DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Escuela Técnica Superior de Ingeniería Informática

etsinf

Bases de Datos y Sistemas de Información

Grado en Ingeniería Informática

Unidad Didáctica 2: El lenguaje SQL: manipulación de datos

Parte 1: LMD: Lenguaje de Manipulación de Datos

(Doc. UD2.1)

Curso 2020/2021

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Índice

1 INTRODUCCIÓN A SQL

El lenguaje de consultas estructurado (*Structured Query Language* -SQL) es un lenguaje estándar para los sistemas de gestión de bases de datos (SGBD's) relacionales comerciales.

Permite, entre otras cosas:

- crear y modificar esquemas de bases de datos y
- especificar operaciones sobre bases de datos.

Consta de varios sublenguajes o conjuntos de instrucciones:

- Lenguaje de Definición de Datos (LDD -DDL en inglés-) para crear y modificar esquemas de bases de datos.
- Lenguaje de Manipulación de Datos (LMD -DML en inglés-) para consulta y actualización de las bases de datos. Consta de las instrucciones siguientes:
  - Consulta: consta de una sola instrucción, SELECT (para hacer consultas sobre una o varias tablas).
  - Actualización: consta de tres instrucciones que permiten mantener actualizados los datos de la base de datos.
    - INSERT (para la inserción de tuplas en una sola tabla).
    - DELETE (para el borrado de tuplas en una sola tabla).
    - UPDATE (para la modificación de tuplas en una sola tabla).
  - Instrucciones de control de transacciones.
- Lenguaje de Control. Sólo se citará alguna instrucción.
- Otras instrucciones (no se estudiarán).

Este documento se centrará en el LMD, del que se describirán sólo algunas de entre todas sus posibilidades para adaptar el contenido al alcance de esta asignatura. Concretamente, se estudiará la parte de las instrucciones que se pueden ejecutar desde un intérprete de SQL, lo que se denomina *SQL interactivo*.

La instrucción SELECT es la instrucción que permite la declaración de consultas para la recuperación de información de una o más tablas de una base de datos. Esta instrucción ofrece múltiples opciones y variantes para expresar una misma consulta, que se irán describiendo progresivamente. El SQL es un lenguaje declarativo, esto es, sirve para especificar qué es lo que se quiere y no cómo conseguirlo, por lo que una instrucción determinada puede ejecutarse internamente de distinta forma en SGBD's diferentes.

Hay que señalar que el resultado de una consulta SQL es siempre una tabla que, a diferencia del modelo relacional teórico, puede contener más de una tupla con los mismos valores. Así pues, en general no obtiene conjuntos sino multiconjuntos de tuplas, ya que puede haber elementos repetidos. Sin embargo, se verá más adelante que se puede forzar la obtención de conjuntos, eliminando las repeticiones.

Las distintas opciones de la instrucción SELECT se ilustrarán con ejemplos de consultas sobre la base de datos *Ciclismo* presentada en el documento UD2.2. Las consultas de los ejemplos se han realizado en el SGBD ORACLE a través de la herramienta SQL Developer.

Para comprender el esquema de la base de datos *Ciclismo* que se usa como ejemplo, se puede consultar el documento UD2.2.

Con el fin de que se pueda seguir con mayor facilidad este documento, se incluye de nuevo aquí el esquema gráfico de la base de datos (Figura 1).

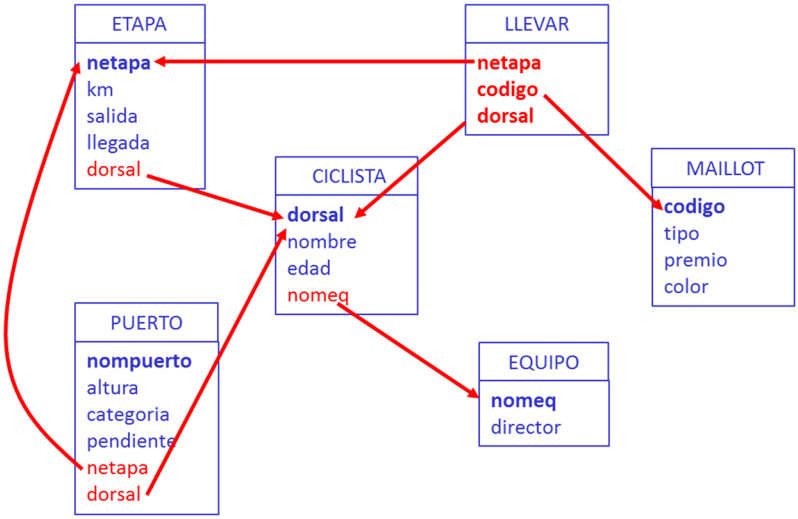


Figura 1: Esquema gráfico de la base de datos *Ciclismo*

Para poder describir la sintaxis de este lenguaje se utiliza una notación especial basada en la notación BNF. La sintaxis informa sobre cuál es el orden de las distintas partes o cláusulas de la instrucción, además de indicar cuáles son obligatorias y cuáles opcionales, y qué elementos se pueden incluir en cada caso.

La notación es la siguiente:

Cursiva	: se usa para nombres de componentes de la base de datos.
MAYÚSCULAS	: se escriben así las palabras reservadas de SQL.
Texto normal	: se usa para los elementos a definir más adelante.
$E_1, E_2, \ldots, E_n$	: lista de elementos E_i separados por comas, donde i>0.
|	: separa opciones alternativas.
[ ]	: indica contenido opcional.
{ }	: indica contenido obligatorio.

Cuando un contenido opcional incluye varias alternativas a elegir (separadas por la barra vertical), siempre debe estar definida una de esas alternativas como valor por defecto, que se aplicará en cado de que no se incluya ninguna de las alternativas.

Antes de presentar las instrucciones es importante aclarar que, en todo este documento, se explicará el funcionamiento de las instrucciones desde un punto de vista didáctico para ayudar a aprender a utilizarlas, sin embargo, debe quedar claro que los sistemas, para ser eficientes, es posible que resuelvan las instrucciones utilizando métodos que no se corresponden con la explicación que se haya podido dar.

2 INSTRUCCIÓN SELECT

2.1 Consultas sencillas sobre una tabla.

En esta sección se mostrará la instrucción SELECT para obtener información de una sola tabla.

Se ilustrará el uso de la consulta con varios ejemplos, para terminar explicando la sintaxis completa de esta instrucción.

Una instrucción SELECT consta, en su forma más sencilla, de cuatro elementos o cláusulas. El Ejemplo 1 es una consulta muy sencilla sobre la base de datos *Ciclismo*.

Ejemplo 1: Obtener el tipo de todos los maillots que hay**.**

```
SELECT tipo
FROM Maillot;
```

El resultado de esta consulta es a su vez una tabla que consta de una sola columna correspondiente al atributo *tipo*, y de tantas filas como tuplas hay en la tabla *Maillot* **.**

Esta consulta nos devolverá la tabla de la Figura 2 .

```
TIPO
--------------------
General
Montaña
Mas Sufrido
Metas volantes
Regularidad
Sprints especiales

6 rows selected
```

Figura 2: Resultado de ejecutar la consulta del ejemplo 1.

Cualquier consulta sobre una tabla en SQL comienza por la palabra reservada SELECT que va seguida al menos de un elemento, y a continuación contiene la palabra FROM seguida de un nombre de tabla.

Lo primero que el SGBD hace al ejecutar una consulta es acceder a la tabla indicada tras la palabra FROM, ya que de allí va a extraer la información especificada por los elementos que siguen a la palabra SELECT.

Ejemplo 2: Obtener el nombre y la edad de todos los ciclistas.

```
SELECT nombre, edad
FROM Ciclista;
```

Como puede verse, tras la palabra SELECT se puede incluir una lista de elementos separados por comas. La consulta resultante será una tabla con tantas columnas como elementos haya detrás de SELECT y antes de FROM y tantas filas como haya en la tabla. En el Ejemplo 2 se obtiene tabla con dos columnas y cien filas.

Ejemplo 3: Obtener edades de las que hay ciclistas.

Si se hace la consulta siguiente:

```
SELECT edad
FROM Ciclista;
```

se obtendría un resultado que no sería el esperado, como puede verse en la Figura 3 (en la que se muestra un fragmento del resultado de dicha consulta), ya que aparecen edades repetidas, en un total de 100 filas, tantas como ciclistas hay en la tabla.

```
EDAD
--------
32
35
27
30
32
33
30
33
100 rows selected
```

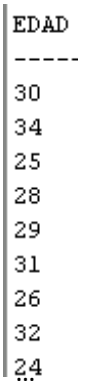
Figura 3: Fragmento del resultado de la consulta: “SELECT edad FROM Ciclista”

En realidad, para responder a la petición del Ejemplo 3 sería suficiente dar las edades distintas de las que hay algún ciclista. Por lo tanto, la consulta adecuada en SQL sería:

```
SELECT DISTINCT edad
FROM Ciclista;
```

La palabra DISTINCT hace que se eliminen del resultado todas las filas repetidas, por lo que la consulta devolverá menos filas que las existentes en la tabla *Ciclista* y obtendrá el resultado mostrado en la

Figura **4**.



```
EDAD
-----
30
34
25
28
29
31
26
32
24
...

15 rows selected
```

Figura 4: Fragmento del resultado de la consulta: "SELECT DISTINCT edad FROM Ciclista"

Ejemplo 4: Obtener toda la información de los equipos.

```
SELECT *
FROM Equipo;
```

El símbolo '*' detrás de la palabra SELECT se usa como una simplificación para representar la lista de todos los atributos de la tabla. En este caso, el resultado sería el mismo que el de la consulta:

```
SELECT nomeq, director
FROM Equipo;
```

Más adelante se mostrarán más posibilidades de selección de elementos en las consultas.

Los ejemplos vistos hasta ahora han obtenido el resultado a partir de todas las filas de una tabla. En muchas ocasiones se desea extraer información relativa sólo a algunas de las filas. Estas filas se caracterizan por medio de condiciones.

Ejemplo 5: Obtener el nombre y la altura de todos los puertos de 1ª categoría.

```
SELECT nompuerto, altura
FROM Puerto
WHERE categoria = '1';
```

En este ejemplo se ha incorporado a la instrucción una nueva parte o cláusula, que comienza con la palabra reservada WHERE y que va seguida de una condición, que en este caso es una comparación de igualdad. Con esta condición se le está indicando al SGBD que sólo debe extraer el valor de *nompuerto* y *altura* de aquellas filas de *Puerto* para las que el atributo *categoria* tome el valor '1'.

El SGBD resolverá esta consulta accediendo a la tabla *Puerto* y recorriendo una a una sus filas al tiempo que comprueba para cada una de ellas la condición indicada tras la palabra WHERE. Sólo aquellas filas para las que la condición es cierta serán las utilizadas para extraer la información requerida.

En muchas consultas se desea que la información se muestre en un orden determinado.

Ejemplo 6: Obtener el nombre, la altura y la categoría de todos los puertos ordenados por altura y categoría.

```
SELECT nompuerto, altura, categoria
FROM Puerto
ORDER BY altura, categoria;
```

Se ha usado una nueva cláusula que comienza por las palabras reservadas ORDER BY, seguida por uno o varios nombres de atributos. En la respuesta a esta consulta, los puertos aparecerán en orden ascendente

respecto a su altura y, si hubiera dos puertos con la misma altura, éstos estarían ordenados respecto a la categoría, también en orden ascendente.

Con la notación que se presentó más arriba, la sintaxis de una consulta sencilla sobre una tabla es la siguiente:

```
SELECT [ALL | DISTINCT] {expresión1, expresión2,..., expresiónn | *}
FROM tabla
[WHERE condición]
[ORDER BY columna1, columna2,..., columnan]
```

En realidad, un SGBD relacional ejecuta las cláusulas de la instrucción SELECT en un orden diferente al de la sintaxis. El orden es el siguiente:

```
4 SELECT [ALL | DISTINCT] {expresión1, expresión2,..., expresiónn | *}
1 FROM tabla
2 [WHERE condición]
3 [ORDER BY columna1, columna2,..., columnan]
```

Esto significa que el SGBD[1]:

1. Accede a la tabla de la que se quiere extraer la información.
2. Recorre fila por fila aplicando la condición y quedándose sólo con las filas que la hacen cierta.
3. Ordena el resultado.
4. Selecciona los elementos solicitados (o calcula las funciones agregadas).

Los elementos de la instrucción que todavía no se habían definido se presentan a continuación:

- [ALL | DISTINCT]: esta opción permite incluir una de estas dos palabras detrás de SELECT, con el significado siguiente:
  - ALL: Permite la aparición de filas idénticas (es el valor por defecto).
  - DISTINCT: Elimina del resultado las filas repetidas.
- Como *expresión* se pueden incluir:
  - Nombres de atributos.
  - Constantes numéricas, literales o fechas (en los dos casos últimos irán entre comillas).
  - Funciones agregadas, que devuelven valores obtenidos a partir de varias filas.
  - Expresiones aritméticas formadas con elementos de todos los tipos anteriores que sean numéricos.
  - El símbolo '*'.

En los casos en que sea necesario, una expresión puede ir seguida de un nombre separado de ella por un espacio en blanco. Este nombre será el asignado a la columna correspondiente a la expresión en la tabla resultante. (Véase el ejemplo 7).

La sintaxis de uso de una función agregada es la siguiente:

```
COUNT (*)|
{AVG|MAX|MIN|SUM|COUNT} ([ALL | DISTINCT] expresión)
```

Esto significa que se puede elegir entre dos opciones a un primer nivel:

- COUNT (*): es una función agregada que da como resultado la cantidad de filas de la selección; es decir, cuenta las filas.
- {AVG|MAX|MIN|SUM|COUNT} ([ALL|DISTINCT] expresión): esta opción indica que, a su vez, se debe elegir uno de los nombres (abreviado) de las funciones, que son, respectivamente la media (*average* en inglés), el máximo, el mínimo, la suma y la cuenta, seguido de un paréntesis dentro del que se pueden incluir algunos de los elementos vistos en *expresión*, precedidos o no de las palabras ALL o DISTINCT, con el mismo significado explicado antes. Se excluye de esta opción el

[1] Esta forma de imaginar cómo el SGBD resuelve la consulta puede no coincidir con la realidad ya que, para resolver de forma eficiente una consulta los sistemas pueden utilizar otros métodos.

'*' porque sólo se puede usar con la función COUNT como se ha visto antes, y en ese caso no se puede añadir ni ALL ni DISTINCT.

Hay que tener en cuenta lo siguiente:

- Para las funciones SUM y AVG los argumentos deben ser numéricos.
- Los cálculos se hacen después de la selección y de aplicar las condiciones.
- Los valores nulos son eliminados antes de realizar los cálculos (incluido COUNT).
- Si el número de filas de la selección es 0, la función COUNT devuelve el valor 0 y las otras funciones devuelven el valor nulo, algo lógico ya que la cuenta de un conjunto de filas vacío es 0, y no puede saberse cuál es el máximo, por ejemplo, de un conjunto de filas vacío, por lo que está indefinido.

Ejemplo 7: Obtener el número de ciclistas que hay y su edad media.

```
SELECT 'Núm. de ciclistas =', COUNT (*) Cuenta, 'Edad media=', AVG (edad)
FROM Ciclista;
```

El resultado de esta consulta es una única fila, como la que se muestra en la Figura 5. Nótese cómo la función COUNT (*) se ha renombrado como *Cuenta.*

'NÚM.DECICLISTAS='	CUENTA	'EDADMEDIA='	AVG(EDAD)
Núm. de ciclistas =	100	Edad media=	29,89

Figura 5: Resultado de la consulta del Ejemplo 7

En las consultas sencillas, la selección no podrá incluir a la vez referencias a funciones agregadas y a atributos, ya que las funciones devuelven un único valor, por lo que el resultado de la consulta es una tabla con una sola fila, mientras que los atributos pueden devolver varios valores, es decir, una tabla con varias filas, lo que es incompatible en un sólo resultado.

A continuación, se muestra un ejemplo de consulta incorrecta:

Ejemplo 8:

```
SELECT nombre, AVG (edad)
FROM Ciclista;
```

La consulta es incorrecta debido a que se están solicitando nombres de ciclistas, de los que hay más de uno, junto a la media de la edad, que da como resultado un sólo valor, por lo que no sería aceptada por el SGBD y daría un error.

Una vez descritas las posibilidades de la cláusula SELECT, a continuación, se presenta la sintaxis de la cláusula WHERE. Como se ha dicho, el hecho de que en la sintaxis aparezca entre corchetes indica que es opcional, es decir, que se puede omitir. Si no se usa, el sistema extraerá la información para la respuesta a partir de todas las filas de la tabla. Cuando se incluye, la palabra WHERE va seguida de una condición.

Una condición es una expresión más o menos compleja, que al evaluarse toma un valor lógico: *cierto*, *falso* o *indefinido*. La cláusula WHERE indica al SGBD que utilice para construir el resultado de la consulta sólo aquellas tuplas o filas de la tabla para las que la condición sea cierta.

La *condición* está formada por un conjunto de predicados combinados con las conectivas lógicas AND, OR y NOT, y con el uso de paréntesis.

Los predicados utilizados que permiten comparar columnas son:

- Predicados de comparación: =, <>, >, <, >=, <=.

  Los predicados de comparación se escriben entre dos operandos que se van a comparar, y que pueden ser expresiones con atributos o constantes; así se forman condiciones simples. Varias de estas condiciones se pueden combinar por medio de los operadores lógicos AND, OR o NOT y usando paréntesis en caso de que sea necesario para formar condiciones más complejas.

Ejemplo 9: Obtener el nombre de todos los ciclistas del equipo Banesto de más de 27 años.

```
SELECT nombre
FROM Ciclista
WHERE nomeq= 'Banesto' AND edad > 27;
```

- **Predicado LIKE**: Sirve para comparar una tira de caracteres con un patrón. Se usa cuando se quiere buscar una tira de caracteres de la que sólo se conocen algunas características.

  Este predicado requiere a la izquierda un atributo de tipo tira de caracteres, y a la derecha una cadena de caracteres que puede incluir algún carácter especial que será utilizado como patrón. En SQL estándar (y en ORACLE), estos caracteres especiales son '%', que representa una cadena de cualquier longitud, y '_', que representa cualquier carácter.

Ejemplo 10: Obtener el número de las etapas para las que el nombre de la ciudad de llegada tenga por segunda letra una 'e' o el nombre de la ciudad de salida lleve dos o más letras 'A'.

```
SELECT netapa
FROM Etapa
WHERE llegada LIKE '_e%' OR salida LIKE '%A%A%';
```

- **Predicado BETWEEN**: Permite comprobar si un escalar está en un rango comprendido entre dos valores.

  Este predicado requiere a la izquierda una expresión y a la derecha dos valores del mismo tipo de la expresión separados por la palabra AND, siendo el primero menor o igual que el segundo.

Ejemplo 11: Obtener el nombre de los ciclistas cuya edad está entre 20 y 30 años.

```
SELECT nombre
FROM Ciclista
WHERE edad BETWEEN 20 AND 30;
```

El predicado BETWEEN es equivalente a una condición con comparaciones de la siguiente forma:

```
expresión BETWEEN expresión1 AND expresión2
```

es equivalente a

```
(expresión >= expresión1) AND (expresión<= expresión2)
```

- **Predicado IN**: sirve para comprobar si un valor está dentro de un conjunto de valores.

Ejemplo 12: Obtener el nombre de los puertos de 1ª, 2ª o 3ª categoría.

```
SELECT nompuerto FROM Puerto
WHERE categoria IN ('1','2','3');
```

El predicado IN también es derivado, de manera que:

```
expresión IN (expresión1, expresión2, ..., expresiónn)
```

es equivalente a

```
(expresión = expresión1) OR (expresión = expresión2) OR ...
       OR (expresión = expresiónn)
```

- **Predicado IS NULL**: permite comprobar si el valor de un atributo es nulo.

  No se pueden hacer comparaciones entre cualquier valor y el valor NULL, no está permitido. Por otra parte, la comparación con un atributo que tome para una fila el valor nulo se evalúa como *indefinido*.

Ejemplo 13: Obtener la información de las etapas que no llegan a la misma ciudad de la que salen.

```
SELECT *
FROM Etapa
WHERE salida < > llegada;
```

Si el valor de uno de esos dos atributos en una fila fuera nulo, el resultado de la comparación sería indefinido y la fila no se seleccionaría.

Ejemplo 14: Obtener el nombre de los equipos que no tienen director.

La forma correcta es:

```
SELECT nomeq
FROM Equipo
WHERE director IS NULL;
```

La expresión "WHERE director = NULL" sería incorrecta, y daría un error de sintaxis, porque no está permitido usar los operadores de comparación con el valor NULL.

Un elemento *expresión* puede usar operadores aritméticos (por ejemplo: +, –, *, /, etc. para la suma, diferencia, producto, división, etc., respectivamente). Este uso está permitido tanto en la selección como en la condición.

Ejemplo 15: Obtener el tipo y el premio en dólares (supóngase que está en euros y que 1$ = 0.75 €) de aquellos maillots cuyo premio supere los 100 dólares.

```
SELECT tipo, premio/0.75 FROM Maillot
WHERE premio/0.75 > 100;
```

2.2 Consultas simples sobre varias tablas

Cuando la información que se desea obtener de la base de datos está almacenada en varias tablas, la consulta debe incluir dichas tablas en la cláusula FROM para indicarle al SGBD de dónde debe extraer la información.

Al incluir dos tablas en esta cláusula, el SGBD se prepara para poder obtener la información haciendo el **producto cartesiano** de las tablas; es decir, forma todas las posibles tuplas combinando cada una de las tuplas de la primera tabla con cada una de las tuplas de la segunda.

Ejemplo 16: Obtener toda la información de etapas y puertos.

```
SELECT *
FROM Etapa, Puerto;
```

El resultado de esta consulta contiene 294 filas, que es el resultado de combinar las 21 etapas con los 14 puertos (21 x 14 = 294), parte de las cuales se muestran en la Figura 6.

NETAPA	KM	SALIDA	LLEGADA	DORSAL	NOMPUERTO	ALTURA	CATEGORIA	PENDIENTE	NETAPA_1	DORSAL_1
1	9	Valladolid	Valladolid	1	Alto del Naranco	565	1	6,9	10	30
2	180	Valladolid	Salamanca	36	Alto del Naranco	565	1	6,9	10	30
3	240	Salamanca	Caceres	12	Alto del Naranco	565	1	6,9	10	30
4	230	Almendralejo	Córdoba	83	Alto del Naranco	565	1	6,9	10	30
5	170	Córdoba	Granada	27	Alto del Naranco	565	1	6,9	10	30
6	150	Granada	Sierra Nevada	52	Alto del Naranco	565	1	6,9	10	30
7	250	Baza	Alicante	22	Alto del Naranco	565	1	6,9	10	30
8	40	Benidorm	Benidorm	1	Alto del Naranco	565	1	6,9	10	30
9	150	Benidorm	Valencia	35	Alto del Naranco	565	1	6,9	10	30
10	200	Igualada	Andorra	2	Alto del Naranco	565	1	6,9	10	30
...										
16	160	Santander	Lagos de Covadonga	5	Sierra Nevada	2500	E	6	2	26
17	140	Cangas de Onis	Alto del Naranco	4	Sierra Nevada	2500	E	6	2	26
18	195	Ávila	Ávila	8	Sierra Nevada	2500	E	6	2	26
19	190	Ávila	Destilerias Dyc	2	Sierra Nevada	2500	E	6	2	26
20	52	Segovia	Destilerias Dyc	2	Sierra Nevada	2500	E	6	2	26
21	170	Destilerias Dyc	Madrid	27	Sierra Nevada	2500	E	6	2	26

Figura 6: Resultado (parcial) de la consulta "SELECT * FROM Etapa, Puerto"

Si en la cláusula FROM se incluyen tres tablas o más, el resultado de multiplicar las dos primeras se multiplica a su vez por la tercera, y así sucesivamente, con lo que la tabla que forma el SGBD para resolver la consulta puede tener fácilmente miles de tuplas.

Consultas de este tipo no son habituales, ya que la mayoría de las filas formadas combinando todas las tuplas de todas las tablas no tienen ninguna propiedad en común, es decir, no aportan información interesante.

Lo habitual es añadir una condición que seleccione, de entre todas las combinaciones formadas, aquellas que interese conocer.

Al construir estas condiciones y también al seleccionar algunos atributos, puede surgir un problema: puede haber nombres de atributos coincidentes en las distintas tablas, de manera que la expresión formada en la condición sea ambigua para el SGBD. En la tabla formada por la consulta del ejemplo 16 hay dos columnas llamadas *netapa*, y dos llamadas *dorsal*. En la Figura 6 puede verse que ORACLE ha renombrado las repeticiones para que tengan distinto nombre. En SQL estándar ese problema se resuelve calificando los nombres de atributos con el nombre de la tabla a la que corresponden, usando una notación de punto.

Ejemplo 17: Obtener pares de números de etapas y nombres de puertos ganados por el mismo ciclista.

```
SELECT Etapa.netapa, nompuerto
FROM Etapa, Puerto
WHERE Etapa.dorsal=Puerto.dorsal;
```

El atributo *netapa* de la selección sería ambiguo si no se especificara cuál se desea (y en este ejemplo, el valor de *netapa* en *Etapa* no tiene por qué coincidir con el valor de *netapa* en *Puerto*, así que se debe especificar el adecuado, en este caso el de *Etapa*). De la misma manera, las columnas *dorsal* de la condición deben calificarse con el nombre de la tabla (*Etapa* y *Puerto* respectivamente); si no, la expresión es ambigua y el SGBD no puede saber a cuál se refiere la consulta, por lo que daría error.

Esta forma de calificar los atributos hace demasiado largas algunas consultas, por lo que en SQL se puede usar, en lugar del nombre de la tabla, una *variable_recorrido.*

Una *variable_recorrido* es un nuevo nombre o *alias* para referirse a una tabla. Normalmente es más corta que el nombre de la tabla, y se suele usar la inicial de la tabla, a veces seguida de algún carácter más si coinciden las iniciales de varias tablas. Muchas veces se usa para abreviar la escritura de las consultas, pero en algunos casos es obligatorio usarla: cuando interviene la misma tabla más de una vez en la misma consulta. Entonces es imprescindible renombrar al menos una de esas participaciones, para distinguir los atributos de cada una (Véase el Ejemplo 21).

La sintaxis para calificar los atributos es la siguiente:

```
[tabla | variable_recorrido].columna
```

Y la de definición de la *variable_recorrido* es:

```
tabla [AS] variable_recorrido
```

El ejemplo 17 puede escribirse de forma más abreviada usando la *variable_recorrido*, como se presenta en el Ejemplo 18.

Ejemplo 18: Obtener pares de números de etapas y nombres de puertos ganados por el mismo ciclista.

```
SELECT E.netapa, nompuerto
FROM Etapa E, Puerto P
WHERE E.dorsal = P.dorsal;
```

En el ejemplo 18 se ha establecido una relación entre el atributo *E.dorsal* y *P.*dorsal, y ninguno de los dos es clave ajena que se refiera al otro. Este uso es correcto, pero, lo más frecuente es que la condición que relaciona dos tablas sea una igualdad entre la clave ajena de una tabla y la clave primaria de la otra tabla a la que se hace referencia. Pueden verse condiciones de ese tipo en los ejemplos siguientes.

Ejemplo 19: Obtener los nombres de los ciclistas pertenecientes al equipo dirigido por Álvaro Pino.

```
SELECT nombre
FROM Ciclista C, Equipo E
WHERE C.nomeq = E.nomeq AND E.director = 'Álvaro Pino';
```

Se puede ver que en este caso hay dos comparaciones en la condición, unidas por un operador AND. La

primera de las comparaciones selecciona, de todo el producto cartesiano, aquellas filas que están relacionadas por su igualdad en el atributo *nomeq*. Es decir, se extraen sólo las filas que tiene información de un ciclista y de su equipo. La segunda comparación selecciona las filas en las que el director del equipo es Álvaro Pino. Al estar combinadas las dos condiciones con AND, sólo se seleccionan las filas que cumplan ambas.

En general, por cada tabla de más que se necesite en la cláusula FROM, se debe añadir una comparación a la condición. Así, si hay cuatro tablas en la cláusula FROM, debe haber tres comparaciones que permitan relacionarlas entre sí. Se debe recordar que, si no se eligen bien las condiciones, el número de filas resultantes puede ser muy grande y no responder a lo que se solicitaba.

Ejemplo 20: Obtener los directores de los equipos cuyos ciclistas han llevado algún maillot de color amarillo.

```
SELECT DISTINCT director
FROM Equipo E, Ciclista C, Llevar L, Maillot M
WHERE E.nomeq = C.nomeq AND
      C.dorsal=L.dorsal AND
      L.codigo=M.codigo AND
      M.color='Amarillo';
```

Como se ha comentado más arriba, es muy frecuente que las condiciones para relacionar las tuplas de las distintas tablas sean comparaciones de igualdad entre una clave ajena y la clave primaria a la que se refiere. No es obligatorio que sea así, las comparaciones pueden incluir expresiones, como en el Ejemplo 21, u otros predicados de comparación.

Ejemplo 21: Obtener la llegada de cada etapa y la salida de la etapa siguiente.

```
SELECT E1.llegada, E2.salida
FROM Etapa E1, Etapa E2
WHERE E1.netapa + 1= E2.netapa;
```

La sintaxis de las consultas con varias tablas es la siguiente:

```
SELECT [ALL | DISTINCT] {expresión1, expresión2,..., expresiónn | *}
FROM tabla1, tabla2 …, tablan
[WHERE condición]
[ORDER BY columna1, columna2,..., columnan]
```

2.2.1 Aparición de filas repetidas en consultas de varias tablas

Después de combinar varias tablas (sean, por ejemplo, *R* y *S*) en una consulta y de aplicar la condición correspondiente, una misma fila de una tabla *R* puede aparecer relacionada con varias filas de otra tabla *S*. Si la consulta pide información sólo de *R*, se pueden obtener filas repetidas, que en la mayoría de los casos se deben eliminar pues no son apropiadas en la respuesta.

Ejemplo 22: Obtener número y longitud de las etapas que tienen puertos de montaña.

```
SELECT DISTINCT E.netapa, km
FROM Etapa E, Puerto P
WHERE E.netapa = P.netapa;
```

En esta consulta se hace necesario incluir la palabra DISTINCT ya que, en esta base de datos, hay etapas en las que se encuentran varios puertos de montaña. Por cada uno de los puertos, la consulta obtiene una fila, pero no es necesario que salga repetida esa información, hay que obtener sólo una fila por cada etapa con puertos. En la Figura 7 se muestra el resultado de este ejemplo.

NETAPA	KM
15	207
10	200
11	195
19	190
2	180
16	160
18	195

Figura 7: Resultado de la consulta del Ejemplo 22

En la Figura 8 se muestra el resultado que se obtiene con esa misma consulta si se elimina la palabra DISTINCT. Puede verse que aparecen varias filas repetidas, debido a que esas etapas tienen varios puertos cada una, y sale su información una vez por cada puerto.

NETAPA	KM
10	200
10	200
11	195
10	200
10	200
11	195
16	160
19	190
15	207
19	190
18	195
18	195
18	195
2	180

Figura 8: Resultado de la consulta del Ejemplo 22 sin DISTINCT

2.3 Consultas complejas: Subconsultas

Una *subconsulta* es una sentencia SELECT entre paréntesis.

Para ilustrar el uso de subconsultas, a continuación se incluye un ejemplo.

Ejemplo 23: Obtener número y longitud de las etapas que tienen puertos de montaña.

Se puede observar que los atributos solicitados (*netapa* y *km*) se encuentran en la tabla *Etapa*, y que la tabla *Puerto* sólo se necesita para comprobar qué etapas de la tabla *Etapa* aparecen en ella.

Para este ejemplo se muestran dos soluciones: la solución a) sin subconsultas es la que se haría con las consultas sencillas que se han explicado hasta ahora de SQL.

a) Solución sin subconsultas:

```
SELECT DISTINCT E.netapa, km
FROM Etapa E, Puerto P
WHERE E.netapa = P.netapa;
```

b) Solución con subconsultas

```
SELECT netapa, km
FROM Etapa
WHERE netapa IN
    (SELECT netapa
     FROM Puerto);
```

La solución b), que usa una subconsulta, comienza con una consulta a la que se llama *principal*. En ella se

seleccionan los atributos requeridos de la tabla *Etapa* que es la única que aparece en la cláusula FROM, y en la condición, que usa el predicado IN ya estudiado, en vez de enumerar una serie de valores posibles, se incluye una consulta. Como esta última consulta está dentro de otra, se llama subconsulta. Mirando dicha subconsulta en el ejemplo, se deduce que la respuesta a la misma va a ser una lista de números de etapa, es decir aquellos valores de *netapa* que aparecen en *Puerto*. O sea, que en el uso del predicado IN se ha sustituido la lista de valores que iría detrás de él entre paréntesis, por una consulta que calcula esos valores.

Se puede observar que la consulta b) es más legible que la a), porque está constituida por dos consultas más sencillas, y esta ventaja se nota más cuantas más tablas sean necesarias para obtener la respuesta.

También se puede apreciar que en la solución b) no es necesario usar DISTINCT ya que en la consulta principal las tuplas de *Etapa* no aparecen repetidas.

2.3.1 Características de las subconsultas

Como se ha dicho, una subconsulta es una consulta que se incluye dentro de otra, encerrada entre paréntesis. Esto quiere decir que cualquier consulta se puede incluir como subconsulta de otra.

El uso de subconsultas, en muchos casos, es opcional; es decir, que muchas consultas pueden resolverse indistintamente usando todas las tablas en la cláusula FROM, o usando subconsultas. Aunque hay algunos casos, como cuando se quieren hacer comparaciones con valores obtenidos por las funciones agregadas, en que es imprescindible usar subconsultas.

Por otra parte, no siempre se pueden utilizar. Para poder incluir subconsultas es necesario que los atributos que se quieren obtener pertenezcan a tablas que están en la consulta principal. Las subconsultas incluirán sólo aquellas tablas que se usan para comprobar condiciones, no para extraer la información a obtener. Esto se debe a que la consulta principal sólo 've' el resultado de la subconsulta, no las tablas que ha utilizado y, por lo tanto, no puede usar los atributos de dichas tablas, sólo puede usar los valores resultantes como operandos de los predicados. Por el contrario, desde las subconsultas sí que se pueden usar todos los atributos de las tablas que hay en la consulta principal.

2.3.2 Ejecución de las consultas con subconsultas

Es importante comprender cómo ejecutan los SGBD's las consultas con subconsultas.

En principio el SGBD accede a la tabla o tablas de la consulta principal. Si hay varias tablas, obtiene una tabla que será la combinación de todas las tuplas de todas las tablas entre sí, como se vio para una consulta normal. Luego, recorre fila por fila dicha tabla resultante y para cada fila que selecciona, a la que llamaremos *tupla actual*, le aplica las condiciones de la cláusula WHERE. Si ésta incluye una subconsulta, el SGBD la ejecutará y el resultado obtenido para ella será utilizado para evaluar la condición de la consulta principal. Esto significa que para cada fila de la consulta principal se ejecuta la subconsulta. En muchos casos el resultado de la subconsulta es el mismo para todas las filas de la consulta principal, si la subconsulta es independiente de aquella. Pero en otros muchos casos, la subconsulta usa en sus condiciones atributos de la consulta principal, y por lo tanto el resultado de la subconsulta varía dependiendo de cuál sea la tupla actual de la consulta principal.

2.3.3 Anidamiento de subconsultas

Dado que cualquier consulta puede usarse como subconsulta, una consulta que contenga una subconsulta puede a su vez ser subconsulta de otra, produciendo así un anidamiento de consultas a varios niveles. Esta situación es normal, y no hay límite en el número de niveles de anidamiento, aunque muchos niveles pueden dificultar la comprensión de lo que se quiere obtener.

Una opción intermedia en estos casos es hacerla mixta, es decir, usar subconsultas en la consulta principal para evitar el uso de DISTINCT, pero luego usar las tablas restantes en la subconsulta.

Ejemplo 24: Obtener los directores de los equipos cuyos ciclistas han llevado algún maillot de color amarillo.

Para este ejemplo se ha utilizado el mismo enunciado del Ejemplo 20, y aquí se muestran tres soluciones:

a) sin usar subconsultas (que ya se vio en el ejemplo citado), b) usando sólo subconsultas, es decir, cada tabla en una subconsulta) y c) mixta, o sea una consulta principal y una subconsulta con las tablas restantes para construir la condición. Nótese que en la solución a) es imprescindible el uso de DISTINCT.

a) Solución sin subconsultas:

```
SELECT DISTINCT director
FROM Equipo E, Ciclista C, Llevar L, Maillot M
WHERE E.nomeq = C.nomeq AND C.dorsal=L.dorsal AND
    L.codigo=M.codigo AND M.color='Amarillo';
```

b) Solución sólo con subconsultas:

```
SELECT director
FROM Equipo
WHERE nomeq IN
       (SELECT nomeq
       FROM Ciclista
       WHERE dorsal IN
              (SELECT dorsal
              FROM Llevar
              WHERE codigo IN
                     (SELECT codigo
                     FROM Maillot
                     WHERE color='Amarillo')));
```

c) Solución mixta:

```
SELECT director
FROM Equipo E
WHERE nomeq IN
       (SELECT C.nomeq
       FROM Ciclista C, Llevar L, Maillot M
       WHERE C.dorsal=L.dorsal AND L.codigo=M.codigo AND
              M.color='Amarillo');
```

2.3.4 Predicados que se pueden usar con subconsultas

Con las subconsultas pueden usarse diferentes predicados. Aquí se presentan los más comunes:

1. Predicados de comparación.
2. Predicado IN.
3. Predicado EXISTS.

2.3.4.1 Predicados de comparación

Los predicados de comparación son los mismos que se ya estudiaron para comparar atributos con valores u otros atributos, es decir {=, <>, >, <, >=, <=}.

Estos predicados permiten comparar dos elementos del mismo tipo. El uso aquí es comparar un atributo con el valor resultante de una subconsulta. Para poder usar subconsultas con estos predicados, hay que asegurarse de que la subconsulta devuelve un único valor. Además, el valor obtenido debe ser del mismo tipo que el atributo, para poder compararlo.

Si una subconsulta puede devolver varias filas, entonces no se puede usar con estos predicados de comparación, ya que no está definida una comparación de un elemento con una lista de elementos.

Si el resultado de la subconsulta está vacío, entonces se considera que es el valor nulo, y el valor resultante de la comparación sería indefinido.

Ejemplo 25: Obtener los nombres de los puertos cuya altura es mayor que la media de la altura de los puertos de segunda categoría.

```
SELECT nompuerto
FROM Puerto
WHERE altura > (SELECT AVG (altura)
                FROM Puerto
                WHERE categoría = '2');
```

La subconsulta ejecutada como una consulta sobre la base de datos de *Ciclismo* obtiene la media de la altura de los puertos de segunda categoría, y da como resultado 1640,5, por lo que ese será el valor con el que se compare la altura de cada tupla de la tabla *Puerto*.

Hay que resaltar que para resolver la petición de este ejemplo es necesario usar una subconsulta, pues las funciones agregadas sólo pueden obtenerse como objetivo de una consulta usándose a continuación de la palabra SELECT.

Sería totalmente incorrecto usar una comparación como la siguiente:

```
… WHERE altura > AVG (altura)
```

Es una sintaxis incorrecta, ya que AVG debe ser un objetivo de una consulta SELECT y, si fuera posible esta instrucción, no tendría sentido calcular una función agregada para una sola tupla

2.3.4.2 Predicado IN

En el Ejemplo 23 ya se ha visto el uso del predicado IN con una subconsulta. Su significado es el de comprobar si un valor se encuentra en una lista de valores del mismo tipo.

La sintaxis del predicado es:

```
expresión [NOT] IN (expresión_tabla)
```

Como ya se comentó, el predicado usa la respuesta a la subconsulta (que en la sintaxis aparece como *expresión_tabla*) como la lista de valores en la que debe buscar el valor de la expresión que aparece a la izquierda. La subconsulta debe tener un sólo elemento en la cláusula SELECT y el tipo de datos de éste debe ser el mismo que el de la expresión de la izquierda.[2]

Si la subconsulta da un resultado vacío, entonces el valor resultante para el predicado sería *falso*.

Como puede verse, el predicado IN puede ir precedido de la palabra NOT, con el significado siguiente: una expresión no está en una lista de valores (NOT IN) si es distinta de todos los valores de la lista. Este significado puede producir resultados inesperados cuando la subconsulta devuelve algún valor nulo, ya que el SGBD comparará con todos los valores resultantes de la subconsulta y, al comparar con el valor nulo obtendrá un resultado indefinido, por lo que la condición de la cláusula WHERE no se hará cierta para ninguna tupla (ya que la negación del valor *indefinido* es también el valor *indefinido*).

Ejemplo 26: Obtener los dorsales y los nombres de los ciclistas cuya edad no coincide con la de ninguno de los ciclistas del equipo Kelme.

```
SELECT dorsal, nombre
FROM Ciclista
WHERE edad NOT IN (SELECT edad
                   FROM Ciclista
                   WHERE nomeq = 'Kelme');
```

Esta consulta en la base de datos *Ciclismo* devuelve 53 tuplas, porque es cierto para esos 53 ciclistas que su edad es distinta de las de todos los de Kelme y porque no hay ningún valor *nulo* en el atributo edad; pero si una de las edades de los ciclistas de Kelme no se conociera y fuera nula, el resultado del ejemplo estaría vacío. La razón es que sólo puede dar como respuesta aquellos ciclistas para los que, después de comparar su edad con todas las edades de los de Kelme y con NULL, obtenga para todas las comparaciones un valor *falso*, que con la partícula NOT se transformaría en *cierto*, pero la comparación con NULL da *indefinido.*

Es importante desechar un error en el que se cae con frecuencia en los inicios del aprendizaje de las consultas SQL: no tiene sentido usar una consulta con varias tablas en la cláusula FROM para extraer

[2] Ésta es una versión simplificada del predicado IN con subconsultas.

información de una tupla de una de las tablas que no esté relacionada con tuplas de las otras basándose en el predicado <>. Este error se ilustra con una solución incorrecta al mismo enunciado del Ejemplo 26, a continuación:

Ejemplo 27: Obtener los dorsales y los nombres ciclistas cuya edad no coincide con la de ninguno de los ciclistas del equipo Kelme.

```
SELECT dorsal, nombre
FROM Ciclista C1, Ciclista C2
WHERE C1.edad <> C2.edad AND
      C2.nomeq = 'Kelme';
```

Para que una tupla de *Ciclista* aparezca en el resultado de esta consulta, sólo hace falta que la edad sea distinta de alguna edad de los ciclistas de Kelme, no distinta de las edades de todos

Ejemplo 28: Obtener el dorsal y el nombre de los ciclistas que no han ganado ninguna etapa.

a) Solución correcta:

```
SELECT dorsal, nombre
FROM Ciclista
WHERE dorsal NOT IN
                  (SELECT dorsal
                   FROM Etapa
                   WHERE dorsal IS NOT NULL)
```

b) Solución incorrecta

```
SELECT dorsal, nombre
FROM Ciclista C, Etapa E
WHERE C.dorsal <> E. dorsal;
```

La solución correcta a) selecciona los datos pedidos de los ciclistas cuyo dorsal no aparece entre los dorsales de la tabla *Etapa*, es decir, que no han ganado ninguna etapa. La solución b), que es incorrecta, seleccionaría todos los ciclistas, ya que combina todas las tuplas de *Ciclista* con todas las de *Etapa* y selecciona a aquellos cuyo dorsal es distinto del que ha ganado la etapa. Sólo con que haya una etapa que un ciclista no ha ganado, ese ciclista saldría en el resultado, aunque hubiera ganado otras (y no debería salir).

2.3.4.3 Predicado EXISTS

Este predicado es unario, a diferencia de los anteriores que son binarios. Como consecuencia no compara dos operandos, sino sólo uno, una subconsulta, y devuelve el valor *cierto* o el valor *falso* dependiendo de que la subconsulta devuelva tuplas resultantes o no, respectivamente. Así, podemos decir que el predicado EXISTS aplicado a una subconsulta es *cierto* cuando existe alguna tupla como resultado de la subconsulta, y *falso* en caso contrario.

Se puede ver que una consulta que contenga la condición siguiente:

```
WHERE EXISTS (SELECT * FROM…),
```

es equivalente a otra en la que la condición sea:

```
WHERE (SELECT COUNT (*) FROM…)>0
```

En el ejemplo 29 se muestra una consulta que puede resolverse indistintamente usando el predicado IN visto en la sección anterior o el predicado EXISTS, como sucede en muchas consultas, ya que estos dos predicados son intercambiables en general. Hay que decir que el uso del predicado NOT EXISTS no tiene el efecto indeseado que tiene el predicado NOT IN cuando hay valores nulos, por lo que el uso de aquel está más extendido y es más recomendable.

Ejemplo 29: Obtener el nombre de los ciclistas que han llevado un maillot con premio menor de 50000.

a) Solución con predicado IN:

```
SELECT nombre
FROM Ciclista C
WHERE C.dorsal IN
        (SELECT L.dorsal
         FROM Llevar L, Maillot M
         WHERE L.codigo=M.codigo AND M.premio<50000);
```

b) Solución con predicado EXISTS:

```
SELECT nombre
FROM Ciclista C
WHERE EXISTS
        (SELECT *
         FROM Llevar L, Maillot M
         WHERE L.codigo=M.codigo AND M.premio<50000 AND C.dorsal= L.dorsal);
```

Estas dos soluciones son equivalentes.

En la solución a) la subconsulta devuelve los dorsales de los ciclistas que han llevado un maillot de premio inferior a 50000 (en todas las ejecuciones serán los mismos dorsales), y la consulta principal compara el dorsal de cada tupla de *Ciclista* con los obtenidos por la subconsulta.

En cambio, en la solución b) la subconsulta extrae toda la información de aquellas tuplas de *Llevar* combinado con *Maillot* a través del código tales que el premio sea menor de 50000, y con una condición nueva: que el dorsal del que ha llevado el maillot coincida con el dorsal del ciclista actual de la consulta principal; a éste se hace referencia por su nombre, cualificado con el nombre o el alias de su tabla; en este ejemplo es *C.dorsal* (recuérdese que las tablas de la consulta principal son visibles desde las subconsultas). Es decir, la subconsulta sólo obtiene las tuplas que en las que el ciclista actual de la consulta principal ha llevado un maillot de premio menor que 50000; recuérdese que al ejecutar las consultas con subconsultas para cada tupla de la consulta principal se calcula la subconsulta. En este caso, el resultado de la subconsulta será distinto para cada tupla de la consulta principal, ya que incluye una condición que depende de ella.

El resultado de evaluar el predicado EXISTS dependerá de si hay o no alguna tupla resultante de la subconsulta. Si hay una, varias o muchas, devuelve *cierto*, por lo que la tupla actual de la consulta principal se selecciona para el resultado. Si no hay ninguna tupla en el resultado de la subconsulta, el predicado devuelve *falso*, y no se selecciona la tupla actual de la consulta principal.

En las subconsultas que siguen al predicado EXISTS se suele usar 'SELECT *', no porque se necesite toda la información, sino porque esa es la forma más breve de escribir el objetivo de la subconsulta, y de ella no hace falta ningún atributo en concreto, sólo interesa saber si hay alguna tupla que cumpla las condiciones. Esta es la forma habitual de escribir la selección con el predicado EXISTS.

En una misma consulta, una tabla puede usarse tanto en la consulta principal como en alguna subconsulta. Lo más habitual es renombrar con un *alias* al menos una de ellas, pero si no se hace y coinciden en el nombre, o si se renombran con un mismo nombre, no es un error de sintaxis, pero hay que tener cuidado, porque predomina la tabla de la subconsulta, es decir, cualquier uso de dicho nombre se referirá a la tabla de la subconsulta y no a la de la consulta principal. Como recomendación general, cada vez que se use una tabla en una consulta se aconseja darle un nombre distinto.

La sintaxis del predicado EXISTS es la siguiente:

```
[NOT] EXISTS (expresión_tabla)
```

Esto significa que puede ir precedido de la palabra NOT y en ese caso, como es lógico, devuelve *cierto* cuando no hay ninguna tupla resultante de la subconsulta, y *falso* cuando hay al menos una. NOT EXISTS *expresión_tabla* significa que está vacía la *expresión_tabla.*

Se presenta a continuación la consulta del Ejemplo 26, pero resuelta con el predicado EXISTS.

Ejemplo 30: Obtener (usando el predicado EXISTS) los dorsales y los nombres ciclistas cuya edad no coincide con la de ninguno de los ciclistas del equipo Kelme.

```
SELECT dorsal, nombre
FROM Ciclista C1
WHERE NOT EXISTS (SELECT *
                  FROM Ciclista C2
                  WHERE nomeq = 'Kelme' AND  C1.edad=C2.edad);
```

La subconsulta obtiene los ciclistas del equipo Kelme que tienen la misma edad que el ciclista que se está comparando, el actual. Si no hay ninguno, es decir, si la subconsulta está vacía, EXISTS se evalúa a *falso* y NOT EXISTS, por lo tanto, se evalúa a *cierto* y la tupla actual de *Ciclista* intervendrá en el resultado.

De manera semejante a la equivalencia que se comentó antes, se puede ver que una consulta que contenga la condición siguiente:

```
WHERE NOT EXISTS (SELECT * FROM …)
```

es equivalente a otra en la que la condición sea:

```
WHERE (SELECT COUNT (*) FROM …)=0
```

Aplicado al enunciado del Ejemplo 26, se obtendría el ejemplo siguiente:

Ejemplo 31: Obtener (sin usar los predicados IN o EXISTS) los dorsales y los nombres ciclistas cuya edad no coincide con la de ninguno de los ciclistas del equipo Kelme.

```
SELECT dorsal, nombre
FROM Ciclista C1
WHERE (SELECT COUNT(*)
       FROM Ciclista C2
       WHERE nomeq = 'Kelme' AND  C1.edad=C2.edad)=0;
```

2.3.4.4 Uso de EXISTS para cuantificación universal

Sea el enunciado siguiente: 'Obtener el nombre del ciclista que ha ganado todas las etapas de más de 200 km'.

Antes de resolver este enunciado se van a comentar algunos aspectos del mismo.

Algunas consultas como ésta, buscan una información que debe cumplirse para todas las filas de un conjunto. En este caso la condición para que un ciclista salga en el resultado es que haya ganado 'todas las etapas de más de 200 km'.

Cuando se encuentra una condición de este tipo, que afecta a todas las tuplas de un conjunto determinado, se dice que la consulta tiene *cuantificación universal*, por su semejanza con el significado del cuantificador ∀ (para todo) de la lógica.

Se ha estudiado antes que hay un predicado EXISTS en SQL, que es el equivalente al cuantificador existencial ∃ (existe), y sin embargo en las implementaciones de SQL no hay un predicado equivalente al cuantificador ∀. Sí que está definido en el estándar, pero no está disponible en Oracle. Esto es un problema, pero tiene solución usando una equivalencia lógica:

$$\forall X\ F(X) \equiv \neg\exists X\ \neg\ F(X)$$

Esta expresión significa que es equivalente la parte izquierda, que se lee: 'para todo X se cumple la propiedad F(X)', a la parte derecha de la expresión que significa: 'no existe un X para el que no se cumpla la propiedad F(X)'.

Así, las consultas que aparezcan como una cuantificación universal, se deben reformular para que queden expresadas como una doble negación, usando la equivalencia anterior.

Para ello, hay que localizar en el enunciado cuál es el conjunto X afectado por el cuantificador ∀, (para lo que puede orientar la palabra 'todas' en este caso, y otras similares en otros enunciados) y tener claro cuál es la propiedad que se debe cumplir (F(X)).

En el enunciado de más arriba,

- X = etapas de más de 200 km. (ya que la palabra *todas* se refiere a ellas) , y
- F(X) = que la haya ganado el ciclista actual

Así pues, para reformular la expresión se debe comprobar que no hay (¬∃) una etapa de más de 200 km que no haya ganado el ciclista actual ¬ F(X). Reformulado el enunciado quedaría así: 'Obtener el nombre del ciclista tal que *no existe* una etapa de más de 200 km. que él *no* haya ganado'. A continuación se resuelve este enunciado.

Ejemplo 32: Obtener el nombre del ciclista que ha ganado todas las etapas de más de 200 km. Se sabe que en esta base de datos hay etapas de más de 200 km.

```
SELECT nombre
FROM Ciclista C
WHERE NOT EXISTS                        /*¬∃*/
      (SELECT * FROM Etapa E
       WHERE km > 200 AND               /*X=etapas de más de 200 km*/
          NOT (C.dorsal = E.dorsal));  /*¬F(X)≅no la ha ganado el ciclista actual*/
```

Sería más sencillo poner C.dorsal <> E.dorsal en vez de NOT (C.dorsal = E.dorsal), pero se ha mantenido con NOT en este caso para mostrar las dos negaciones.

Hay que comentar que la puntualización que se hace en el enunciado del Ejemplo 32 acerca de que se sabe que hay etapas de más de 200 km es importante. En el Ejemplo 33 se muestra el mismo enunciado, con el cambio de la longitud a más de 300 km (no hay ninguna etapa de más de 300 km en la base de datos de *Ciclismo*); la subconsulta, que tiene una condición con dos comparaciones unidas por AND, se evaluaría a *falso* ya que la primera de las condiciones (km > 300) no sería cierta para ninguna tupla de *Etapa* y, por lo tanto, el resultado de la subconsulta estaría vacío. Y entonces, la condición NOT EXISTS aplicada a esa subconsulta sería cierta para todas las tuplas de *Ciclista*. Este resultado no es el esperado, pues implícitamente en el enunciado se espera que los ciclistas obtenidos (en realidad sólo se podría obtener uno como máximo) hayan ganado etapas, y no es así. Para asegurarse de que las consultas de cuantificación universal no tienen ese comportamiento inesperado, hay que comprobar que la condición que representa a X no esté vacía (si no se conoce previamente que no lo estará). Para ello, hay que añadir una condición, a la que llamaremos *coletilla* a la consulta, como se ve en la solución.

Ejemplo 33: Obtener el nombre del ciclista que ha ganado todas las etapas de más de 300 km. No se sabe si en esta base de datos hay o no etapas de más de 300 km.

```
SELECT nombre
FROM Ciclista C
WHERE NOT EXISTS (SELECT * FROM Etapa E
                  WHERE km > 300 AND NOT (C.dorsal = E.dorsal))
    AND EXISTS(SELECT * FROM Etapa E  WHERE km > 300);
```

Puede verse que la última condición, que está al mismo nivel que NOT EXISTS, comprueba que haya alguna etapa de más de 300 km., de manera que, si no la hay, como sucede en esta base de datos, la coletilla se evaluará a *falso* siempre, por lo que ninguna tupla de *Ciclista* cumplirá la condición de la cláusula WHERE, como se esperaba y el resultado estará vacío.

Hay otras posibilidades para resolver las consultas con cuantificación universal sin usar el predicado EXISTS. A continuación, se muestra una de ellas, que usa la función agregada COUNT. Más adelante, en la sección 2.6 se verá otra.

Ejemplo 34: Obtener (sin usar el predicado EXISTS) el nombre del ciclista que ha ganado todas las etapas de más de 200 km. Se sabe que en esta base de datos hay etapas de más de 200 km.

```
SELECT nombre
FROM Ciclista C
WHERE  (SELECT COUNT (*) FROM Etapa E WHERE C.dorsal = E.dorsal and km>200)
     =  (SELECT COUNT (*) FROM Etapa WHERE km > 200);
```

En esta solución se está comparando la cantidad de etapas de más de 200 km ganadas por el ciclista actual con la cantidad de etapas de más de 200 km que hay; si esas cantidades coinciden, significará que el

ciclista actual ha ganado todas las etapas de más de 200 km.

Si no se conoce con certeza que la cantidad vaya a ser distinta de 0, hay que comprobarlo con la coletilla, como se puede ver en el Ejemplo 35.

Ejemplo 35: Obtener (sin usar el predicado EXISTS) el nombre del ciclista que ha ganado todas las etapas de más de 300 km. No se sabe si en esta base de datos hay o no etapas de más de 300 km.

```
SELECT nombre
FROM Ciclista C
WHERE  (SELECT COUNT (*) FROM Etapa E WHERE C.dorsal = E.dorsal and km>300)
         = (SELECT COUNT (*) FROM Etapa E WHERE km > 300)
    AND 0 < (SELECT COUNT (*) FROM Etapa E WHERE km > 300);
```

2.4 Consultas complejas: Agrupación

Para ilustrar el uso de las consultas agrupadas, se usará un ejemplo sencillo:

Ejemplo 36: Obtener el nombre de los equipos cuyo nombre es menor que 'G', y la edad media de sus ciclistas.

En las consultas sencillas se explicó que la selección no podía incluir a la vez referencias a funciones agregadas o constantes y a atributos, y eso es justamente lo que solicita este enunciado. Así, con lo explicado hasta ahora no es posible definir una consulta para obtener dicha información. La solución a este ejercicio es la siguiente:

```
SELECT nomeq, AVG(edad) Media_edad
FROM Ciclista WHERE nomeq < 'G'
GROUP BY nomeq;
```

Se ha usado la cláusula GROUP BY, que significa 'agrupar por', cuyo efecto es el siguiente:

Después de que el SGBD seleccione de la tabla *Ciclista* aquellas filas que hacen cierta la condición (*nomeq* < 'G'), basándose en el nombre del equipo (*nomeq)*, que es el atributo que sigue a GROUP BY, el SGBD forma con esas filas tantos **grupos de filas** como valores distintos de *nomeq* menores que 'G' aparezcan en la tabla *Ciclista*. Cada grupo estará formado por todos los ciclistas de un mismo equipo. Y ya que para todos ellos el valor de *nomeq* es el mismo, puede devolverlo una sola vez, junto con la media de la edad de los ciclistas de dicho equipo.

En la Figura 9 se muestran una tabla con la información de los ciclistas que pertenecen a equipos de nombre menor que 'G' sombreados con distinto color los de cada equipo, lo que permite distinguir los grupos de filas formados por la cláusula GROUP BY. La segunda tabla que aparece corresponde a la respuesta del Ejemplo 36, donde se ve cada equipo en una fila, sombreado con el mismo color que en la otra tabla, junto con la media de edad de los ciclistas de cada equipo (la edad aparece con un sólo decimal, para lo cual se ha usado una función cuya explicación se sale del alcance de esta asignatura).

DORSAL	NOMBRE	EDAD	NOMEQ
74	Claudio Chioccioli	36	Amore Vita
65	Pascal Lino	29	Amore Vita
12	Alessio Di Basco	31	Amore Vita
90	Ivan Ivanov	27	Artiach
21	Erwin Nijboer	31	Artiach
87	Fernando Mota	32	Artiach
92	Federico Garcia	27	Artiach
45	Alberto Elli	26	Artiach
20	Alfonso Gutiérrez	29	Artiach
98	Eleuterio Anguita	25	Artiach
56	Andrew Hampsten	29	Banesto
31	Vicente Aparicio	30	Banesto
30	Melchor Mauri	28	Banesto
28	Jesus Montoya	33	Banesto
26	Mikel Zarrabeitia	27	Banesto
19	Julian Gorospe	34	Banesto
1	Miguel Induráin	32	Banesto
2	Pedro Delgado	35	Banesto
94	Marino Alonso	30	Banesto
71	Prudencio Induráin	27	Banesto
70	Sandro Heulot	29	Banesto
17	Bruno Leali	37	Bresciali-Refin
11	Flavio Giupponi	31	Bresciali-Refin
72	Stefano Colage	28	Bresciali-Refin
60	Giovanni Lombardi	28	Bresciali-Refin
85	Dimitri Abdoujaparov	30	Carrera
24	Claudio Chiappucci	29	Carrera
89	Stefan Roche	36	Carrera
69	Eddy Seigneur	27	Castorama
67	Armand de las Cuevas	28	Castorama
46	Agustin Sagasti	24	Euskadi
38	Javier Palacin	25	Euskadi

NOMEQ	MEDIA
Amore Vita	32.0
Artiach	28.1
Banesto	30.4
Bresciali-Refin	31.0
Carrera	31.7
Castorama	27.5
Euskadi	24.5

Figura 9: Ilustración del Ejemplo 36: en la tabla grande, las filas seleccionadas de *Ciclista* y en la tabla pequeña el resultado de la consulta.

Para que el SGBD permita que una consulta combine atributos (o expresiones) y funciones, la consulta debe estar siempre agrupada, es decir, debe incluir la cláusula GROUP BY. Los efectos de esta cláusula son varios:

1. Sólo pueden ponerse como objetivos detrás de la palabra SELECT aquellos atributos que se han usado para agrupar (los que aparecen en la cláusula GROUP BY), además de funciones sobre cualquier atributo, expresiones y constantes. Cualquier atributo que no esté detrás de GROUP BY puede tomar valores diferentes dentro de un mismo grupo, por lo que no se podría devolver una sola fila por grupo, y no se podría responder la consulta. Sin embargo, puede haber atributos por los que se agrupa, que no se incluyan en la consulta.
2. El SGBD considera tantos grupos en la tabla como valores distintos tomen el conjunto de atributos por los que se agrupa.
3. Todas las filas que tengan el mismo valor en dichos atributos formarán un grupo.
4. Como se ha visto en el Ejemplo 36, las funciones agregadas tienen un efecto diferente cuando la consulta es agrupada, ya que se aplican a cada grupo formado, y devuelven tantos valores como grupos haya.
5. Se puede añadir la cláusula HAVING, que sólo está permitida en consultas agrupadas, y que va seguida de una condición semejante a la condición de la cláusula WHERE; la condición se aplicará a los grupos

formados, de manera que sólo se formará el resultado de la consulta a partir de los grupos que hagan cierta la condición.

6. En la condición de la cláusula HAVING se pueden usar, como operandos, elementos del mismo tipo que los indicados en el apartado 1, es decir propiedades de los grupos formados, y se pueden usar subconsultas con los predicados que las admiten. Hay que resaltar que, en las consultas agrupadas, las funciones agregadas se pueden usar también para construir comparaciones, ya que se pueden ver como 'atributos' de los grupos (véase el ejemplo 38). Recuérdese que este uso está prohibido en la cláusula WHERE.

7. Las consultas agrupadas permiten anidar, a un solo nivel, las funciones agregadas, algo que tampoco estaba permitido en las consultas sin agrupación (véase el ejemplo 39). La explicación es sencilla: debido a que las funciones devuelven un valor por cada grupo, tiene sentido obtener el máximo, el mínimo, la suma o la cuenta de esos valores. Sin embargo, este anidamiento devolverá una sola fila por lo que no se puede combinar en la misma consulta con otros objetivos que puedan devolver varias filas.

Así pues, siempre que una consulta requiera el valor de uno o más atributos y el de alguna función agregada, necesariamente la consulta tiene que ser agrupada, y entre los atributos por los que se agrupa deben aparecer los requeridos en la consulta.

La sintaxis de la cláusula GROUP BY dentro de una consulta es la siguiente (el orden de ejecución de las cláusulas es el indicado por los números del principio de la línea):

```
6 SELECT [ALL | DISTINCT] {expresión1, expresión2,…,  expresiónn | *}
1 FROM referencia_tabla1, referencia_tabla2,…,  referencia_tablan
2 [WHERE condición]
3 [GROUP BY columna1, columna2,…, columnan
4    [HAVING condición]]
5 [ORDER BY columna1, columna2,…, columnan ]
```

Así, después de acceder a las tablas, el SGBD selecciona las filas que hacen cierta la condición, luego las agrupa por los atributos indicados, a continuación, aplica la condición que sigue a HAVING para quedarse sólo con los grupos que la hacen cierta, posteriormente aplica las funciones agregadas, extrae los objetivos y por último ordena el resultado según los atributos o funciones indicados. Obviamente, las cláusulas opcionales pueden faltar.

A continuación se ilustra con un ejemplo incorrecto un error muy común que se debe evitar.

Ejemplo 37: Sea la consulta incorrecta:

```
SELECT nomeq, nombre, AVG(edad)
FROM Ciclista
GROUP BY nomeq;
```

Esta consulta no se puede resolver, porque no tiene sentido, ya que mezcla los nombres de ciclistas de un equipo, que normalmente serán varias filas por cada equipo, con el nombre del equipo y la media de edad del equipo, que devuelven una sola fila. Daría un error de sintaxis en el que el SGBD informaría de que hay valores que no son únicos para los grupos.

De una consulta agrupada sólo se puede realizar una selección que devuelva una fila por cada grupo formado en la consulta.

Ejemplo 38: Obtener, de las etapas que tienen dos o más puertos de más de 1000 m. de altura, el número de la etapa y la media de la pendiente de dichos puertos.

```
SELECT netapa, AVG(pendiente)
FROM Puerto
WHERE altura >1000
GROUP BY netapa
HAVING COUNT (*)>1;
```

Nótese que se seleccionan sólo las filas de *Puerto* de altura mayor que 1000 m. Después de agrupar por *netapa*, en cada grupo hay tantas filas como puertos de más de 1000 m. haya en cada una de esas etapas, y sólo se usarán para el resultado los grupos que tengan más de una fila. Obsérvese el uso en la cláusula

HAVING de una comparación con una función, COUNT(*).

Ejemplo 39: Obtener cuál es la edad máxima de las edades mínimas de cada equipo.

```
SELECT MAX(MIN(edad)) "Máximo de las edades mínimas"
FROM ciclista
GROUP BY nomeq;
```

El resultado será una fila única que representará el valor máximo de todas las edades mínimas de los equipos.

A veces, los datos que interesan en una consulta son insuficientes para agrupar correctamente las tuplas, y hace falta añadir más atributos de agrupación. Esto ocurre cuando los atributos de la selección pueden aparecer repetidos en la tabla original. En ese caso, hacer grupos por esos atributos sólo, puede producir resultados inesperados y erróneos. En el siguiente ejemplo se ilustra uno de estos casos.

Ejemplo 40: Obtener el nombre de los ciclistas y el número de puertos que ha ganado cada uno, siendo la media de la pendiente de éstos superior a 6.

```
SELECT nombre, COUNT (nompuerto)
FROM Ciclista C, Puerto P
WHERE C.dorsal = P.dorsal
GROUP BY C.dorsal, nombre
HAVING AVG(pendiente)>6
```

Esta consulta pide el nombre y cantidad de puertos ganados; del ciclista interesa sólo el nombre y sin embargo, hay que agrupar por nombre y por dorsal (da igual *C.dorsal* o *P.dorsal* ya que son iguales). Si se agrupara sólo por nombre y hubiera dos ciclistas en la tabla con el mismo nombre, contaría juntos los puertos que hubieran ganado los dos ciclistas, lo que sería erróneo. Al agrupar también por dorsal se garantiza que cada grupo formado corresponde a un ciclista distinto.

2.5 Consultas complejas: Concatenación

El operador concatenación (JOIN en SQL) combina dos tablas utilizando diferentes formas, que son variantes del operador ⊗ del Álgebra Relacional. Este operador se incluye en la cláusula FROM, lo que puede interpretarse como que la consulta se hace a la tabla resultante de la concatenación. Las formas de este operador que se van a presentar son las siguientes:

- Concatenación interna.
- Concatenación externa.
- Producto cartesiano.

2.5.1 Concatenación interna (INNER JOIN)

El operador INNER JOIN es muy utilizado en las consultas a bases de datos. Permite combinar la información de dos tablas basándose en el valor de unos atributos determinados en ambas tablas. Se puede usar eligiendo una de las tres opciones que permite, y que se presentan primero con ejemplos. La palabra reservada INNER es el valor por defecto y se suele omitir.

El resultado del operador INNER JOIN es una tabla que como esquema tendrá la unión de los esquemas de las tablas concatenadas, y su extensión sólo contendrá filas formadas por una fila de cada tabla según la condición de concatenación.

A continuación, se presenta un ejemplo resuelto de dos maneras:

Ejemplo 41: Obtener el nombre de los ciclistas que han ganado al menos una etapa

a) Solución sin concatenación:

```
SELECT DISTINCT nombre
FROM Ciclista C, Etapa E
WHERE C.dorsal=E.dorsal;
```

b) Solución concatenando:

```
SELECT DISTINCT nombre
FROM Ciclista C NATURAL JOIN Etapa E;
```

Estas dos soluciones son equivalentes: a) con lista de tablas en la cláusula FROM relacionadas por la condición del WHERE, y b) con el operador JOIN en la cláusula FROM.

En el Ejemplo 41, de las opciones posibles del operador, la utilizada es NATURAL JOIN, que concatena las tablas basándose en la igualdad del valor de todos los atributos que son comunes (tienen el mismo nombre) a ambas tablas. Es el equivalente del operador ⊗ del Álgebra Relacional.

El uso de la opción NATURAL JOIN tiene un riesgo: si con el tiempo se cambia el esquema de la base de datos, añadiendo algún atributo en algunas de las tablas implicadas, el resultado de las consultas que usen este operador (que se hubieran definido y guardado con anterioridad a dicho cambio) puede verse modificado, ya que pueden coincidir los nombres de más atributos, y obtener resultados diferentes a los que se pretendió inicialmente.

Ejemplo 42: Obtener el nombre de los ciclistas que han ganado al menos una etapa y que tienen menos de 27 años.

a) Solución sin concatenación:

```
SELECT DISTINCT nombre
FROM Ciclista C, Etapa E
WHERE C.dorsal=E.dorsal AND edad < 27;
```

b) Solución concatenando:

```
SELECT DISTINCT nombre
FROM Ciclista C NATURAL JOIN Etapa E
WHERE edad <27;
```

Como puede verse, de las dos condiciones que aparecían en la cláusula WHERE, la que se refiere a la concatenación de las tablas se representa con NATURAL JOIN, y la que impone cualquier otra condición a las filas, debe ir en la cláusula WHERE.

Las cláusulas restantes de la instrucción SELECT con este operador son iguales a las ya vistas.

Otra de las opciones del operador JOIN es: USING *(columna$_1$, columna$_2$,…, columna$_n$),* que concatena las tablas basándose en la igualdad del valor de los atributos incluidos entre paréntesis y que deben ser comunes a ambas tablas. Con este operador se eliminan parte de los riesgos comentados para el NATURAL JOIN, ya que siempre usará los atributos especificados, pero conserva la limitación de que en las dos tablas los atributos se tienen que llamar igual para poder concatenarse.

Ejemplo 43: Obtener los nombres de los puertos, el número de la etapa en la que están y la longitud de la etapa, para los puertos de altura superior a 800.

```
SELECT nompuerto, netapa, km
FROM Puerto JOIN Etapa USING (netapa)
WHERE altura>800;
```

Sería incorrecta la solución siguiente:

```
SELECT nompuerto, netapa, km
FROM Puerto NATURAL JOIN Etapa
WHERE P.altura>800;
```

La razón por la que sería incorrecta la segunda solución es que concatenaría también por *dorsal* pues existe ese atributo en las dos tablas; en ese caso devolvería una respuesta que cumpliría también la condición de que “el puerto haya sido ganado por el mismo ciclista que la etapa”, y esta condición no estaba en el enunciado.

En las dos opciones presentadas, la tabla resultante no tiene repetidos los atributos comunes de las dos tablas que se combinan, por lo que no hay que cualificarlos con el nombre de la correspondiente tabla. Puede verse en el Ejemplo 43 que el atributo *netapa* no se cualifica.

La tercera opción es JOIN ON *condición*, que permite concatenar las tablas basándose en que la *condición* sea cierta. Esta alternativa del operador no tiene ninguno de los riesgos comentados para el NATURAL JOIN y el USING *(columna$_1$, columna$_2$,…, columna$_n$)*, ya que siempre usará los atributos especificados basándose en la condición. Como puede verse, la condición puede usar operadores distintos al de igualdad. En cuanto a los nombres de atributos que sean comunes a las dos tablas, sí que hay que cualificarlos, tal como se hacía en las consultas sin JOIN. Esta opción es la más versátil y no presenta problemas como las dos anteriores por lo que se recomienda su uso.

Ejemplo 44: Obtener los nombres de los puertos y el número de la etapa en la que están, si hay alguna etapa posterior que tiene más de 200 km.

```
SELECT DISTINCT nompuerto, P.netapa
FROM Puerto P JOIN Etapa E ON P.netapa<E.netapa
WHERE E.km>200;
```

La sintaxis de la concatenación interna es la siguiente:

```
referencia_tabla [NATURAL] [INNER] JOIN referencia_tabla
  [ON condición | USING (columna1, columna2,…, columnan)]
```

en la que necesariamente se debe elegir una de entre las tres opciones: NATURAL, ON condición o USING (*columna$_1$, columna$_2$,…, columna$_n$*).

Debe recordarse que el operador JOIN se usa tras la palabra FROM, no en la cláusula WHERE.

El uso del operador de concatenación cuando se combinan más de dos tablas debe ser especialmente cuidadoso, ya que las condiciones deben ponerse intercaladas para cada combinación, de manera que el operador JOIN actúe siempre sobre dos tablas, pudiendo ser una de ellas el resultado de combinar otras. Se aconseja el uso de paréntesis para referirse a una tabla resultante de una combinación, pues, aunque no es obligatorio, mejora su comprensión.

Ejemplo 45: Obtener el dorsal y el nombre de los ciclistas que han ganado al menos una etapa que tenga algún puerto.

```
SELECT DISTINCT C.dorsal, C.nombre
FROM (Ciclista C JOIN Etapa E ON C.dorsal=E.dorsal)
                 JOIN Puerto P ON P.netapa=E.netapa;
```

También sería correcta la siguiente instrucción, en la que se ha cambiado la posición de las condiciones:

```
SELECT DISTINCT C.dorsal, C.nombre
FROM Ciclista C JOIN (Etapa E JOIN Puerto P ON P.netapa=E.netapa)
                      ON C.dorsal=E.dorsal;
```

Sin embargo, sería incorrecta la siguiente instrucción:

```
SELECT DISTINCT C.dorsal, C.nombre, e.netapa
FROM Ciclista C JOIN Etapa E JOIN Puerto P
        ON C.dorsal=E.dorsal ON P.netapa=E.netapa;
```

Debido a que la condición se aplica a las tablas que la preceden inmediatamente. En el último caso la condición *C.dorsal=E.dorsal* usa el atributo *C.dorsal* que no aparece en ninguna de las dos tablas de su izquierda.

Así mismo, pueden combinarse de la manera siguiente:

```
SELECT DISTINCT C.dorsal, C.nombre, E.netapa
FROM (Ciclista C JOIN Etapa E ON C.dorsal=E.dorsal), Puerto P
WHERE E.netapa= P.netapa;
```

Esta última forma no es aconsejable pues introduce confusión el hecho de que una condición de concatenación aparezca en la cláusula FROM y la otra aparezca como una condición en la cláusula WHERE.

2.5.2 Concatenación externa (OUTER JOIN)

La operación de concatenación externa permite que, cuando se combinen dos tablas en las que una o

más filas de una o ambas tablas no cumplan la condición de concatenación, el resultado final incluya también dichas filas, completando con valores nulos todos los atributos correspondientes a la otra u otras tablas.

Ejemplo 46: Obtener, para cada ciclista que hay en la base de datos, su dorsal, su nombre, el código de cada maillot que ha llevado y el número de etapa en la que lo ha llevado.

```
SELECT C.dorsal, nombre, codigo, netapa
FROM Ciclista C LEFT JOIN Llevar L ON  C.dorsal = L.dorsal;
```

DORSAL	NOMBRE	CODIGO	NETAPA
...			
1	Miguel Induráin	MGE	17
1	Miguel Induráin	MGE	18
2	Pedro Delgado	MGE	7
2	Pedro Delgado	MGE	6
2	Pedro Delgado	MGE	5
2	Pedro Delgado	MMO	21
3	Alex Zulle	MGE	11
3	Alex Zulle	MGE	12
4	Tony Rominger	MGE	8
5	Gert-Jan Theunisse	(null)	(null)
6	Adriano Baffi	(null)	(null)
7	Massimiliano Lelli	(null)	(null)
8	Jean Van Poppel	MSE	2
8	Jean Van Poppel	MSE	4
9	Massimo Podenzana	(null)	(null)
10	Mario Cipollini	MSE	18
11	Flavio Giupponi	(null)	(null)
12	Alessio Di Basco	MSE	3
12	Alessio Di Basco	MSE	5
12	Alessio Di Basco	MSE	6
13	Lale Cubino	(null)	(null)
14	Roberto Pagnin	(null)	(null)

Figura 10: Resultado (parcial) de ejecutar la instrucción Ejemplo 46

En este ejemplo, en cuanto a la sintaxis, la única diferencia con una concatenación interna es la palabra LEFT, que precede a JOIN, y que significa que se quiere que, de la tabla que está a la izquierda (LEFT en inglés) del operador JOIN, que en este caso es la tabla *Ciclista*, aparezcan todas sus filas.

Recuérdese que se desea obtener información de todos los ciclistas, no sólo de los que han llevado algún maillot; es decir, se desea que en la respuesta aparezcan todas las filas de la tabla *Ciclista,* aunque muchos ciclistas no hayan llevado maillots. Se hace necesario usar concatenación externa para que se preserven todas las filas de la tabla *Ciclista*.

Puede verse que un ciclista que haya llevado al menos un maillot, aparece tantas veces en el resultado como veces aparezca su dorsal en la tabla *Llevar*, pero si no ha llevado ningún maillot, también aparece en el resultado, pero una sola vez, y los atributos de la tabla *Llevar* son nulos, lógicamente.

Ejemplo 47: Obtener, para cada ciclista que hay en la base de datos, su dorsal, su nombre y el número de veces que ha llevado algún maillot.

```
SELECT C.dorsal, nombre, COUNT(L.dorsal)
FROM Ciclista C LEFT JOIN Llevar L ON  C.dorsal = L.dorsal
GROUP BY C.dorsal, nombre;
```

Sería igual de correcta la solución siguiente:

```
SELECT C.dorsal, nombre, COUNT(L.dorsal)
FROM Llevar L RIGHT JOIN Ciclista C ON C.dorsal = L.dorsal
GROUP BY C.dorsal, nombre;
```

Nótese que en este caso se ha cambiado el orden de aparición de las tablas, por lo que la tabla *Ciclista* queda a la derecha (RIGHT en inglés) del operador JOIN.

En este ejemplo también es importante notar que el resultado habría sido distinto si en la función COUNT hubiéramos usado el atributo *C.dorsal*, ya que *C.dorsal* no es nulo en ningún caso, y la función nos devolvería un 1 para los ciclistas que no hubieran llevado ningún maillot, lo que no sería correcto. Al usar *L.dorsal*, puesto que las funciones agregadas descartan los valores nulos, nos devuelve el resultado esperado.

La sintaxis de la concatenación externa es la siguiente:

```
referencia_tabla [NATURAL] {LEFT  | RIGHT | FULL } [OUTER]
     JOIN referencia_tabla
          [ON condición | USING (columna1, columna2,…, columnan)]
```

Como puede verse, también permite las tres formas de combinación, aunque la más recomendable, como se ha dicho antes, es ON condición.

Las diferencias de sintaxis que presenta la concatenación externa respecto a la interna son:

- la palabra OUTER en vez de INNER (OUTER también es opcional y suele omitirse).
- la inclusión obligatoria delante de JOIN (si se usa OUTER, delante de ésta) de una de las tres palabras LEFT, RIGHT o FULL. Se ha explicado el significado de las dos primeras; la última - FULL - significa que se quieren preservar todas las filas de las dos tablas que se combinan.

2.5.3 Producto cartesiano (CROSS JOIN)

Esta variante devuelve el producto cartesiano de dos tablas, es decir, es equivalente a separar dos tablas por una coma.

2.6 Consultas conjuntistas

Existen distintas formas de combinar varias tablas en consultas que dan lugar a una “expresión de tabla”. Algunas se han visto ya, y otras se nombran en esta sección por primera vez.

Las formas de combinar dos o más tablas en el lenguaje SQL son:

1. Inclusión de varias tablas en la cláusula FROM.
2. Uso de subconsultas en las condiciones de las cláusulas WHERE o HAVING.
3. Concatenaciones de tablas: combinan dos tablas utilizando diferentes formas variantes del operador concatenación (⊗) del Álgebra Relacional.
4. Combinaciones conjuntistas de tablas: utilizan operadores de la teoría de conjuntos para combinar filas de varias tablas.

Las formas 1, 2 y 3 ya se han explicado; en esta sección se describe la forma 4 que utiliza operadores de la teoría de conjuntos. Algunas consultas se pueden responder indistintamente con diferentes formas, pero hay algún tipo de consultas que requiere alguno de los operadores bien conjuntistas o bien de concatenación, como se verá más adelante.

Los operadores conjuntistas son:

- el de la unión de conjuntos (UNION),
- el de la intersección de conjuntos (INTERSECT) y
- el de la diferencia de conjuntos (MINUS en ORACLE, aunque en SQL estándar se llama EXCEPT).

Todos ellos actúan sobre dos operandos que deben ser expresiones de tablas (es decir, consultas SELECT) con esquemas compatibles (idénticos en número de atributos, nombre de los mismos y tipo de datos). Este requisito es lógico, ya que dicho esquema común será también el esquema de la tabla resultante. Se aceptan nombres de atributos distintos, realizando la unión por orden de atributos, aunque el resultado es confuso ya que en el esquema resultante sólo puede usar los nombres de una de las tablas, la primera, por lo que se usará esta posibilidad sólo cuando sea necesario. También pueden ser operandos de los operadores conjuntistas términos formados con estos mismos operadores aplicados sobre otras expresiones, también con esquemas compatibles.

Todos los operadores conjuntistas en SQL eliminan valores duplicados excepto en un caso que se explica en la sección siguiente.

2.6.1 Operador UNION

Este operador representa la unión de conjuntos, siendo cada conjunto una tabla determinada por una consulta SELECT (o el resultado de otra operación conjuntista). Obtiene una tabla que se forma con la unión de las filas de las tablas provenientes de las dos expresiones.

Ejemplo 48: Obtener los dorsales de los ciclistas que han llevado un maillot o han ganado un puerto.

```
SELECT dorsal FROM Llevar
    UNION
SELECT dorsal FROM Puerto;
```

Se obtiene como resultado una tabla con un atributo, *dorsal*, y con tantas filas como dorsales distintos aparezcan en cualquiera de las dos tablas (en total, 25 dorsales).

La sintaxis es la siguiente:

```
expresión_tabla UNION [ALL] término_tabla
```

La palabra ALL significa aquí también (como cuando va a continuación de SELECT), que aparecerán filas repetidas si algún valor se encuentra en más de una expresión de tabla.

Ejemplo 49: Obtener todos los dorsales de los ciclistas que han llevado un maillot o han ganado un puerto o una etapa, apareciendo cada uno tantas veces como número de esos logros haya conseguido.

```
SELECT dorsal FROM Llevar
    UNION ALL
SELECT dorsal FROM Puerto
    UNION ALL
SELECT dorsal FROM Etapa;
```

El uso de la opción ALL en esta consulta hace que aparezcan repeticiones.

Este operador permite también resolver consultas que se han resuelto con la concatenación externa.

Ejemplo 50: Obtener para cada ciclista que hay en la base de datos su dorsal, su nombre y el número de veces que ha llevado algún maillot.

a) Solución incorrecta:

```
SELECT C.dorsal, nombre, COUNT(C.dorsal)
FROM Ciclista C, Llevar L
WHERE C.dorsal=L.dorsal
GROUP BY C.dorsal, nombre;
```

b) Solución correcta con concatenación externa:

```
SELECT C.dorsal, nombre, COUNT(L.dorsal)
FROM Ciclista C LEFT JOIN Llevar L ON  C.dorsal = L.dorsal
GROUP BY C.dorsal, nombre;
```

c) Solución correcta con la unión:

```
SELECT C.dorsal, nombre, COUNT(C.dorsal)
FROM Ciclista C, Llevar L
WHERE C.dorsal=L.dorsal
GROUP BY C.dorsal, nombre
             UNION
SELECT C.dorsal, nombre, 0
FROM Ciclista C
WHERE C.dorsal NOT IN (SELECT dorsal FROM Llevar);
```

Si se resolviera como en la solución a), sólo se obtendría la información de aquellos ciclistas que han llevado alguna vez un maillot, por lo que sería incorrecta, ya que la consulta pide todos los ciclistas de la base de datos. Por lo tanto, al conjunto de ciclistas obtenidos por la solución a), hay que añadirles los ciclistas que

no han llevado ningún maillot, para quienes el número de veces será 0. Una forma de añadirlos es con la solución b) usando la concatenación externa o el operador UNION, como se muestra en c).

Obsérvese en la solución c) que el esquema de las dos expresiones es idéntico, pues el 0 que se ha puesto como tercer objetivo de la consulta es un entero, como el resultado de la función COUNT, por lo que se puede realizar la unión sin problemas.

2.6.2 Operador INTERSECT

Este operador representa la intersección de conjuntos, por lo que obtendrá aquellas filas que se encuentren en las dos expresiones.

Ejemplo 51: Obtener los dorsales de los ciclistas que han ganado algún puerto y alguna etapa.

```
SELECT dorsal FROM Puerto
    INTERSECT
SELECT dorsal FROM Etapa;
```

La sintaxis del operador es:

```
expresión_tabla INTERSECT término_tabla.
```

2.6.3 Operador MINUS

Este operador representa la diferencia de conjuntos, es decir, devuelve aquellas filas de la primera expresión que no se encuentran en la segunda expresión. Es útil para expresar de forma más sencilla alguna consulta, como la que se resuelve en el ejemplo siguiente:

Ejemplo 52: Obtener los números de las etapas que no tienen puertos.

a) Solución con subconsultas:

```
SELECT netapa
FROM Etapa
WHERE netapa NOT IN (SELECT netapa FROM Puerto);
```

b) Solución con MINUS:

```
SELECT netapa
FROM Etapa
  MINUS
SELECT netapa
FROM Puerto;
```

Las soluciones a) y b) son equivalentes.

La sintaxis del operador es:

```
expresión_tabla MINUS término_tabla.
```

A continuación se puede ver otra forma de resolver el Ejemplo 32, que incluía cuantificación universal, usando el operador MINUS.

Ejemplo 53: Obtener (usando el operador MINUS) el nombre del ciclista que ha ganado todas las etapas de más de 200 km. Se sabe que en esta base de datos hay etapas de más de 200 km.

```
SELECT nombre
FROM Ciclista C
WHERE  NOT EXISTS
       (SELECT E.netapa FROM Etapa E WHERE km > 200
                      MINUS
        SELECT F.netapa FROM Etapa F WHERE km > 200 AND C.dorsal=F.dorsal);
```

Esta consulta obtiene el nombre del ciclista tal que si a las etapas de más de 200 km se le restan las etapas de más de 200 km que él ha ganado no queda ninguna, ya que eso quiere decir que ese ciclista las ha ganado todas.

3 INSTRUCCIÓN INSERT

Esta instrucción permite añadir una o más filas a una tabla.

Tiene diferentes opciones que se muestran en los ejemplos que siguen.

Ejemplo 54: Añadir un ciclista de dorsal 101, de nombre 'Joan Peris', del equipo 'Kelme' y de 27 años.

```
INSERT INTO Ciclista
VALUES (101, 'Joan Peris', 27, 'Kelme');
```

Esta instrucción añade una fila completa a la tabla *Ciclista*, en la que todos sus atributos tienen asignado un valor no nulo.

DORSAL	NOMBRE	EDAD	NOMEQ
1	Miguel Induráin	32	Banesto
2	Pedro Delgado	35	Banesto
…	…	…	…
99	Per Pedersen	29	Seguros Amaya
100	William Palacios	30	Jolly Club
101	Joan Peris	27	Kelme

Figura 11: Tabla *Ciclista* después de la inserción del ejemplo 54

Si se desconoce el valor de algún atributo, la instrucción debe incluir entre paréntesis el nombre de los atributos a los que se les va a asignar valor, y en el mismo orden en que van a introducir dichos valores.

Ejemplo 55: Añadir un ciclista de dorsal 101, nombre 'Joan Peris', y del equipo 'Kelme' (no se sabe la edad).

```
INSERT INTO Ciclista (dorsal, nomeq, nombre)
VALUES (101, 'Kelme', 'Joan Peris');
```

DORSAL	NOMBRE	EDAD	NOMEQ
1	Miguel Induráin	32	Banesto
2	Pedro Delgado	35	Banesto
…	…	…	…
99	Per Pedersen	29	Seguros Amaya
100	William Palacios	30	Jolly Club
101	Joan Peris		Kelme

Figura 12: Tabla *Ciclista* después de la inserción del **Ejemplo 55**.

También se pueden insertar muchas filas de una vez si se obtienen a través de una consulta SELECT que se escribe después del nombre de la tabla donde se va a insertar.

Ejemplo 56: Sea la tabla *Ciclista_ganador*, que tiene el mismo esquema que *Ciclista*. Añadir a dicha tabla toda la información de los ciclistas que hayan ganado alguna etapa.

```
INSERT INTO Ciclista_ganador
SELECT * FROM Ciclista
WHERE dorsal IN (SELECT dorsal FROM etapa);
```

Todas las filas obtenidas con la consulta SELECT se añadirían con esta operación a la tabla *Ciclista_ganador*.

También pueden asignarse a una fila valores por defecto contenidos en la definición de la tabla.

La sintaxis de la instrucción INSERT es la siguiente:

```
INSERT INTO tabla [(columna1, columna2,…, columnan)]
{DEFAULT VALUES | VALUES(átomo1, átomo2,…, átomon) | expresión_tabla}
```

4 INSTRUCCIÓN DELETE

Esta instrucción permite eliminar una o más filas completas de una tabla. En su forma más simple eliminaría todas las filas de la tabla, según se ve en el ejemplo siguiente:

Ejemplo 57: Borrar todos los ciclistas de la tabla *Ciclista*.

```
DELETE FROM Ciclista;
```

Dejaría la tabla vacía, sin una sola fila, pero la tabla seguiría existiendo.

Normalmente se desea eliminar sólo una o varias filas, pero no todas. Para hacerlo hay que incluir en la instrucción una cláusula WHERE, con la misma sintaxis que se ha visto en la instrucción SELECT. Aquellas filas de la tabla que hagan cierta la condición se borrarán.

Ejemplo 58: Borrar los ciclistas menores de 20 años.

```
DELETE FROM Ciclista
WHERE edad < 20;
```

Eliminaría toda la información de los ciclistas que sean menores de 20 años, si hay alguno.

La sintaxis de la instrucción es:

```
DELETE FROM tabla [WHERE condición]
```

5 INSTRUCCIÓN UPDATE

Esta instrucción permite modificar el valor de uno o más atributos en una o más filas de una tabla.

Ejemplo 59: Incrementar en uno la edad de los ciclistas, porque ha pasado un año desde que se introdujo.

```
UPDATE Ciclista SET edad = edad + 1;
```

Ejemplo 60: Modificar el equipo Banesto, ya que se ha quedado sin director.

```
UPDATE Equipo SET director = NULL
WHERE nomeq = 'Banesto';
```

La sintaxis de la instrucción es la siguiente:

```
UPDATE tabla SET asignación1, asignación2,…, asignaciónn [WHERE condición]
```

donde una *asignación* es de la forma:

```
columna = {DEFAULT | NULL | expresión_escalar}
```

6 INSTRUCCIONES DE CONTROL DE TRANSACCIONES

En la última sección del documento UD1.2 se vio el concepto de transacción como un conjunto de operaciones que, o se ejecutan todas o no se ejecuta ninguna.

En SQL se inicia una transacción cada vez que se comienza una sesión de acceso a la base de datos. También se inicia una transacción cada vez que se da por finalizada otra.

Para indicar el fin de una transacción se pueden usar dos instrucciones distintas:

- COMMIT (confirmar).
- ROLLBACK (volver atrás).

Con la instrucción COMMIT se indica al SGBD dónde acaba una transacción y, al mismo tiempo, que se desea que los cambios que ha hecho la transacción se consideren aceptados, es decir, que el usuario desea que se trasladen a la base de datos los cambios indicados por las operaciones de la transacción. Después de acabar dicha transacción, el SGBD comprobará si dichos cambios cumplen todas las restricciones de integridad y, en ese caso, los cambios serán definitivos. En caso de que se viole alguna restricción, el SGBD dejará la base de datos como estaba antes de que se hubiera iniciado la transacción.

Con la instrucción ROLLBACK se indica al SGBD dónde acaba una transacción y, al mismo tiempo, que se desea que los cambios que ha hecho la transacción se descarten, es decir, que el usuario desea que no se hagan los cambios indicados por las operaciones de la transacción. Por lo tanto, cualquier cambio que se hubiera hecho durante la transacción no será trasladado a la base de datos, que quedará como estaba antes de que se hubiera iniciado la transacción.