [AS] <tipo_datos> [<def_defecto>]
[<restricciones_dominio>];
```

Donde tenemos lo siguiente:

- **def_defecto** tiene el siguiente formato:

```
DEFAULT {<literal>|<función>|NULL};
```

- **restricciones_dominio** tiene el formato siguiente:

```
CONSTRAINT [<nombre_restricción>] CHECK (<condiciones>)
```

9. Modificación de un dominio:

```
ALTER DOMAIN <nombre_dominio> {<acción_modificar_dominio>|
<acción_modificar_restricción_dominio>};
```

Donde tenemos lo siguiente:

- **acción_modificar_dominio** puede ser:

```
SET {<def_defecto>|DROP DEFAULT}
```

- **acción_modificar_restricción_dominio** puede ser:

```
{ADD <restricciones_dominio>|DROP CONSTRAINT <nombre_restricción>}
```

10. Borrado de dominios creados por el usuario:

```
DROP DOMAIN <nombre_dominio> {RESTRICT|CASCADE};
```

11. Definición de una aserción:

```
CREATE ASSERTION <nombre_aserción> CHECK (<condiciones>);
```

12. Borrado de una aserción:

```
DROP ASSERTION <nombre_aserción>;
```

13. Creación de una vista:

```
CREATE VIEW <nombre_vista> [(lista_columnas)] AS (consulta)
[WITH CHECK OPTION];
```

14. Borrado de una vista:

```
DROP VIEW <nombre_vista> {RESTRICT|CASCADE};
```

Anexo 2. Sentencias de manipulación de datos

1. Inserción de filas en una tabla:

```
INSERT INTO <nombre_tabla> [(<columnas>)]
{VALUES ({<v1>|DEFAULT|NULL}, ..., {<vn>|DEFAULT|NULL})|
<consulta>};
```

2. Borrado de filas de una tabla:

```
DELETE FROM <nombre_tabla> [WHERE <condiciones>];
```

3. Modificación de filas de una tabla:

```
UPDATE <nombre_tabla>
SET <nombre_columna> = {<expresión>|DEFAULT|NULL}
    [, <nombre_columna> = {<expresión>|DEFAULT|NULL} ...]
WHERE <condiciones>;
```

4. Consultas de una BD relacional:

```
SELECT [DISTINCT] <nombre_columnas_seleccionar>
FROM <tablas_consultar>
[WHERE <condiciones>]
[GROUP BY <atributos_según_los_que_agrupar>]
[HAVING <condiciones_por_grupos>]
[ORDER BY <nombre_columna_ord> [DESC][, <nombre_columna_ord> [DESC]...]]
```

Anexo 3. Sentencias de control

1. Iniciación de transacciones:

```
SET TRANSACTION {READ ONLY|READ WRITE};
```

2. Finalización de transacciones:

```
{COMMIT|ROLLBACK} [WORK];
```

3. Autorizaciones:

```
GRANT <privilegios> ON <objeto> TO <usuarios>
[WITH GRANT OPTION];
```

4. Desautorizaciones:

```
REVOKE [GRANT OPTION FOR] <privilegios> ON <objeto> FROM <usuarios>
{RESTRICT|CASCADE};
```