Tema 4. Lenguajes de consulta comerciales

4.1. Introducción

En el tema anterior hemos estudiado las operaciones del álgebra relacional y el cálculo relacional, imprescindibles para entender las consultas que podemos realizar en una base de datos relacional. En general, el álgebra relacional está clasificada como un lenguaje de consulta formal procedimental, en el que el usuario indica *cómo* (en qué orden) se deben especificar las operaciones de la consulta para obtener el resultado deseado, mientras que el cálculo relacional es *no procedimental*.

En este tema vamos a estudiar dos lenguajes de consulta comerciales, como son el SQL, basado en álgebra relacional y cálculo relacional de tuplas, y el QBE, basado en cálculo relacional de dominios.

Los SGBDRs comerciales cuentan con una interfaz de lenguaje declarativo de alto nivel en el que se especifica la consulta en SQL (Lenguaje de Consulta Estructurado, *del inglés Structured Query Language*) o QBE, y el propio SGBD es el que se encarga de realizar las optimizaciones necesarias para ejecutar la consulta.

4.2 SQL

SQL fue diseñado e implementado por IBM Research y se ha convertido en un estándar para las bases de datos relacionales. Además, es posible que los programadores de aplicaciones escriban sentencias SQL en las aplicaciones de bases de datos, dando lugar a lo que se conoce como *SQL embebido*.

SQL es un lenguaje de consulta completo, y se puede utilizar como DDL o como DML, ya que cuenta con sentencias para

- Definición de datos
- Consulta de datos
- Actualización de datos

Además, SQL cuenta con mecanismos para la definición de vistas de la base de datos, creación y eliminación de índices y para la incorporación de sentencias SQL en lenguajes de programación de propósito general.

4.2.1. Definición de datos en SQL

SQL utiliza los términos de tabla (table), fila (row) y columna (column) en lugar de los términos relación, tupla y atributo del álgebra relacional. Las órdenes SQL para la definición de datos son `CREATE` (crear), `ALTER` (modificar) y `DROP` (eliminar).

Antes de estudiar estas operaciones, analizaremos brevemente los tipos de dominios disponibles y los conceptos de esquema y catálogo.

4.2.1.1. Tipos de dominios

SQL tiene distintos tipos de datos disponibles, entre los que caben destacar aquellos para la definición de caracteres numéricos, cadenas, fecha y hora.

Los tipos de datos numéricos permiten la definición de números enteros (`INTEGER` o `INT`, `SMALLINT`) o reales (`FLOAT`, `REAL`, `DOUBLE PRECISION`). Incluso es posible definir números reales especificando la precisión en forma de número de decimales (`DECIMAL(i,j)`, donde *i* especifica la precisión, y *j* el número de dígitos que hay detrás del punto decimal).

Los tipos de datos cadena pueden ser de longitud fija (`CHAR(n)` o `CHARACTER(n)`, donde *n* representa el número de caracteres) o de longitud variable (`VARCHAR(n)`, donde *n* es el número máximo de caracteres).

En cuanto a los tipos de fecha y hora, las fechas se definen mediante `DATE` y las horas mediante `TIME`. Las componentes de `DATE` son `YEAR`, `MONTH` y `DAY`, y suelen ir en la forma YYYY-MM-DD. Las componentes de `TIME` son `HOUR`, `MINUTE` y `SECOND`, y suele ir en la forma HH:MM:SS.

Si queremos definir un dominio explícitamente (como un tipo en un lenguaje de programación), podemos usar `CREATE DOMAIN` *`dominio tipo_datos`*. Posteriormente podremos usar este dominio al definir los atributos de las tablas.

4.2.1.2. Esquemas y catálogos

Las primeras versiones de SQL no contemplaban el concepto de esquema de base de datos relacional, y todas las tablas (relaciones) se consideraban parte del mismo esquema. El concepto de esquema SQL se incorporó en SQL2 con el fin de agrupar tablas y otros elementos pertenecientes a la misma aplicación.

Un esquema SQL se identifica con un nombre de esquema y consta de:

- Un *identificador de autorización* que indica al usuario o la cuenta que es propietario del esquema.
- Los *descriptores* de cada elemento del esquema. Entre estos elementos se encuentran las tablas, vistas, dominios y autorizaciones).

Los esquemas se crean mediante la instrucción `CREATE SCHEMA`, que puede contener las definiciones de todos los elementos del esquema. También podemos asignar un nombre y autorización al esquema de forma que se puedan definir sus elementos más adelante.

Por ejemplo, esta instrucción crea un esquema denominado BANCO, cuyo propietario es el usuario con autorización MLOPEZ:

```
CREATE SCHEMA BANCO AUTHORIZATION MLOPEZ;
```

Además del concepto de esquema, SQL2 utiliza el concepto de catálogo, que es una colección con nombre de esquemas SQL.

4.2.1.3. Definición de tablas. Especificación de restricciones

Mediante la orden CREATE TABLE podemos crear cada una de las tablas de nuestra base de datos especificando su nombre, sus atributos y sus restricciones, donde el nombre se puede especificar sin más, o bien después del nombre del esquema seguido de un punto.

```
CREATE TABLE BANCO.CLIENTE... o CREATE TABLE CLIENTE...
```

En primer lugar se especifican los atributos, cada uno con su nombre, su dominio (un tipo de datos) y las restricciones en el caso de que las haya (p.e. ser un valor no nulo).

A continuación se especifican las restricciones de clave, de integridad de entidades y de integridad referencial.

Aquí vemos un ejemplo de creación de una tabla en SQL:

```
CREATE TABLE CLIENTES
      (NOMBRECLI VARCHAR(50) NOT NULL,
      DNICLI VARCHAR(8) NOT NULL,
      DOMICILIO VARCHAR(50) NOT NULL,
      PRIMARY KEY (DNICLI)
      ON DELETE CASCADE ON UPDATE CASCADE
      );
CREATE TABLE CUENTAS
      (NUMEROCTA VARCHAR(10) NOT NULL,
      SALDO DECIMAL(12,2)     NOT NULL,
      NOMBRESUC VARCHAR(50) NOT NULL,
      PRIMARY KEY (NUMEROCTA)
      ON DELETE CASCADE ON UPDATE CASCADE
      );
CREATE TABLE CTACLI
      (DNICLI VARCHAR(8) NOT NULL,
      NUMEROCTA VARCHAR(20) NOT NULL,
      PRIMARY KEY (DNICLI, CTACLI),
      FOREIGN KEY (DNICLI) REFERENCES CLIENTES(DNICLI),
      FOREIGN KEY (NUMEROCTA) REFERENCES
```

```
                                        CUENTAS(NUMEROCTA)
        );
```

La forma más sencilla de especificar en la definición de una tabla que un atributo o un conjunto de atributos son clave es mediante una declaración `PRIMARY KEY` en la definición de la tabla. Los atributos que forman la clave irán encerrados entre paréntesis formando una lista separada por comas. En el ejemplo anterior, en la tabla `CLIENTES`, `DNICLI` ha sido definido como clave primaria mediante `PRIMARY KEY (DNICLI)`. Como una clave no puede tomar un valor nulo, los atributos que formen la clave deberán estar definidos como no nulos `(NOT NULL)`.

El proceso de definición de claves externas es parecido al de definición de claves primarias salvo que se utiliza `FOREIGN KEY` en lugar de `PRIMARY KEY`. No obstante, en la definición de una clave externa también hay que especificar a qué atributo se está haciendo referencia. Eso se hace escribiendo `REFERENCES` seguido del nombre de la tabla, y entre paréntesis, el nombre del atributo al que se hace referencia. En la tabla `CTACLI` del ejemplo anterior, `DNICLI` está definido como una clave externa y hace referencia al atributo `DNICLI` de `CLIENTES`.

La definición de clave primarias y claves externas especifican restricciones. A estas restricciones se les puede asignar un nombre, el cual debe ser único en todo el esquema. La utilidad de asignar nombres a las restricciones radica en que quedan identificadas para cuando posteriormente se decida modificarlas.

Para especificar restricciones basadas en valores, podemos incluir en la definición de un atributo la palabra reservada `CHECK`, seguida de una condición entre paréntesis, que deberá ser verificada por los valores del atributo. Si la restricción es a nivel de tupla, la cláusula `CHECK` se introduce en la definición de la tabla, como las restricciones de clave primaria o externa.

Las restricciones `CHECK` tienen el inconveniente de que cuando hacen referencia a relaciones mediante una subconsulta y hay cambios en las relaciones referenciadas pueden dejar de cumplirse. Para evitar este problema, disponemos de las *aserciones*, restricciones que siempre deben ser verdaderas, y se comprueban cada vez que se modifica cualquiera de las relaciones mencionadas en ellas.

4.2.1.4 Modificación de esquemas

La definición de una tabla se puede modificar mediante la orden `ALTER TABLE`, y las posibles acciones que se pueden realizar son:

- Incorporación o eliminación de una columna (atributo)
- Modificación de la definición de una columna
- Incorporación o eliminación de las restricciones de una tabla

Por ejemplo, si queremos añadir a la tabla `CLIENTES` la ciudad en la que viven, utilizaremos la orden

```
ALTER TABLE BANCO.CLIENTES ADD CIUDAD VARCHAR(20);
```

Una vez modificada la definición de la tabla, se completan con los valores correspondientes utilizando la orden `UPDATE` que veremos más adelante.

Para eliminar una columna tendremos que elegir `CASCADE` o `RESTRICT` para especificar la forma de eliminación. Si se elige `CASCADE`, todas las restricciones y vistas creadas a partir de dicha columna se eliminarán automáticamente del esquema. Por el contrario, si se elige `RESTRICT` sólo se eliminará la columna si no hay ninguna vista ni restricción sobre dicha columna.

Por ejemplo, esta orden elimina el campo que hemos añadido antes:

```
ALTER TABLE BANCO.CLIENTES DROP CIUDAD CASCADE;
```

Para modificar la definición de una columna de la tabla utilizaremos ALTER. Por ejemplo, para eliminar la restricción definida sobre la columna DOMICILIO de CLIENTES que impide que haya valores nulos, escribiríamos

```
ALTER TABLE BANCO.CLIENTES ALTER DOMICILIO DROP NOT NULL;
```

A la hora de eliminar algo, puede darse el caso de que lo que se quiere eliminar contenga elementos. En SQL es posible eliminar sólo en el caso de que no existan elementos, o bien forzar la eliminación, independientemente de si contiene elementos o no, y propagar los cambios en consecuencia. Para ello se utilizan `RESTRICT` y `CASCADE`, respectivamente.

La orden `DROP TABLE CLIENTES RESTRICT;` elimina la tabla `CLIENTES` si no hay ninguna restricción definida sobre ella en otro componente del esquema. La orden `DROP SCHEMA BANCO CASCADE;` elimina el esquema `BANCO` de la base de datos y todas sus tablas y demás componentes.

4.2.1.5. El diccionario de datos

Todos los SGBD relacionales gestionan su propio *diccionario de datos* (aunque sólo sea la descripción de las tablas presentes en la base de datos) usando un esquema relacional. Las tablas que contienen información sobre las tablas contienen los *metadatos* y constituyen el *catálogo* de la base de datos.

El diccionario de datos podría manipularse a nivel de actualización de datos como otra tabla, pero esto no sería adecuado. Lo que si podemos hacer es *consultarlo* igual que otras tablas de las base de datos.

El estándar de diccionario de datos de SQL-2 no es seguido estrictamente en los SGBD relacionales, debido sobre todo a la información sobre cuestiones de estructuras de almacenamiento de la información, pero ofrece una plantilla que suele ser respetada por estos sistemas. El estándar tiene dos niveles, DEFINITION_SCHEMA, que describe todas las estructuras de la base de datos, e INFORMATION_SCHEMA, que contiene vistas sobre el DEFINITION_SCHEMA y ofrece una interfaz con el diccionario de datos. Como ejemplo de estas vistas, tenemos DOMAINS, TABLES, VIEWS, o COLUMNS. La vista COLUMNS incluye una fila para cada columna de la base de datos, con información sobre el nombre de la tabla a la que pertenece, su

posición dentro de la misma, su dominio o tipo de datos, el valor predeterminado (si lo tiene), si admite nulos o no, y otra información adicional.

4.2.2 Consulta de datos en SQL

4.2.2.1. Estructura básica de una consulta SQL

La estructura básica de una orden SQL consta de tres cláusulas: `SELECT`, `FROM` y `WHERE`

La cláusula `SELECT` se corresponde con la operación de proyección del álgebra relacional, y se utiliza para indicar cuáles son los atributos (separados por comas) que se desea que aparezcan en el resultado de la consulta

La cláusula `FROM` corresponde a la operación de producto cartesiano del álgebra relacional, y se utiliza para indicar cuáles son las relaciones (separadas por comas) que participan en la consulta.

La cláusula `WHERE` se corresponde con los predicados de selección del álgebra relacional, y dichos predicados hacen referencia a atributos de relaciones especificadas en la cláusula `FROM`.

Una consulta típica en SQL tiene esta forma:

SELECT A_1, A_2, ..., A_n

FROM R_1, R_2, ..., R_m

WHERE P

donde cada A_i representa a un atributo, cada R_i representa a una relación y `P` es el predicado o condición de la consulta.

Esta consulta equivale a la siguiente expresión del álgebra relacional:

$\Pi_{A1, A2, ..., An}(\sigma_P (R_1 \times R_2 \times ... \times R_m))$

La lista de atributos puede sustituirse por un asterisco (*) para seleccionar todos los atributos de todas las relaciones que aparecen en la cláusula `FROM`, y la claúsula `WHERE` puede omitirse, por lo que el predicado siempre será verdadero, y se seleccionarán todas las filas de todas las tablas que aparezcan en la cláusula `FROM`.

Por ejemplo, para saber cuáles son los nombres de todos los clientes del banco escribiríamos la sentencia SQL siguiente:

```
SELECT nombreCli
FROM clientes;
```

Y para saber cuáles son los nombres de los empleados que trabajan en la sucursal de "`Sol`", escribiríamos la sentencia SQL siguiente:

```
SELECT nombreEmp
FROM empleados
WHERE nombreSuc = "Sol";
```

En la cláusula SELECT también pueden aparecer expresiones con operadores aritméticos (+, -, *, /) o de otros tipos, por lo que para ver el saldo de las cuentas en pesetas en lugar de euros, escribiríamos

```
SELECT numeroCta, saldo * 166.386
FROM cuentas;
```

4.2.2.2. Gestión de duplicados y valores nulos

Si hubiese dos empleados con el mismo nombre, aparecerían valores duplicados. Si estamos interesados en que no aparezcan duplicados se debe utilizar `SELECT DISTINCT`. No obstante, si queremos que se conserven explícitamente todos los duplicados escribiremos `SELECT ALL` o `SELECT`.

Por ejemplo, para saber cuáles son los nombres de los empleados que trabajan en la sucursal de "`Sol`" sin que aparezcan duplicados escribiríamos la sentencia SQL siguiente:

```
SELECT DISTINCT nombreEmp
FROM empleados
WHERE nombreSuc = "Sol";
```

Cuando en una consulta necesitamos manejar valores nulos, tenemos que usar las condiciones `IS NULL` e `IS NOT NULL`. Por ejemplo, para ver los nombres de los clientes que no tienen domicilio, usaríamos esta instrucción SQL:

```
SELECT nombreCli
FROM clientes
WHERE domicilio IS NULL;
```

4.2.2.3. Combinación de relaciones y otros aspectos básicos

SQL no dispone de una operación directa para combinar tablas mediante el producto natural, pero como éste es un producto cartesiano con una selección, es relativamente sencillo escribir una orden SQL para realizar un producto natural.

Recordemos que en el álgebra relacional, para realizar el producto natural a partir del producto cartesiano para y obtener los nombres de todos los empleados que trabajan en todas las ciudades podríamos escribir:

$\Pi_{\text{empleados.nombreEmp}}$ ($\sigma_{\text{sucursales.nombreSuc = empleados.nombreSuc}}$ (sucursales x empleados))

La sentencia SQL equivalente a esta expresión del álgebra relacional sería:

```
SELECT empleados.nombreEmp
FROM sucursales, empleados
WHERE sucursales.nombreSuc = empleados.nombreSuc;
```

aunque se podría haber obtenido el mismo resultado a partir únicamente de la tabla *empleados*.

Obsérvese que se ha seguido la notación *nombreRelacion*. *nombreAtributo* para eliminar posibles ambigüedades.

Para poder utilizar más de un predicado en una cláusula WHERE se pueden utilizan los conectores lógicos AND, OR y NOT en lugar de los conectores del álgebra relacional $\wedge$, $\vee$, y $\neg$.

Por ejemplo, para saber cuáles son los nombres de los empleados que trabajan en alguna de las sucursales de la ciudad de "Madrid", escribiríamos:

```
SELECT empleados.nombreEmp
FROM sucursales, empleados
WHERE sucursales.nombreSuc = empleados.nombreSuc AND
     sucursales.ciudadSuc = "Madrid";
```

SQL incluye el operador BETWEEN para simplificar las cláusulas WHERE que se refieren a valores que sea mayor o igual que uno y menor o igual que otro, por lo que para saber cuáles son los números de las cuentas que tienen un saldo comprendido entre 20000 y 50000 podemos escribir

```
SELECT numeroCta
FROM cuentas
WHERE saldo BETWEEN 20000 AND 50000;
```

en lugar de

```
SELECT numeroCta
FROM cuentas
WHERE saldo >= 20000 AND saldo <= 50000;
```

Análogamente, podemos utilizar el operador NOT BETWEEN.

Por último, si queremos realizar consultas SQL comparando ciertos atributos con valores de cadena que utilicen comodines utilizaremos el operador LIKE, teniendo en cuenta que el carácter de comodín para un carácter es el subrayado (_) y que el carácter de comodín para un conjunto de caracteres es el tanto por ciento (%).

De esta forma, para saber cuáles son los nombres de los clientes que viven en la calle "Gibraltar español" escribiríamos lo siguiente:

```
SELECT nombreCli
FROM clientes
WHERE domicilio LIKE "Gibraltar español%";
```

SQL permite la búsqueda de desigualdades con el operador NOT LIKE.

SQL permite al usuario especificar criterios para la presentación ordenada de resultados, y lo hace mediante la cláusula `ORDER BY`, que en el caso de que se utilice debe ir después de la cláusula `WHERE` (en caso de que exista).

Por ejemplo, para mostrar una lista ordenada alfabéticamente de los clientes que tienen alguna cuenta en la sucursal "`Sol`", escribiríamos lo siguiente:

```
SELECT DISTINCT nombreCli
FROM clientes, cuentas, ctacli
WHERE clientes.dniCli = ctacli.dniCli AND
      cuentas.numeroCta = ctacli.numeroCta AND
      cuentas.nombreSuc = "Sol"
ORDER BY nombreCli;
```

Por omisión, SQL lista las tuplas en orden ascendente (`ASC`), pero para especificar un orden descendente, escribiríamos `DESC`, de esta forma:

```
ORDER BY col1 [ASC|DESC] [, col2 [ASC|DESC]], ...
```

En SQL, la combinación de relaciones puede realizarse también de otras formas ya vistas en el álgebra relacional. A continuación veremos el `INNER JOIN`, `NATURAL INNER JOIN` y los `OUTER JOIN`. Para ilustrar estas operaciones nos basaremos en las tablas de Empleados y Sucursales.

El `INNER JOIN` o reunión interna es una forma compacta de combinar relaciones atendiendo a una condición. Su sintaxis es

```
Tabla1 INNER JOIN Tabla2 ON condicion
```

Para ilustrar su uso, si queremos obtener el nombre de los empleados y la ciudad en la que está la sucursal en la que trabajan utilizando un `INNER JOIN` escribiremos

```
SELECT nombreEmp, ciudadSuc
FROM Empleados INNER JOIN Sucursales ON
     Empleados.nombreSuc = Sucursales.nombreSuc;
```

En la reunión interna hay que especificar explícitamente la condición de *join* de la consulta, la cual no tiene por que ser siempre ligando los atributos que relacionan las tablas que participan en la consulta (véase sección 4.6). Sin embargo, como gran parte de las consultas que se realizan sobre una base de datos suele combinar las tablas utilizando los atributos comunes SQL ofrece el `NATURAL INNER JOIN` para realizar el producto natural de dos tablas.

La consulta siguiente sería equivalente a la anterior, salvo que la que se muestra a continuación lo hace con `NATURAL INNER JOIN`.

```
SELECT nombreEmp, ciudadSuc
FROM Empleados NATURAL INNER JOIN Sucursales;
```

Tanto la reunión interna como la natural interna combinan dos relaciones de forma que en el resultado sólo aparecen tuplas que presentan valores si en cada tabla hay tuplas con valores coincidentes. En cambio, las reuniones externas relajan esta condición porque permiten que en el resultado haya tuplas que se correspondan a tuplas de una tabla que no tienen tuplas relacionadas en la otra tabla.

Existen tres tipos de reuniones externas, que son la reunión externa izquierda (`LEFT OUTER JOIN`), la reunión externa derecha (`RIGHT OUTER JOIN`) y la reunión externa completa (`FULL OUTER JOIN`). A continuación se muestra cada una de ellas y su sintaxis, similar a la de `INNER JOIN`.

La reunión externa izquierda se calcula como la reunión interna, pero además, también se incluirán en el resultado las tuplas de la relación de la izquierda que no estén relacionadas con alguna tupla de la relación de la derecha, rellenándose los valores de los atributos correspondientes a la relación de la derecha con nulos. La consulta siguiente realiza la reunión externa izquierda de Sucursales de `Horseneck` con `Empleados`

```
SELECT Sucursales.nombreSuc, nombreEmp
FROM Sucursales LEFT OUTER JOIN Empleados ON
     Sucursales.nombreSuc = Empleados.nombreSuc
WHERE ciudadSuc = "Horseneck";
```

El resultado de esta consulta sería $$

Sucursales.nombreSuc	nombreEmp
Perrydge	Hansen
Perrydge	Dubitzky
Mianus	Henson
Round Hill	Null

Análogamente, la reunión externa derecha (`RIGHT OUTER JOIN`) funciona de forma similar que la reunión externa izquierda salvo considerando las tuplas de la relación que aparece en la derecha.

Por último, la reunión externa completa es la operación más relajada de todas, ya devuelve las tuplas de ambas relaciones aunque no tengan tuplas relacionadas en la otra tabla. Por ejemplo, la consulta siguiente realiza la reunión externa completa de Sucursales y Empleados combinando las tuplas de Sucursales y Empleados. Si hay tuplas relacionadas las añade al resultado. Si no es así, añade la tupla no relacionada y completa los valores con nulos. (No obstante, en este caso y por la integridad referencial definida en Empleados no habrá tuplas en Empleados que no tengan tuplas relacionadas en Sucursales, por lo que la reunión externa completa devolverá el mismo resultado que una reunión externa izquierda.)

```
SELECT Sucursales.nombreSuc, nombreEmp
FROM Sucursales FULL OUTER JOIN Empleados ON
     Sucursales.nombreSuc = Empleados.nombreSuc;
```

4.2.2.4. Agrupación y funciones de agregación

SQL permite la posibilidad de calcular funciones en grupos de tuplas utilizando la cláusula `GROUP BY`. Para formar estos grupos, se utilizan todos los atributos que aparecen en la cláusula `GROUP BY`, y cada una de las tuplas que tienen el mismo valor en cada uno de los atributos que se especifican en la cláusula `GROUP BY` se colocan en el mismo grupo.

SQL incluye funciones para calcular:

- Promedio: `AVG`
- Mínimo: `MIN`
- Máximo: `MAX`
- Total: `SUM`
- Cuenta: `COUNT`

A estas operaciones se les denomina funciones de agregación, ya que operan sobre grupos de tuplas, y el resultado de estas funciones es un valor único.

Por ejemplo, para conocer el activo medio por ciudades escribiríamos lo siguiente:

```
SELECT ciudadSuc, AVG(activo)
FROM sucursales
GROUP BY ciudadSuc;
```

Sobre el resultado de una agrupación es posible establecer una condición, pero ésta actuará a nivel de grupo, y no a nivel de tupla, como lo hace el `WHERE`. Las condiciones a nivel de grupo se definen mediante la cláusula `HAVING`, y se especifica detrás de `GROUP BY`. Así, si queremos saber cuáles son las medias de los activos de las sucursales por ciudades para aquellas en que la media sea superior a 20000, escribiríamos

```
SELECT ciudadSuc, AVG(activo)
FROM sucursales
GROUP BY ciudadSuc
HAVING AVG(activo) > 20000;
```

Cuando se utilizan grupos hay que tener en cuenta que en la cláusula `SELECT` sólo pueden aparecer los atributos por los que se está realizando la agrupación, o bien funciones de agregación.

Para saber el saldo máximo de las cuentas de las sucursales por ciudad agruparíamos por ciudad como muestra la consulta siguiente

```
SELECT ciudadSuc, MAX(saldo)
FROM Sucursales, Cuentas
```

```
WHERE Sucursales.nombreSuc = Cuentas.nombreSuc
GROUP BY ciudadSuc;
```

Para saber la ciudad en la que están las sucursales con más de una cuenta necesitamos conocer primero cuáles son las sucursales que tienen más de una cuenta, y luego utilizar esto para obtener en qué ciudad está

```
SELECT ciudadSuc
FROM Sucursales
WHERE nombreSuc IN
     (SELECT nombreSuc
     FROM Cuentas
     GROUP BY nombreSuc
     HAVING COUNT(numeroCta) > 1);
```

A veces, es necesario tratar toda la relación como si fuese un solo grupo, por lo que en estos casos no será necesario utilizar `GROUP BY`. Por ejemplo, para obtener el saldo medio, máximo y mínimo de las cuentas escribiríamos

```
SELECT AVG(saldo), MAX(saldo), MIN(saldo)
FROM Cuentas;
```

Otra utilidad que se desprende de considerar toda la tabla como un grupo puede ser para conocer el número de tuplas de una relación. Para ello, podríamos escribir lo siguiente para conocer el número de clientes del banco

```
SELECT COUNT(*)
FROM Clientes;
```

Para conocer el número de ciudades en los que el banco tiene sucursales escribiríamos

```
SELECT COUNT (DISTINCT ciudadSuc)
FROM Sucursales
```

y se debe utilizar `DISTINCT` porque si no se obtendría simplemente el número de sucursales que tiene el banco, ya que se repetirían las ciudades en las que están las sucursales de una misma ciudad.

Por último, para conocer el nombre de las ciudades que tienen más de dos sucursales lo podríamos hacer con una consulta anidada

```
SELECT ciudadSuc
FROM Sucursales S
WHERE (SELECT COUNT (*)
     FROM Sucursales
     WHERE S.ciudadSuc = Sucursales.ciudadSuc) > 2;
```

4.2.2.5. Operaciones sobre conjuntos

SQL incorpora las operaciones `UNION` (unión), `INTERSECT` (intersección) y `MINUS` (diferencia) que operan sobre relaciones o tablas y se corresponden con las operaciones $\cup$, $\cap$ y - del álgebra relacional.

Veamos cómo podemos utilizar estas operaciones en SQL.

Por ejemplo, para saber cuáles son los nombres de todas las personas relacionadas con el sistema bancario escribiríamos:

```
SELECT nombreEmp
FROM empleados
UNION
SELECT nombreCli
FROM clientes;
```

Para encontrar todos los nombres que aparecen como empleado y como cliente escribiríamos:

```
SELECT nombreEmp
FROM empleados
INTERSECT
SELECT nombreCli
FROM clientes;
```

Por último para encontrar todos los nombres que aparecen como empleado pero no como cliente, escribiríamos:

```
SELECT nombreEmp
FROM empleados
MINUS
SELECT nombreCli
FROM clientes;
```

Por omisión, las operaciones `UNION`, `INTERSECT` y `MINUS` eliminan las tuplas duplicadas, pero si no se desea que se eliminen las tuplas duplicadas, escribiremos `UNION ALL`, `INTERSECT ALL`, y `MINUS ALL`.

4.2.2.6. Creación de alias y variables de tupla

Tal y como vimos en el tema anterior, a veces es necesario renombrar una relación de forma que se puedan comparar dos tuplas de la misma relación. Este era el caso de conocer cuáles son los nombres de los empleados que trabaja en la misma sucursal que "García".

Para crear alias sobre tablas en SQL, también llamados *variables de tupla*, basta con poner en la cláusula `FROM` el nombre del alias a continuación del nombre de la tabla, como se muestra en este ejemplo

```
SELECT E.nombreEmp
FROM Empleados E, Empleados Ebis
WHERE Ebis.nombreEmp = "García" AND
          Ebis.nombreSuc = E.nombreSuc;
```

No obstante, esta consulta también se podría haber realizado con una consulta anidada, las cuales se describen más adelante.

Otro ejemplo podría ser el de obtener las sucursales que tienen un activo superior a cualquier sucursal situada en `Almería`. Para ello, necesitamos hacer referencia dos veces a la misma tabla de forma que se podría hacer de esta forma

```
SELECT E.nombreEmp
FROM Sucursales S1, Sucursales S2
WHERE S1.ciudadSuc = "Almería" AND
          S2.Activo > S1.Activo;
```

Si lo que queremos es cambiar los nombres de los atributos que resultan de una consulta, escribiremos en la cláusula `SELECT` el modificador `AS` entre el antiguo nombre del atributo y el nuevo nombre del atributo, tal y como se indica en este ejemplo

```
SELECT nombreSuc AS nombreDeSucursales
FROM sucursales;
```

4.2.2.7. Consultas anidadas y pertenencia a conjuntos

En el predicado de selección de tuplas (claúsula `WHERE`), además de poder comparar un atributo con un valor, podemos realizar la comparación con un conjunto de valores, es decir podemos especificar explícitamente un conjunto con los valores de la comparación, en lugar de realizar varias comparaciones *OR*. Para ello utilizaremos el operador de pertenencia a conjuntos `IN`, o bien el operador `NOT IN`, si lo que queremos comprobar es la no pertenencia al conjunto.

Por ejemplo, para saber cuáles son los nombres de los empleados que trabajan en las sucursales de "`Sol`" o "`Castellana`" podemos escribir

```
SELECT nombreEmp
FROM empleados
WHERE nombreSuc IN ("Sol", "Castellana");
```

en lugar de escribir

```
SELECT nombreEmp
FROM empleados
WHERE nombreSuc = "Sol" OR
                    nombreSuc = "Castellana";
```

A esta forma de especificación de parámetros se le denomina conjuntos explícitos.

A la vista de este ejemplo, cabe pensar que se podría crear una consulta que utilizase como predicado de comparación otra consulta, es decir, anidar una consulta dentro de otra. A la consulta interna se le denomina consulta anidada, y a la que utiliza el resultado de la ejecución de la consulta anidada se le denomina consulta externa.

Para ello, veamos otra forma de escribir la consulta que devuelve cuáles son los nombres de los empleados que trabajan en las sucursales de la ciudad de "Madrid".

```
SELECT nombreEmp
FROM empleados
WHERE nombreSuc IN (SELECT nombreSuc
                FROM sucursales
                WHERE ciudadSuc = "Madrid");
```

En las consultas anidadas es posible que se puedan originar ambigüedades entre los nombres de los atributos si hay atributos con el mismo nombre en la consulta externa y en la consulta interna. En este caso, y dado que los atributos que no están precedidos por un nombre de relación se refieren a la consulta anidada más interna, se trata de calificar los nombres de los atributos con el nombre de las tablas correspondientes.

4.2.2.8. Consultas correlacionadas

Un caso especial de consultas anidadas son las *consultas correlacionadas*. Se trata de consultas en las que la consulta anidada se ejecuta una vez para cada tupla de consulta externa. Esto ocurre cuando en el WHERE de una subconsulta se hace referencia a un atributo que está en la consulta externa. Por ejemplo, supongamos que tenemos las tablas siguientes

Personal

Nombre	Apellidos	NSS	AñoNacimiento	Sexo	Salario	NSSSuperv
Pedro	Márquez	1	1954	H	1200	2
Isabel	Fernández	2	1972	M	1150	3
María	Yagüe	3	1963	M	2200	null
Juan	Martín	4	1971	H	1050	3

Familia

Nombre	NSSP	AñoNacimiento	Sexo	Parentesco
Juana	1	1974	M	Hija
Manuel	1	1976	H	Hijo
Margarita	1	1958	M	Esposa
María	3	1993	M	Hija
Luis	3	1963	H	Esposo

Si queremos saber el nombre y apellidos de los empleados que tienen familiares con su mismo nombre y sexo podríamos escribir

```
SELECT Nombre, Apellidos
FROM Personal
WHERE Nombre IN
     (SELECT Nombre
     FROM Familia
     WHERE NSS=NSSP AND
          Personal.Nombre = Familia.Nombre AND
          Personal.Sexo = Familia.Sexo);
```

o bien

```
SELECT Personal.Nombre, Personal.Apellidos
FROM Personal, Familia
WHERE NSS=NSSP AND
     Personal.Nombre = Familia.Nombre AND
     Personal.Sexo = Familia.Sexo);
```

En algunos casos es necesario que en las consultas anidadas se compruebe si el resultado de la subconsulta es vacío o no. Esto se realiza utilizando la función `EXISTS` en la cláusula `WHERE`. Por ejemplo, para el caso anterior se trataría de obtener registros de empleado para los que "existen" familiares con su mismo nombre y sexo.

```
SELECT Nombre, Apellidos
FROM Personal
WHERE EXISTS
     (SELECT *
     FROM Familia
     WHERE NSS=NSSP AND
          Personal.Nombre = Familia.Nombre AND
          Personal.Sexo = Familia.Sexo);
```

Si ahora queremos saber el nombre de las personas que no tienen familiares escribiríamos

```
SELECT Nombre, Apellidos
FROM Personal
WHERE NOT EXISTS
     (SELECT *
     FROM Familia
     WHERE NSS=NSSP);
```

A veces es necesario que alguna de las condiciones del predicado de la consulta realicen comparaciones sobre conjuntos, como pueden ser cuáles son los nombres de las sucursales que tienen un activo superior a cualquiera de las sucursales de la ciudad de "`Almería`", o bien el nombre de las sucursales que tengan un activo mayor que el de todas las sucursales de la ciudad de "`Almería`".

Para ello utilizaremos `SOME` y `ALL` de esta forma.

En primer lugar, para conocer los nombres de las sucursales que tienen un activo superior a cualquiera de las sucursales de la ciudad de "Almería" utilizaremos > `SOME` de esta forma

```
SELECT nombreSuc
FROM sucursales
WHERE activo > SOME (SELECT activo
                     FROM sucursales
                     WHERE ciudadSuc = "Almería");
```

No obstante, y tal y como vimos en la sección anterior, esto también se puede realizar mediante el uso de alias de tabla (variables de tupla).

Si ahora lo que quisiésemos fuese el nombre de las sucursales que tienen un activo superior al de todas las sucursales de la ciudad de "Almería" utilizaremos > ALL de esta forma

```
SELECT nombreSuc
FROM sucursales
WHERE activo > ALL (SELECT activo
                    FROM sucursales
                    WHERE ciudadSuc = "Almería");
```

Ni que decir tiene que SQL también permite las construcciones < SOME, <= SOME, >= SOME, <> SOME, = SOME, < ALL, <= ALL, >= ALL, = ALL y <> ALL.

Por tanto IN, SOME y ALL permiten comprobar un único valor con un conjunto de valores.

4.2.2.9. Vistas. Actualización de vistas

Una vista SQL es una tabla derivada a partir de otras. Estas tablas pueden ser tablas base (tablas de las definidas originalmente en los esquemas) o vistas. Una vista no tiene por que estar almacenada físicamente, sino que puede recuperar la información bajo demanda. Así, las vistas son consideradas como *tablas virtuales*, a diferencia de las tablas base que sí almacenan realmente los registros. Este hecho limita la actualización de las vistas, sobre todo cuando se realiza la combinación de tablas o cuando se realizan operaciones de agregación. No obstante, y a efectos de consulta, una vista y una tabla no ofrecen diferencia alguna, y esto es lo que hace que las vistas sean un mecanismo muy utilizado, dada la utilidad que presentan para la personalización de información (p.e. ocultar información de las tablas base) o para la simplificación de consultas.

En SQL las vistas se crean mediante la orden

```
CREATE VIEW nombreVista AS ExpresionSELECT;
```

El ejemplo siguiente crearía una vista que devuelve el nombre, teléfono y la ciudad en la que trabajan cada uno de los empleados del banco.

```
CREATE VIEW EmpleCiudad
AS SELECT nombreEmp, telefono, ciudadSuc
     FROM Empleados, Sucursales
     WHERE Empleados.nombreSuc = Sucursales.nombreSuc;
```

La vista queda formada por las columnas que aparezcan en el SELECT que defina la vista, aunque también es posible especificar el nombre que van a tener las columnas de la vista indicándolas explícitamente, como muestra el ejemplo siguiente que crea una vista en la que se obtiene el DNI de cada cliente y el saldo total de sus cuentas.

```
CREATE VIEW ClientesSumaSaldo (DNICli, SaldoTotal)
AS SELECT DNICli, SUM(Saldo)
```

```
FROM Clientes, CtaCli, Cuentas
WHERE Clientes.DNICli = CtaCli.DNICli and
      CtaCli.numeroCta = Cuentas.numeroCta
GROUP BY Clientes.DNICli;
```

Así, el esquema de la primera vista sería `EmpleCiudad(nombreEmp, telefono, ciudadSuc)` y el de la segunda sería `ClientesSumaSaldo (DNICli, SaldoTotal)`.

La eliminación de vistas se hace mediante la orden

```
DROP VIEW nombreVista
```

Las órdenes `DROP VIEW EmpleCiudad` y `DROP VIEW ClientesSumaSaldo` eliminarían las dos vistas que hemos definido.

Hemos dicho que las vistas son realmente definiciones de consultas, y no se almacenan como conjuntos de filas en la base de datos. Sin embargo, hay condiciones en las que se pueden *actualizar* las vistas. Realmente, estas modificaciones se efectuarán sobre la tabla de la que procede la vista. Para saber si una vista es actualizable o no, debemos tener en cuenta lo siguiente:

- Una vista definida sobre una sóla tabla es actualizable si los atributos de la vista incluyen la clave primaria de dicha tabla y todos los atributos que no pueden ser nulos.
- Una vista definida sobre la combinación de dos o más tablas no es actualizable.
- Una vista definida usando agrupación y funciones de agregación no es actualizable.

4.2.3. Modificación de datos en SQL

Hasta ahora nos hemos limitado a recuperar información de la base de datos. SQL dispone de órdenes para añadir, eliminar y actualizar los datos de una base de datos mediante las instrucciones `INSERT`, `DELETE` y `UPDATE`, respectivamente. A continuación se estudia cada una de ellas.

4.2.3.1. Inserción

En SQL los datos son introducidos en las tablas mediante la orden INSERT, la cual se puede utilizar para insertar un solo registro en una tabla o bien para insertar en una tabla el resultado de una consulta.

Para insertar un registro en una tabla utilizaremos

```
INSERT INTO nombreTabla VALUES (valor1, valor2, ...);
```

Por ejemplo, la expresión siguiente crearía un nuevo empleado llamado `Harry` en la sucursal de "`Los Pinos`". `Harry` tiene el DNI `16` y su teléfono es `161616`.

```
INSERT INTO Empleados
```

```
VALUES ("Harry", "16", "161616", "Los Pinos");
```

En el caso de que la tabla tuviese definida alguna columna que permitiese valores nulos o asignase valores predeterminados en el caso de no indicar ningún valor, se podrían especificar los nombres de columna a los que afectan los datos que se quieren insertar. Por ejemplo, si en la tabla de empleados se permitiese que el teléfono fuese nulo podríamos escribir para un empleado de la sucursal de `Los Pinos` que se llamase `Mukos` y con DNI `17` lo siguiente

```
INSERT INTO Empleados  (nombreEmp, dniEmp, nombreSuc)
VALUES ("Mukos", "17", "Los Pinos");
```

Por ultimo, una sentencia `INSERT` también puede insertar los valores que procedan de la ejecución de una consulta en lugar de especificar explícitamente los valores que se quieren introducir. En este caso la sintaxis es

```
INSERT INTO nombreTabla ExpresionSELECT;
```

Por ejemplo, podemos crear una tabla temporal para guardar el número de empleados para cada sucursal e introducir en ella el resultado de una consulta, como se muestra a continuación

```
CREATE TABLE NumeroEmple
(Sucursal VARCHAR(50),
Empleados INTEGER);
INSERT INTO NumeroEmple
     SELECT NombreSuc, COUNT(nombreEmp)
     FROM Empleados
     GROUP BY NombreSuc;
```

4.2.3.2. Eliminación

La orden SQL para eliminar datos de una tabla es `DELETE` y elimina los registros de una tabla que satisfacen una condición. Su sintaxis es parecida al `SELECT`, salvo que no especifica atributos para la proyección.

```
DELETE
FROM Tabla
WHERE Condicion;
```

Por ejemplo, la orden siguiente eliminaría el registro del empleado `Mukos`

```
DELETE
FROM Empleados
WHERE nombreEmp = "Mukos";
```

Como condición se puede establecer cualquier predicado por complejo que sea, o incluso se pueden especificar subconsultas. Por ejemplo, la expresión siguiente elimina todos los empleados de las sucursales de la ciudad de `Madrid`.

```
DELETE
FROM Empleados
WHERE nombreSuc IN (SELECT nombreSuc
                    FROM Sucursales
                    WHERE ciudadSuc = "Madrid");
```

La eliminación de registros en una tabla puede desencadenar la eliminación de registros en otras tablas si se han definido eliminaciones en cascada. Por último, si no especifica la cláusula `WHERE` se eliminarán todos los registros de la tabla.

4.2.3.3. Actualización de la información de la base de datos

Los registros de una tabla se pueden modificar mediante la orden `UPDATE` de SQL. Esta orden actualiza el valor de las columnas de los registros que cumplan una condición. Al igual que en la orden `DELETE`, la condición se especifica en una cláusula `WHERE`. La sintaxis es la siguiente

```
UPDATE tabla
SET Modificacion
WHERE Condicion
```

La consulta siguiente aumenta en `500` el saldo de las cuentas de la sucursal de `Castellana`.

```
UPDATE Cuentas
SET saldo = saldo + 500
WHERE nombreSuc = "Castellana";
```

En la cláusula `WHERE` se pueden especificar subconsultas como hicimos con la instrucción `DELETE`. Por ejemplo, para incrementar en un 10 por ciento el saldo de las cuentas de sucursales de la ciudad de `Almería` escribiríamos

```
UPDATE Cuentas
SET saldo = saldo * 1.1
WHERE nombreSuc IN (SELECT nombreSuc
                    FROM Sucursales
                    WHERE ciudadSuc = "Almería");
```

4.3. QBE

QBE (Query by Example) es un lenguaje para bases de datos relacionales basado en el cálculo relacional de dominios, y es uno de los primeros lenguajes de consulta

orientados a la pantalla o gráficos, con una sintaxis mínima. Fue desarrollado por IBM como lenguaje de consulta para DB2, y actualmente está incorporado con una interfaz de señalar y seleccionar en MS Access. La diferencia fundamental con SQL radica en que el usuario no tiene que especificar una consulta estructurada, sino que rellena *plantillas* de relaciones en la pantalla. El usuario no tiene que recordar los nombres de las tablas y sus atributos, porque se muestran en las plantillas, y lo que hace es introducir constantes y variables en dichas plantillas para construir un *ejemplo* relacionado con la solicitud de obtención de datos o actualización.

A continuación se muestra la plantilla correspondiente a la tabla *clientes*:

CLIENTES	NOMBRECLI	DNICLI	DOMICILIO

4.3.1. Recuperaciones básicas en QBE

Para especificar una consulta de recuperación de datos, hay que indicar los criterios de selección introduciendo *valores de ejemplo* bajo los atributos correspondientes, y hay que introducir el operador P. en los atributos que deseamos visualizar. Si deseamos visualizar todos los atributos de una tabla, ponemos el operador P. bajo el nombre de la tabla. Primero se eligen las tablas necesarias para realizar la consulta, y a continuación se rellenan las plantillas y se ejecuta la consulta.

La siguiente consulta muestra el dni y domicilio del cliente Aranda:

CLIENTES	NOMBRECLI	DNICLI	DOMICILIO
	Aranda	P.	P.

Si queremos ver todos los datos de todos los clientes, introduciremos esta consulta:

CLIENTES	NOMBRECLI	DNICLI	DOMICILIO
P.			

También podemos usar operadores como >, <, >= o <=. Por ejemplo, la siguiente consulta muestra los números de cuenta y saldos de cuentas con un saldo superior a 100 €:

CUENTAS	NUMEROCTA	SALDO	NOMBRESUC
	P.	P.>100	

Para introducir condiciones unidas con AND, introducimos los valores correspondientes en la misma fila de la plantilla. Para unirlas con OR, los introducimos en filas diferentes.

La siguiente consulta muestra todos los datos de las cuentas de la sucursal Castellana con saldo superior a 150 €:

CUENTAS	NUMEROCTA	SALDO	NOMBRESUC
P.		>150	Castellana

La siguiente consulta muestra los nombres de los empleados que trabajan en las sucursales Castellana y Sol:

EMPLEADOS	NOMBREEMP	DNIEMP	NOMBRESUC
	P.		Castellana
	P.		Sol

Para combinar tablas, necesitaremos usar *variables*. Una variable QBE va precedida de un subrayado, como _X. Usaremos la misma variable en los atributos de las tablas que deben tener el mismo valor al combinarlas.

La siguiente consulta muestra los nombres y dni de empleados que trabajan en sucursales de Almería:

EMPLEADOS	NOMBREEMP	DNIEMP	NOMBRESUC
	P.	P.	_X

SUCURSALES	NOMBRESUC	CIUDADSUC	ACTIVO
	_X	Almería	

Si queremos obtener atributos de tablas distintas, podemos poner una variable en cada uno de ellos, y crear una tabla resultado con todas las variables de salida, porque si no, el sistema mostrará los valores en tablas distintas, dificultando la lectura.

La siguiente consulta muestra los dni de cliente, números de cuenta y saldos de las cuentas de la sucursal Paseo:

CUENTAS	NUMEROCTA	SALDO	NOMBRESUC
	_C	_S	Paseo

CTACLI	NUMEROCTA	DNICLI
	_C	_D

RESULTADO			
P.	_D	_C	_S

Para especificar una secuencia de ordenación de los resultados, se usan los operadores AO. (ascendente) y DO. (descendente) en las columnas.

Cuando una condición es compleja y no se puede expresar directamente en las plantillas, se puede escribir directamente en un *cuadro de condición*.

4.3.2. Agrupación y agregación

En QBE, el operador de agregación es G., que indica que los valores de la tabla deben agruparse en función de los valores de esa columna. Se dispone de las funciones AVG., SUM., CNT., MAX. y MIN. Al contrario de SQL, estas funciones se aplican a valores *distintos* dentro de un grupo. Para tener en cuenta todos los valores, añadiremos el operador ALL.

La siguiente consulta cuenta el número de ciudades en las que hay sucursales:

SUCURSALES	NOMBRESUC	CIUDADSUC	ACTIVO
		P.CNT.	

Y la siguiente consulta suma el activo de todas las sucursales:

SUCURSALES	NOMBRESUC	CIUDADSUC	ACTIVO
			P.SUM.ALL.

La consulta siguiente muestra el saldo medio de las cuentas en cada sucursal:

CUENTAS	NUMEROCTA	SALDO	NOMBRESUC
		P.AVG.ALL.	P.G.

La siguiente consulta muestra los nombres de clientes con dos o más cuentas:

CLIENTES	NOMBRECLI	DNICLI	DOMICILIO
	P.	_X	

CTACLI	NUMEROCTA	DNICLI
	CNT._Y	G._X

CONDICIONES
CNT._Y>=2

4.3.3. Modificación de la base de datos

QBE tiene tres operadores para modificar la base de datos: I. (insertar), U. (actualizar) y D. (eliminar). Los operadores de insertar y eliminar se indican en la columna del nombre de la tabla, lo que resulta lógico ya que son operadores a nivel de fila. El operador de actualización se indica en la columna o columnas que se van a actualizar.

A continuación se muestra como insertar un empleado de la sucursal Los Pinos que se llamase Mukos y con DNI 17:

EMPLEADOS	NOMBREEMP	DNIEMP	NOMBRESUC
I.	Mukos	17	Los Pinos

Para eliminar, es necesario especificar la condición que cumplen las filas a suprimir. Por ejemplo, podríamos eliminar el empleado que acabamos de insertar de este modo:

EMPLEADOS	NOMBREEMP	DNIEMP	NOMBRESUC
D.		17	

En la actualización, es necesario indicar la condición que cumplen las filas a modificar, y los nuevos valores que se van a dar a las columnas modificadas.

Para incrementar en 100 € el saldo de todas las cuentas de la sucursal Sol, utilizaríamos esta especificación:

CUENTAS	NUMEROCTA	SALDO	NOMBRESUC
		U._S+100	Sol