TEMA 8: REALIZACIÓN DE CONSULTAS AVANZADAS (SUBCONSULTAS Y CONSULTAS MULTITABLAS)

El lenguaje de definición de datos (DML)

OBJETIVOS:

- **Identificar las herramientas y sentencias para realizar consultas.**
- **Realizar consultas simples sobre una tabla**
- **Realizar consultas que generan valores de resumen**
- **Realizar consultas sobre el contenido de varias tablas mediante composiciones internas**
- **Realizar consultas sobre el contenido de varis tablas mediante composiciones externas.**
- **Realizar consultas con subconsultas**
- **Valorar las ventajas e inconvenientes de las distintas opciones válidas para llevar a cabo una consulta determinada**

GESTIÓN DE BASE DE DATOS
IES PUERTO DE LA CRUZ

Filtros de Grupos

Los filtros de grupos deben realizarse mediante el uso de la cláusula HAVING puesto que WHERE actúa antes de agrupar los registros. Es decir, si se desea filtrar resultados calculados mediante agrupaciones se debe usar la siguiente sintaxis:

```
SELECT [DISTINCT] select_expr [,select_expr] ...
[FROM tabla]
[WHERE filtro]
[GROUP BY expr [, expr].... ]
[HAVING filtro_grupos]
[ORDER BY {nombre_columna | expr | posición} [ASC | DESC] , ... ]
```

HAVING aplica los mismos filtros que la cláusula WHERE. A continuación se ilustran algunos ejemplos:

```
#Consulta 1:
#Seleccionar los equipos de la nba cuyos jugadores
#pesen de media más de 228 libras
SELECT Nombre_equipo,avg(peso)
FROM jugadores
GROUP BY Nombre_equipo
HAVING avg(peso)>228 ORDER BY avg(peso);
+---------------+-----------+
| Nombre_equipo | avg(peso) |
+---------------+-----------+
 Suns           228.8462
 Wizards        229.6923
 Lakers         230.0000
 Jazz           230.0714
 Knicks         235.4667
+---------------+-----------+
```

```
#query 2
#seleccionar qué equipos de la nba tienen más de 1 jugador español
      SELECT Nombre_equipo,count(*)
         FROM jugadores
         WHERE procedencia='Spain'
         GROUP BY Nombre_equipo
         HAVING count(*)>l;
         +---------------+----------+
         | Nombre_equipo | count(*) |
         +---------------+----------+
         | Raptors 2 |
         +---------------+----------+
```

Subconsultas

Las subconsultas se utilizan para realizar filtrados con los datos de otra consulta. Estos filtros pueden ser aplicados tanto en la cláusula WHERE para filtrar registros como en la cláusula HAVING para filtrar grupos. Por ejemplo, con la base de datos de la NBA, es posible codificar una consulta para pedir los nombres de los jugadores de la división 'SouthEast':

```
SELECT nombre FROM jugadores
WHERE Nombre_equipo IN
(SELECT Nombre FROM equipos WHERE division='SouthWest');
+--------------------+
|nombre
+--------------------+
  Andre Brown
  Kwame Brown
  Brian Cardinal
  Jason Collins

+--------------------+
```

Se observa que la sub consulta es precisamente la sentencia SELECT encerrada entre paréntesis. De esta forma, se hace uso del operador *in* para tomar los equipos de la división 'SouthEast'. Si se ejecuta la subconsulta por separado se obtiene:

```
SELECT Nombre FROM equipos
WHERE division='SouthWest';
+-----------+
|Nombre
+-----------+
Hornets
Spurs
Rockets
Mavericks
Grizzlies
+-----------+
```

En las siguientes secciones se detallan los posibles operadores que se pueden usar con las subconsultas.

La sub consulta se convierte en algo equivalente a:

```
SELECT nombre FROM jugadores
WHERE Nombre_equipo IN
('Hornets' ,'Spurs' ,'Rockets',
'Mavericks' ,'Grizzlies')
```

En las siguientes secciones se detallan los posibles operadores que se pueden usar con las subconsultas.

Test de Comparación

Consiste en usar los operadores de comparación =, >=, <=, <>, >y <para comparar el valor producido con un valor único generado por una sub consulta. Por ejemplo, para consultar el nombre del jugador de mayor altura de la nba, es posible hacer algo como esto:

```
SELECT nombre FROM jugadores
WHERE altura =
        (SELECT max(altura) FROM jugadores);
+----------+
I nombre
+----------+
|Yao Ming |
+----------+
```

Se puede comprobar que la subconsulta produce un único resultado, utilizándolo para filtrar. Nótese que con este tipo de filtro la subconsulta sólo debe producir un único valor (una fila y una columna), por tanto, si se codifica algo del tipo:

```
SELECT nombre FROM jugadores
WHERE altura = (SELECT max(altura),max(peso) FROM jugadores);
ERROR 1241 (21000): Operand should contain 1 column(s)
```

También fallaría que la subconsulta devolviera más de una fila:

```
SELECT nombre FROM jugadores
WHERE altura = (SELECT max(altura)
FROM jugadores GROUP BY Nombre_Equipo);
ERROR 1242 (21000): Subquery returns more than 1 row
```

Una restricción importante es que la subconsulta debe estar siempre al lado derecho del operador de comparación. Es decir:

```
Campo <= subconsulta
```

siendo inválida la expresión:

```
subconsulta >= Campo
```

Test de pertenencia a conjunto

Este test consiste en una variante del usado para consultas simples, y es el que se ha utilizado para ilustrar el primer ejemplo de la sección. Consiste en usar el operador IN para filtrar los registros cuya expresión coincida con algún valor producido por la sub consulta.

Por ejemplo, para extraer las divisiones de la nba donde juegan jugadores españoles:

```
SELECT division FROM equipos WHERE nombre IN
(SELECT Nombre_equipo FROM jugadores WHERE procedencia='Spain');
+-----------+
|division
+-----------+
  Atlantic
  NorthWest
  Pacific
  SouthWest
+-----------+
```

Test de existencia

El test de existencia permite filtrar los resultados de una consulta si existen filas en la subconsulta asociada, esto es, si la subconsulta genera un número de filas distinto de 0.
Para usar el test de existencia se utiliza el operador EXISTS:

```
SELECT columnas FROM tabla
WHERE EXISTS (subconsulta)
```

El operador EXISTS también puede ser precedido de la negación (NOT) para filtrar si no existen resultados en la subconsulta:

```
SELECT columnas FROM tabla
WHERE NOT EXISTS (subconsulta)
```

Para seleccionar los equipos que no tengan jugadores españoles se podría usar la siguiente consulta:

```
SELECT Nombre FROM equipos WHERE NOT EXISTS
(SELECT Nombre FROM jugadores
WHERE equipos.Nombre = jugadores.Nombre_Equipo
AND procedencia='Spain');
+-------------+
| Nombre
+-------------+
76ers
Bobcats
Bucks

+-------------+
```

Para comprender la lógica de esta consulta, se puede asumir que cada registro devuelto por la consulta principal provoca la ejecución de la subconsulta, así, si la consulta principal (SELECT nombre FROM Equipos) devuelve 30 registros, se entenderá que se ejecutan 30 subconsultas, una por cada nombre de equipo que retorne la consulta principal. Esto en realidad no es así, puesto que el SGBD optimiza la consulta para hacer tan sólo dos consultas y una operación *join* que se estudiará más adelante, pero sirve de ejemplo ilustrativo del funcionamiento de esta consulta:

```
SELECT Nombre from equipos;
+--------------+
|Nombre
+--------------+
76ers            -> subconsulta ejecutada #1
Bobcats          -> subconsulta ejecutada #2

 ….                    …….
Raptors       -> subconsulta ejecutada #22

……..          -> ……..

+--------------+
```

Cada subconsulta ejecutada sería como sigue:

```
#subconsulta ejecutada #1
SELECT Nombre FROM jugadores
WHERE '76ers' = jugadores.Nombre_Equipo
AND procedencia='Spain';
```

Esta subconsulta no retorna resultados, por tanto, el equipo '76ers' es seleccionado para ser devuelto en la consulta principal, puesto que no existen (NOT EXISTS) jugadores españoles.

Sin embargo para el registro 22 de la consulta principal, aquel cuyo nombre es 'Raptors', la consulta:

```
#subconsulta ejecutada #22
SELECT Nombre FROM jugadores
WHERE 'Raptors' = jugadores.Nombre_Equipo
AND procedencia='Spain';

+----------------+
|Nombre
+----------------+
|Jose Calderon |
|Jorge Garbajosa |
+----------------+
```

devuelve 2 jugadores, por tanto, existen (EXISTS) registros de la subconsulta y por tanto el equipo 'Raptors' NO es seleccionado por la consulta principal.

En conclusión, se puede decir que la consulta principal *enlaza* los registros con los devueltos por las subconsultas.

Test cuantificados ALL y ANY

Los test cuantificados sirven para calcular la relación entre una expresión y todos los registros de la subconsulta (ALL) o algunos de los registros de la subconsulta (ANY).

De esta manera se podría saber los jugadores de la nba que pesan más que todos los jugadores españoles:

```
SELECT nombre,peso from jugadores
WHERE peso > ALL
 (SELECT peso FROM jugadores WHERE procedencia='Spain');
+---------------------+------+
¦nombre ¦peso ¦
+---------------------+------+
Michael Doleac           262
Al Jefferson             265
Chris Richard            270

  +---------------------+------+
```

Al igual que en el caso del operador exists se puede asumir que por cada registro de la consulta principal se ejecuta una subconsulta. En tal caso, para el jugador 'Michael Doleac' cuyo peso es 262 libras, se comprobaría si 262 es mayor que los pesos de todos los jugadores españoles, que son devueltos por la subconsulta (SELECT peso FROM jugadores WHERE procedencia='Spain').

También se podría consultar los bases (en inglés Guard 'G') *posicion='G'*, que pesan más que cualquier (ANY) pivot (en inglés Center 'C') *posicion='C'* de la nba:

```
SELECT nombre,peso from jugadores
WHERE posicion='G' AND
 peso > ANY
 (SELECT peso FROM jugadores WHERE posicion='C');
+-------------+------+
¦nombre       ¦peso ¦
+-------------+------+
¦Joe Johnson  ¦235 ¦
+-------------+------+
```

Se comprueba que en el *caso* de 'Joe Johnson', su peso (235 libras), es mayor que algún peso de algún pivot de la nba, dándose así el peculiar caso de un base más pesado que algún pivot.

Subconsultas anidadas

Se puede usar una sub consulta para filtrar los resultados de otra sub consulta.
De esta manera se *anidan* sub consultas. Por ejemplo, si se desea obtener el nombre de la ciudad donde juega el jugador más alto de la nba, habría que pensar cómo hacerlo de forma estructurada:

1. Obtener la altura del jugador más alto:

```
X <- (SELECT max(altura) from jugadores)
```

2. Obtener el nombre del jugador, a través de la altura se localiza al jugador por tanto, su equipo:

```
Y<- SELECT Nombre_equipo from jugadores WHERE Altura = X
```

3. Obtener la ciudad:

```
SELECT ciudad FROM equipos WHERE nombre= Y
```

Ordenando todo esto, se puede construir la consulta de abajo a arriba:

```
   SELECT ciudad FROM equipos WHERE nombre =
(SELECT Nombre_equipo FROM jugadores WHERE altura =
            (SELECT MAX(altura) FROM jugadores));
   +---------+
   |ciudad
   +---------+
   |Houston |
   +---------+
```

Esta manera de generar consultas es muy sencilla, y a la vez permite explorar la información de la base de datos de forma *estructurado:* En opinión de muchos autores esta forma estructurada de generar consultas es la que dio a SQL su 'S' de *Structured.*

Consultas multitabla

Una consulta multitabla es aquella en la que se puede consultar información de más de una tabla. Se aprovechan los campos relacionados de las tablas para *unirlas* (join). Para poder realizar este tipo de consultas hay que utilizar la siguiente sintaxis :

```
SELECT [DISTINCT] select_expr [,select_expr] ...
[FROM referencias_tablas]
[WHERE filtro]
[GROUP BY expr [, expr] ....
[HAVING filtro_grupos]
[ORDER BY {nombre_columnas |expr |posición} [ASC |DESC] , ... ]
```

La diferencia con las consultas sencillas se halla en la cláusula FROM. Esta vez en lugar de una tabla se puede desarrollar el token referencias.tablas:

```
referencias_tablas:
referencia_tabla [ , referencia_tabla] ...
|referencia_tabla [INNER |CROSS] JOIN referencia_tabla [ON condición]
|referencia_tabla LEFT [OUTER] JOIN referencia_tabla ON condición
|referencia_tabla RIGHT [OUTER] JOIN referencia_tabla ON condición
referencia_tabla:
nombre_tabla [[AS] alias]
```

La primera opción, (referencia.tabla1[, referencia.tabla] ...) es típica de SQL 1 (SQL-86) para las uniones, que consisten en un producto cartesiano más un filtro por las columnas relacionadas, y el resto de opciones son propias de SQL 2 (SQL-92 y SQL-2003).

❖ Consultas multitabla SQL 1

El producto cartesiano de dos tablas son todas las combinaciones de las filas de una tabla unidas a las filas de la otra tabla. Por ejemplo, una base de datos de mascotas con dos tablas animales y propietarios:

```
SELECT * FROM propietarios;
+-----------+--------------------+
|dni  |nombre
+-----------+--------------------+
|51993482Y   |José Pérez
|2883477X    |Matías Fernández
|372763172   |Francisco Martínez
+-----------+--------------------+

SELECT * FROM animales;
+--------+----------+-------+-------------+
|codigo |nombre |tipo |propietario |
+--------+----------+-------+-------------+
1 |Cloncho |gato |51993482Y
2 |Yo da |gato |51993482Y
3 |Sprocket |perro |372763172
+--------+----------+-------+-------------+
```

Un producto cartesiano de las dos tablas se realiza con la siguiente sentencia:

```
SELECT * FROM animales ,propietarios;

+--------+----------++--------+----------++--------+----------++--------+----------+-------+-------------+-----------+------------------+
| codigo | nombre   | tipo   | propietario | dni        | nombre                  |
+--------+----------++--------+----------++--------+----------++--------+----------+-------+-------------+-----------+------------------+
|      1 | Cloncho  | gato   | 51993482Y   | 51993482Y  | José      Pérez         |
|      2 | Yo da    | gato   | 51993482Y   | 51993482Y  | José      Pérez         |
|      3 | Sprocket | perro  | 37276317Z   | 51993482Y  | José      Pérez         |
|      1 | Cloncho  | gato   | 51993482Y   | 2883477X   | Matías    Fernández     |
|      2 | Yo da    | gato   | 51993482Y   | 2883477X   | Matías    Fernández     |
|      3 | Sprocket | perro  | 37276317Z   | 2883477X   | Matías    Fernández     |
|      1 | Cloncho  | gato   | 51993482Y   | 37276317Z  | Francisco Martínez      |
|      2 | Yoda     | gato   | 51993482Y   | 37276317Z  | Francisco Martínez      |
|      3 | Sprocket | perro  | 37276317Z   | 37276317Z  | Francisco Martínez      |

+--------+----------++--------+----------++--------+----------++--------+----------+-------+-------------+-----------+------------------+
```

La operación genera un conjunto de resultados con todas las combinaciones posibles entre las filas de las dos tablas, y con todas las columnas. Aparentemente esto no tiene mucha utilidad, sin embargo, si se aplica un filtro al producto cartesiano, es decir, una condición WHERE que escoja sólo aquellas filas en las que el campo dni (del propietario) coincida con el propietario (de la mascota), se obtienen los siguientes interesantes resultados:

```
SELECT * FROM animales,propietarios
        WHERE propietarios.dni=animales.propietario;
+--------+----------+-------+-------------+-----------+--------------------+
| códig  | nombre   | tipo  | propietario | dni       | nombre
+--------+----------+-------+-------------+-----------+--------------------+
      1   | Cloncho | gato   | 51993482Y | 51993482Y | José Pérez
      2   | Yoda    | gato   | 51993482Y | 51993482Y | José Pérez
      3   | Sprocket | perro | 37276317Z | 37276317Z | Francisco Martínez
+--------+----------+-------+-------------+-----------+--------------------+
```

Mediante esta consulta se ha obtenido información relacionada entre las dso tablas. Se aprecia como los dos gatos (Cloncho y Yoda) aparecen con su dueño (José Pérez), y que, Sprocket el perro, aparece con su dueño Francisco Martínez Esta operación se llama *JOIN* de las tablas Propietarios y Animales, y consiste en realizar un producto cartesiano de ambas y un filtro por el campo relacionado (Clave Foránea *vs* Clave Primaria).
Por tanto, *JOIN = PRODUCTO CARTESIANO + FILTRO*. En el apartado siguiente se estudiará que existen varios tipos de join y que SQL 2 incluye en su sintaxis formas de parametrizar estos tipos de join.

Este mismo procedimiento se puede aplicar con N tablas, por ejemplo, con la base de datos *jardinería",* cuyo gráfico de relaciones es el siguiente:

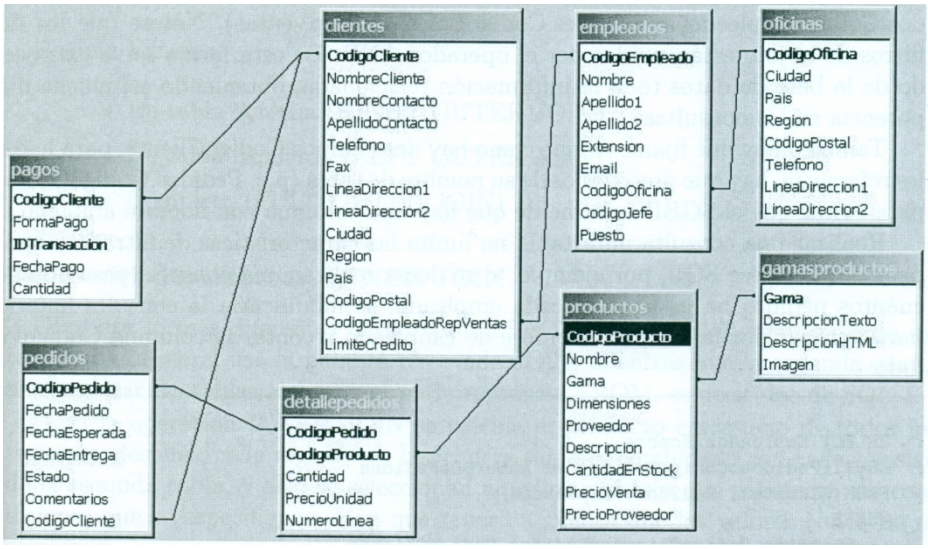

Figura 4.1: Relaciones de la bbdd "jardinería

se puede generar una consulta para obtener un listado de pedidos gestionados por cada empleado:

```
mysql> SELECT Empleados.Nombre,Clientes.NombreCliente,Pedidos.CodigoPedido
  FROM Clientes, Pedidos, Empleados
WHERE Clientes.CodigoCliente=Pedidos.CodigoCliente
            AND
              Empleados.CodigoEmpleado = Clientes.CodigoEmpleadoRepVentas
  ORDER BY Empleados.Nombre;
+----------------+--------------------------------+--------------+
| Nombre | NombreCliente                 | CodigoPedido |
+----------------+--------------------------------+--------------+
| Emmanuel Naturagua                                 18
| Emmanuel Beragua                                   16
……..
| Walter Santiago DGPRODUCTIONS GARDEN   65
| Walter Santiago Gardening & Associates             58
+----------------+--------------------------------+--------------+
```

Se observa que en este caso hay dos JOIN, el primer join entre la tabla Clientes y Pedidos, con la condición Clientes.CodigoCliente=Pedidos.CodigoCliente y la segunda join entre el resultado de la primera join y la tabla Empleados (Empleados.CodigoEmpleado = Clientes.CodigoEmpleadoRepVentas). Nótese que los dos filtros de la join están unidas por el operador AND. De esta forma se va extrayendo de la base de datos toda la información relacionada, obteniendo así mucha más potencia en las consultas.

También hay que fijarse en que como hay dos campos CodigoCliente, para hacerles referencia, hay que precederlos de su nombre de tabla (p.e. Pedidos.CodigoCliente) para evitar que el SGBD informe de que hay una columna con nombre ambiguo.

Realizar una consulta multitabla no limita las características de filtrado y agrupación que ofrece SQL, por ejemplo, si se desea realizar una consulta para obtener cuántos pedidos ha gestionado cada empleado, se modificaría la consulta anterior para agrupar por la columna Nombre de Empleado y contar la columna CodigoPedido:

```
SELECT Empleados.Nombre,
COUNT(Pedidos.CodigoPedido) as NumeroOePedidos
FROM Clientes, Pedidos, Empleados
WHERE
    Clientes.CodigoCliente=Pedidos.CodigoCliente
    ANO
```

```
    Empleados.CodigoEmpleado = Clientes.CodigoEmpleadoRepVentas
GROUP BY Empleados.Nombre
ORDER BY NumeroOePedidos;

+----------------+----------------+
| Nombre         | NumeroOePedidos |
+----------------+----------------+
  Michael                        5
  Lorena                        10

  Walter Santiago    20
 +----------------+----------------+
```

❖ Consultas multitabla SQL 2

SQL 2 introduce otra sintaxis para los siguientes tipos de consultas multitablas: las joins (o composiciones) internas, externas y productos cartesianos (también llamadas composiciones cruzadas):

1. Join Interna:

   • De equivalencia (INNER JOIN)
   • Natural (NATURAL JOIN)

2. Producto Cartesiano (CROSS JOIN)

3. Join Externa

   - De tabla derecha (R1GHT OUTER JOIN)
   - De tabla izquierda (LEFT OUTER JOIN)
   - Completa (FULL OUTER JOIN)

Composiciones internas. INNER JOIN

Hay dos formas diferentes para expresar las INNER JOIN o composiciones internas. La primera, usa la palabra reservada JOIN, mientras que la segunda usa ',' para separar las tablas a combinar en la sentencia FROM, es decir, las de SQL 1.

Con la operación INNER JOIN se calcula el producto cartesiano de todos los registros, después, cada registro en la primera tabla es combinado con cada registro de la segunda tabla, y sólo se seleccionan aquellos registros que satisfacen las condiciones que se especifiquen. Hay que tener en cuenta que los valores nulos no se combinan.

Como ejemplo, la siguiente consulta toma todos los registros de la tabla animales y encuentra todas las combinaciones en la tabla propietarios. La JOIN compara los valores de las columnas dni y propietario. Cuando no existe esta correspondencia entre algunas combinaciones, éstas no se muestran; es decir, que si el dni de un propietario de una mascota no coincide con algún dni de la tabla de propietarios, no se mostrará el animal con su respectivo propietario en los resultados.

```
SELECT * FROM animales INNER JOIN propietarios
   ON  animales.propietario = propietarios.dni;
+--------+----------+-------+-------------+-----------+--------------------+
|codigo |nombre |tipo |propietario |dni |nombre
+--------+----------+-------+-------------+-----------+--------------------+
1 |Cloncho |gato |51993482Y |51993482Y |José Pérez
2 |Yo da |gato |51993482Y |51993482Y |José Pérez
3 | Sprocket |perro |37276317Z |37276317Z |Francisco Martínez
+--------+----------+-------+-------------+-----------+--------------------+
```

```
# Nótese que es una consulta equivalente a la vista en el apartado anterior
# select * from animales,propietarios
 # where animales.propietario=propietarios.dni;
```

Además, debe tenerse en cuenta que si hay un animal sin propietario no saldrá en el conjunto de resultados puesto que no tiene coincidencia en el filtro:

```
INSERT INTO animales VALUES (null, 'Arco', 'perro',null);
SELECT * FROM animales;
+--------+----------+-------+-------------+
|código  |nombre    |tipo |propietario |
+--------+----------+-------+-------------+
1          Cloncho  gato      51993482Y
2          Yo da    gato      51993482Y
3          Sprocket  perro     37276317Z
4          Arco      perro     NULL                          #nueva mascota sin propietario
+--------+----------+-------+-------------+

SELECT * FROM animales INNER JOIN propietarios
     ON animales.propietario = propietarios.dni;
+--------+----------+-------+-------------+-----------+--------------------+
|codigo |nombre |tipo      |propietario        |dni                |nombre
+--------+----------+-------+-------------+-----------+--------------------+
  1    |Cloncho |gato      |51993482Y        |51993482Y        |José Pérez
  2    |Yo da    |gato      |51993482Y        |51993482Y        |José Pérez
  3    |Sprocket |perro     |37276317Z        |37276317Z        |Francisco Martínez
+--------+----------+-------+-------------+-----------+--------------------+
#LA NUEVA MASCOTA NO APARECE!
```

Puede hacerse variantes de la inner join cambiando el operador del filtro, por ejemplo:

```
SELECT * FROM animales INNER JOIN propietarios
     ON propietarios.dni >= animales.propietario;
+--------+----------+-------+-------------+-----------+--------------------+
|codigo |nombre      |tipo     |propietario      |dni                |nombre
+--------+----------+-------+-------------+-----------+--------------------+
1          Cloncho  gato      51993482Y      51993482Y       José Pérez
2          Yoda     gato      51993482Y      51993482Y       José Pérez
3          Sprocket  perro     37276317Z     51993482Y       José Pérez
3          Sprocket  perro     37276317Z     372763172       Francisco Martínez
+--------+----------+-------+-------------+-----------+--------------------+
```

Composiciones naturales. NATURAL **JOIN**

Es una especialización de la INNER JOIN. En este caso se comparan todas las columnas que tengan el mismo nombre en ambas tablas. La tabla resultante contiene sólo una columna por cada par de columnas con el mismo nombre.

```
DESCRIBE Empleados;
+---------------+-------------+------+-----+---------+-------+
| Field         | Type        | Null | Key | Default | Extra |
+---------------+-------------+------+-----+---------+-------+
| CodigoEmpleado | int (11)    | NO   | PRI | NULL    |       |
| Nombre         | varchar(50) | NO   |     | NULL    |       |
| Apellid01      | varchar(50) | NO   |     | NULL    |       |
| Apellid02      | varchar(50) | YES  |     | NULL    |       |
| Extension      | var char (10) | NO |     | NULL    |       |
| Email          | varchar(100) | NO  |     | NULL    |       |
| CodigoOficina  | var char (10) | NO |     | NULL    |       | #relación
| CodigoJefe     | int (11)    | YES  |     | NULL    |       |
| Puesto         | varchar(50) | YES  |     | NULL    |       |
+---------------+-------------+------+-----+---------+-------+
DESCRIBE Oficinas;
+----------------+------------+------+-----+---------+-------+
| Field          | Type       | Null | Key | Default | Extra |
+----------------+------------+------+-----+---------+-------+
| CodigoOficina  | var char (10) | NO | PRI | NULL    |       | #relación
| Ciudad         | varchar (30) | NO |     | NULL    |       |
| Pais           | varchar (50) | NO |     | NULL    |       |
| Region         | varchar(50) | YES |     | NULL    |       |
| CodigoPostal   | var char (10) | NO |     | NULL    |       |
| Telefono       | varchar(20) | NO  |     | NULL    |       |
| LineaDireccion1 | varchar(50) | NO |     | NULL    |       |
| LineaDireccion2 | varchar(50) | YES |     | NULL    |       |
+----------------+------------+------+-----+---------+-------+

#NATURAL JOIN coge los mismos nombres de campo, en este caso CodigoOficina

SELECT CodigoEmpleado,Empleados.Nombre,
      Oficinas.CodigoOficina,Oficinas.Ciudad
          FROM Empleados NATURAL JOIN Oficinas;
+---------------+---------------+--------------+--------------------+
| CodigoEmpleado | Nombre       | CodigoOficina | Ciudad             |
+---------------+---------------+--------------+--------------------+
|       1       | Marcos        | TAL-ES       | Talavera de la Reina |
|       2       | Ruben         | TAL-ES       | Talavera de la Reina |
|      31       | Mariko        | SYD-AU       | Sydney             |
+---------------+---------------+--------------+--------------------+
```

Hay que fijarse en que, aunque CodigoEmpleado es un campo que está en dos tablas, esta vez no es necesario precederlo del nombre de tabla puesto que NATURAL JOIN devuelve un único campo por cada pareja de campos con el mismo nombre.

Producto cartesiano.CROSS JOIN

Este tipo de sintaxis devuelve el producto cartesiano de dos tablas:

```
#equivalente a SELECT * FROM animales ,propietarios;
SELECT * FROM animales CROSS JOIN propietarios;
+--------+----------+-------+------------+-----------+-------------------t
|codigo |nombre   |tipo  |propietario   |    dni    |nombre
+--------+----------+-------+------------+-----------+-------------------t
       1 Cloncho     gato    51993482Y     51993482Y  José Pérez
       1 Cloncho     gato    51993482Y    2883477X    Matías Fernández
       1 Cloncho     gato    51993482Y     37276317Z  Francisco Martínez
       2 Yoda        gato    51993482Y     51993482Y  José Pérez
       2 Yoda        gato    51993482Y    2883477X    Matías Fernández
       2 Yoda        gato    51993482Y     37276317Z  Francisco Martínez
       3 Sprocket     perro 37276317Z     51993482Y  José Pérez
       3 Sprocket     perro 37276317Z    2883477X    Matías Fernández
       3 Sprocket     perro 37276317Z     37276317Z  Francisco Martínez
         4 Arco      perro    NULL        51993482Y  José Pérez
         4 Arco      perro    NULL       2883477X    Matías Fernández
         4 Arco      perro    NULL        37276317Z  Francisco Martínez
+--------+----------+-------+------------+-----------+-------------------t
                                                                          |
```

Nótese que aparece también el nuevo animal insertado sin propietario (Arco).

Composiciones externas.OUTER JOIN

En este tipo de operación, las tablas relacionadas no requieren que haya una equivalencia. El registro es seleccionado para ser mostrado aunque no haya otro registro que le corresponda.

OUTER JOIN se subdivide dependiendo de la tabla a la cual se le admitirán los registros que no tienen correspondencia, ya sean de tabla izquierda, de tabla derecha, o combinación completa.

Si los registros que admiten no tener correspondencia son los que aparecen en la tabla de la izquierda se llama composición de tabla izquierda o LEFT JOIN (o LEFT OUTER JOIN):

```
#ejemplo de LEFT OUTER JOIN
#animales LEFT OUTER JOIN propietarios
#animales está a la izquierda
#propietarios está a la derecha
```

```
SELECT * FROM animales LEFT OUTER JOIN propietarios
  ON animales.propietario = propietarios.dni¡

+--------+----------+-------+------------+-----------+-------------------+
¦codigo ¦nombre ¦tipo ¦propietario ¦dni                ¦nombre
+--------+----------+-------+------------+-----------+-------------------+
       1 Cloncho      gato    51993482Y     51993482Y José Pérez
       2 Yoda         gato    51993482Y     51993482Y José Pérez
       3 Sprocket perro 37276317Z          37276317Z Francisco Martínez
       4 Arco         perro NULL           NULL        NULL
+--------+----------+-------+------------+-----------+-------------------+
```

Se observa que se incluye el perro Arco que no tiene propietario, por tanto, sus campos relacionados aparecen con valor NULL. El sentido de esta query podría ser, sacar todos los animales y si tienen relación, sacar sus propietarios, y si no tiene propietario, indicarlo con un valor NULO o con VACÍO.

¿Sabías que ... ? Oracle implementaba las consultas externas antes de la aparición de las OUTER JOIN, utilizando el operador (+)= en lugar del operador = en la cláusula WHERE. Esta sintaxis aún está disponible en las nuevas versiones de este SGBD.

Si los registros que admiten no tener correspondencia son los que aparecen en la tabla de la derecha, se llama composición de tabla derecha RIGHT JOIN (o RIGHT OUTER JOIN):

```
#ejemplo de RIGHT OUTER JOIN
#animales RIGHT OUTER JOIN propietarios
#animales está a la izquierda
#propietarios está a la derecha
SELECT * FROM animales RIGHT OUTER JOIN propietarios
     ON animales.propietario = propietarios.dni¡
+--------+----------+-------+------------+-----------+-------------------+
¦codigo ¦nombre ¦tipo ¦propietario ¦dni                ¦nombre
+--------+----------+-------+------------+-----------+-------------------+
       1 Cloncho      gato    51993482Y     51993482Y José Pérez
       2 Yo da        gato    51993482Y     51993482Y José Pérez
     NULL NULL        NULL    NULL          2883477X    Matías Fernández
       3 Sprocket perro 37276317Z          37276317Z Francisco Martínez
+--------+----------+-------+------------+-----------+-------------------+
```

En este caso, los que aparecen son todos los propietarios, incluido Matías Fernández que no tiene una mascota. Se ve que el perro Arco no aparece, pues esta vez los registros que se desean mostrar son todos los de la tabla derecha (es decir, propietarios).

La operación que admite registros sin correspondencia tanto para la tabla izquierda como para la derecha, por ejemplo, animales sin propietario y propietarios sin animales, se llama composición externa completa o FULL JOIN (FULL OUTER JOIN). Esta operación presenta los resultados de tabla izquierda y tabla derecha aunque no tengan correspondencia en la otra tabla. La tabla combinada contendrá, entonces, todos los registros de ambas tablas y presentará valores nulos para registros sin pareja.

```
#ejemplo de FULL OUTER JOIN
#animales FULL OUTER JOIN propietarios
#animales está a la izquierda
#propietarios está a la derecha
 SELECT * FROM animales FULL OUTER JOIN propietarios
           ON animales.propietario = propietarios.dni;
+--------+----------+-------+------------+-----------+--------------------+
| codigo | nombre   | tipo  | propietario | dni       | nombre             |
+--------+----------+-------+------------+-----------+--------------------+
|      1 | Cloncho  | gato  | 51993482Y   | 51993482Y | José Pérez         |
|      2 | Yoda     | gato  | 51993482Y   | 51993482Y | José Pérez         |
|      3 | Sprocket | perro | 37276317Z   | 37276317Z | Francisco Martínez |
|      4 | Arco     | perro | NULL        | NULL      | NULL               |
|   NULL | NULL     | NULL  | NULL        | 2883477X  | Matías Fernández   |
+--------+----------+-------+------------+-----------+--------------------+
```

¿Sabías que ... ? En SQL existe el operador UNION, que añade al conjunto de resultados producidos por una SELECT, los resultados de otra SELECT.
La sintaxis es:

SELECT .... FROM
UNION [ALL]
SELECT .... FROM

El parámetro ALL incluye todos los registros de las dos SELECT, incluyendo los que son iguales. Si no se indica ALL, se excluyen los duplicados.

Aunque MySQL no implementa la característica FULL OUTER JOIN, sí que se puede simular haciendo una unión de los resultados de un LEFT OUTER JOIN y los resultados de un RIGHT OUTER JOIN ,puesto que UNION, sin la opción ALL, elimina los registros duplicados, por tanto, se podría codificar la FULL OUTER JOIN anterior de la siguiente forma:

```
mysql> SELECT * FROM animales LEFT OUTER JOIN propietarios
    ->     ON animales.propietario = propietarios.dni
    -> UNION
    -> SELECT * FROM animales RIGHT OUTER JOIN propietarios
    ->     ON animales.propietario = propietarios.dni;

+--------+----------+-------+------------+--------------+--------------------+
|codigo  |nombre    | tipo  | propietario |dni           |nombre              |
+--------+----------+-------+------------+--------------+--------------------+
|      1 | Cloncho  | gato  | 51993482Y   | 51993482Y    | José Pérez         |
|      2 | Yo da    | gato  | 51993482Y   | 51993482Y    | José Pérez         |
|      3 | Sprocket | perro | 372763172   | 37276317Z    | Francisco Martínez |
|      4 | Arco     | perro | NULL        | NULL         | NULL               |
|   NULL | NULL     | NULL  | NULL        | 2883477X     | Matías Fernández   |
+--------+----------+-------+------------+--------------+--------------------+
```

Consultas reflexivas

A veces, es necesario obtener información de relaciones reflexivas, por ejemplo, un informe de empleados donde junto a su nombre y apellidos apareciera el nombre y apellidos de su jefe. Para ello, es necesario hacer una JOIN entre registros de la misma tabla:

```
mysql> desc Empleados;
+---------------+-------------+------+-----+---------+-------+
|Field |Type |Null |Key |Default |Extra |
+---------------+-------------+------+-----+---------+-------+
  CodigoEmpleado     .í.nt Cí.,          NO      PRI    NULL
  Nombre             varchar(50)     NO             NULL
  Apellido1          varchar(50)     NO             NULL
  Apellido2          varchar(50)     YES            NULL
  Extension          varchar(10)     NO             NULL
  Email              varchar (100)   NO             NULL
  CodigoOficina      varchar(10)     NO             NULL
  CodigoJefe         Lrrt Cl ,,          YES            NULL                 #autorelaci6n
  Puesto             varchar(50)     YES            NULL
+---------------+-------------+------+-----+---------+-------+
```

```
SELECT concat(emp.Nombre,' ',emp.Apellido1) as Empleado,
           concat(jefe.Nombre, , ',jefe.Apellido1) as jefe

            FROM Empleados emp INNER JOIN Empleados jefe ON emp.CodigoEmpleado=jefe.CodigoJefe;
+------------------+-------------------------------------+
| Empleado                      | jefe
+------------------+-------------------------------------+
| Marcos Magaña                 Ruben López
| Ruben L6pez                   Alberto Soria

| Alberto Soria                 Kevin Fallmer
| Kevin Fallmer                 Julian Bellinelli
| Kevin Fallmer                 Mariko Kishi
+------------------+-------------------------------------+
```

Analizando la query anterior, primero se observa el uso de la tabla empleados dos veces, una con un alias emp que representa los empleados como subordinados y otra con alias jefe que representa los empleados como jefes. Ambas tablas (aunque en realidad son la misma) se unen en una JOIN a través de la relación CodigoEmpleado y CodigoJefe.

Por otro lado, el primer campo que se selecciona es la concatenación del nombre y apellido del empleado (concatfemp.Nombre,' ',emp.Apellidol) al que a su vez le damos un alias (empleado) y el segundo campo que es la concatenación de los empleados jefes, al que le se le da el alias jefe.

Se puede observar que en esta query no aparecen los empleados sin jefe, puesto que se ha utilizado un INNER JOIN. Para mostrarlos, habría que usar un LEFT o RIGHT OUTER JOIN.

Consultas con tablas derivadas

```
SELECT * FROM
        (SELECT CodigoEmpleado, Nombre FROM Empleados
        WHERE CodigoOficina='TAL-ES') as tabla_derivada;
```

En este caso se ha de distinguir, por un lado la tabla derivada, (SELECT CodigoEmpleado, Nombre FROM Empleados) que tiene un alias tabla.derivada, es decir, una especie de tabla temporal cuyo contenido es el resultado de ejecutar la consulta, su nombre es tabla.derivada y tiene dos columnas, una CodigoEmpleado y otra Nombre. Este tipo de consultas ayudará a obtener información relacionada de forma mucho más avanzada.

Por ejemplo, en la base de datos *jardinería*, si se desea sacar el importe del pedido de menor coste de todos los pedidos, hay que pensar primero como sacar el total de todos los pedidos y de ahí, el pedido con menor coste con la función de columna MIN:

```
#1: Para calcular el total de cada pedido, hay que codificar esta query
SELECT SUM(Cantidad*PrecioUnidad) as total,CodigoPedido
   FROM DetallePedidos
   GROUP BY CodigoPedido;
+-------+--------------+
|total |CodigoPedido |
+-------+--------------+
   1567 |             1
   7113 |             2
   1085 |             3
      0
    154             117
     51             128
+-------+--------------+

#2: Para calcular el menor pedido, se puede hacer una tabla
# derivada de la consulta anterior y con la función MIN
# obtener el menor de ellos:
SELECT MIN(total) FROM (
SELECT SUM(Cantidad*PrecioUnidad) as total,CodigoPedido
             FROM DetallePedidos
         GROUP BY CodigoPedido
       AS TotalPedidos;
+------------+
|MIN(total) |
+------------+
      4      |
+------------+
#TotalPedidos es la tabla derivada formada
#por el resultado de la consulta entre paréntesis
```

Las tablas derivadas no tienen limitación, es decir, se pueden unir a otras tablas, filtrar, agrupar, etc ..