

# SQL (II).

## Consultas

### (1.1) consultas de datos con SQL. DQL

#### (1.1.1) capacidades

DQL es la abreviatura del *Data Query Language* (lenguaje de consulta de datos) de SQL. El único comando que pertenece a este lenguaje es el versátil comando **SELECT**. Este comando permite:

- Obtener datos de ciertas columnas de una tabla (**proyección**)
- Obtener registros (filas) de una tabla de acuerdo con ciertos criterios (**selección**)
- Mezclar datos de tablas diferentes (**asociación, join**)
- Realizar cálculos sobre los datos
- Agrupar datos

#### (4.1.2) sintaxis sencilla del comando SELECT

```
SELECT * | {[DISTINCT] columna | expresión [[AS] alias], ...}  
FROM tabla;
```

Donde:

- \*. El asterisco significa que se seleccionan todas las columnas
- **DISTINCT**. Hace que no se muestren los valores duplicados.
- **columna**. Es el nombre de una columna de la tabla que se desea mostrar
- **expresión**. Una expresión válida SQL
- **alias**. Es un nombre que se le da a la cabecera de la columna en el resultado de esta instrucción.

Ejemplos:

```
/* Selección de todos los registros de la tabla clientes */  
SELECT * FROM Clientes;  
/* Selección de algunos campos */  
SELECT nombre, apellido1, apellido2 FROM Clientes;
```

## (1.2) cálculos

### (1.2.1) aritméticos

Los operadores + (suma), - (resta), \* (multiplicación) y / (división), se pueden utilizar para hacer cálculos en las consultas. Cuando se utilizan como expresión en una consulta SELECT, no modifican los datos originales, sino que como resultado de la vista generada por SELECT, aparece una nueva columna. Ejemplo:

```
SELECT nombre, precio, precio*1.16 FROM articulos;
```

Esa consulta obtiene tres columnas. La tercera tendrá como nombre la expresión utilizada, para poner un alias basta utilizar dicho alias tras la expresión:

```
SELECT nombre, precio, precio*1.16 AS precio_con_iva  
FROM articulos;
```

La prioridad de esos operadores es la normal: tienen más prioridad la multiplicación y división, después la suma y la resta. En caso de igualdad de prioridad, se realiza primero la operación que esté más a la izquierda. Como es lógico se puede evitar cumplir esa prioridad usando paréntesis; el interior de los paréntesis es lo que se ejecuta primero.

Cuando una expresión aritmética se calcula sobre valores **NULL**, el resultado es el propio valor **NULL**.

Se puede utilizar cualquiera de los operadores aritméticos: suma (+), resta (-), multiplicación (\*), división (/).

### (1.2.2) concatenación de textos

Todas las bases de datos incluyen algún operador para encadenar textos. En **SQLSERVER** es el signo + en Oracle son los signos ||. Ejemplo (Oracle o MySQL):

```
SELECT tipo, modelo, tipo || '-' || modelo "Clave Pieza"  
FROM piezas;
```

El resultado sería:

TIPO	MODELO	Clave Pieza
AR	6	AR-6
AR	7	AR-7
AR	8	AR-8
AR	9	AR-9
AR	12	AR-12
AR	15	AR-15
AR	20	AR-20
AR	21	AR-21
BI	10	BI-10
BI	20	BI-20
BI	22	BI-22
BI	24	BI-24

En la mayoría de bases de datos, la función **CONCAT** (SELECT concat(first\_name, last\_name) as nombrecompleto from tabla;) realiza la misma función.

## (1.3) condiciones

Se pueden realizar consultas que restrinjan los datos de salida de las tablas. Para ello se utiliza la cláusula **WHERE**. Esta cláusula permite colocar una condición que han de cumplir todos los registros, los que no la cumplan no aparecen en el resultado.

Ejemplo:

```
SELECT Tipo, Modelo FROM Pieza WHERE Precio>3;
```

### (1.3.1) operadores de comparación

Se pueden utilizar en la cláusula **WHERE**, son:

Operador	Significado
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
=	Igual
<>	Distinto



Operador	Significado
<b>!=</b>	<b>Distinto</b>

Se pueden utilizar tanto para comparar números como para comparar textos y fechas. En el caso de los textos, las comparaciones se hacen en orden alfabético. Sólo que es un orden alfabético estricto. Es decir, el orden de los caracteres en la tabla de códigos.

En muchas bases de datos hay problemas con la Ñ y otros símbolos nacionales (en especial al ordenar o comparar con el signo de mayor o menor, ya que la el orden ASCII no respeta el orden de cada alfabeto nacional). No obstante, es un problema que tiende a arreglarse en la actualidad en todos los SGBD (en Oracle no existe problema alguno) especialmente si son compatibles con Unicode.

### (1.3.2) valores lógicos

Son:

Operador	Significado
<b>AND</b>	Devuelve verdadero si las expresiones a su izquierda y derecha son ambas verdaderas
<b>OR</b>	Devuelve verdadero si cualquiera de las dos expresiones a izquierda y derecha del OR, son verdaderas
<b>NOT</b>	Invierte la lógica de la expresión que está a su derecha. Si era verdadera, mediante NOT pasa a ser falso.

Ejemplos:

```
/* Obtiene a las personas de entre 25 y 50 años */
SELECT nombre, apellido1, apellido2 FROM personas
WHERE edad >= 25 AND edad <= 50;

/* Obtiene a la gente de más de 60 años o de menos de 20 */
SELECT nombre, apellido1, apellido2 FROM personas
WHERE edad > 60 OR edad < 20;

/* Obtiene a la gente de con primer apellido entre la A y la O */
SELECT nombre, apellido1, apellido2 FROM personas
WHERE apellido1 > 'A' AND apellido2 < 'Z';
```

### (1.3.3) BETWEEN

El operador **BETWEEN** nos permite obtener datos que se encuentren en un rango. Uso:

```
SELECT tipo, modelo, precio FROM piezas
WHERE precio BETWEEN 3 AND 8;
```

Saca piezas cuyos precios estén entre 3 y 8 (ambos incluidos).

### (1.3.4) IN

Permite obtener registros cuyos valores estén en una lista de valores:

```
SELECT tipo,modelo,precio FROM piezas
WHERE precio IN (3,5, 8);
```

Obtiene piezas cuyos precios sean 3, 5 u 8 (no valen ni el precio 4 ni el 6, por ejemplo).

### (1.3.5) LIKE

Se usa sobre todo con textos, permite obtener registros cuyo valor en un campo cumpla una condición textual. LIKE utiliza una cadena que puede contener estos símbolos:

Símbolo	Significado
%	Una serie cualquiera de caracteres
_	Un carácter cualquiera

Ejemplos:

```
/* Selecciona nombres que empiecen por S */
SELECT nombre FROM personas WHERE nombre LIKE 'S%';
/*Selecciona las personas cuyo apellido sea Sanchez, Senchez,
Stnchez,...*/
SELECT apellido1 FROM Personas WHERE apellido1
LIKE 'S_nchez';
```

### (1.3.6) IS NULL

Devuelve verdadero si el valor que examina es nulo:

```
SELECT nombre,apellidos FROM personas
WHERE telefono IS NULL
```

Esa instrucción selecciona a la gente que no tiene teléfono. Se puede usar la expresión **IS NOT NULL** que devuelve verdadero en el caso contrario, cuando la expresión no es nula.

### (1.3.7) precedencia de operadores

A veces las expresiones que se producen en los **SELECT** son muy extensas y es difícil saber que parte de la expresión se evalúa primero, por ello se indica la siguiente tabla de precedencia:

Orden de precedencia	Operador
1	*(Multiplicar) / (dividir)
2	+ (Suma) - (Resta)
3	(Concatenación)
4	Comparaciones (>, <, !=, ...)
5	IS [NOT] NULL, [NOT]LIKE, IN
6	NOT
7	AND
8	OR

## (1.4) ordenación

El orden inicial de los registros obtenidos por un **SELECT** no guarda más que una relación respecto al orden en el que fueron introducidos. Para ordenar en base a criterios más interesantes, se utiliza la cláusula **ORDER BY**.

En esa cláusula se coloca una lista de campos que indica la forma de ordenar. Se ordena primero por el primer campo de la lista, si hay coincidencias por el segundo, si ahí también las hay por el tercero, y así sucesivamente.

Se puede colocar las palabras **ASC** O **DESC** (por defecto se toma **ASC**). Esas palabras significan en ascendente (de la A a la Z, de los números pequeños a los grandes) o en descendente (de la Z a la a, de los números grandes a los pequeños) respectivamente.

Sintaxis completa de **SELECT** (para una sola tabla):

```
SELECT { * | [DISTINCT] {columna | expresión} [[AS] alias], ... }  
FROM tabla  
[WHERE condición]  
[ORDER BY expresión1 [,expresión2,...][{ASC|DESC}]];
```

## (1.5) funciones

### (1.5.1) funciones

Todos los SGBD implementan funciones para facilitar la creación de consultas complejas. Esas funciones dependen del SGBD que utilizemos, las que aquí se comentan son algunas de las que se utilizan con Oracle.

Todas las funciones devuelven un resultado que procede de un determinado cálculo. La mayoría de funciones precisan que se les envíe datos de entrada (**parámetros** o **argumentos**) que son necesarios para realizar el cálculo de la función. Este resultado, lógicamente depende de los parámetros enviados. Dichos parámetros se pasan entre paréntesis. De tal manera que la forma de invocar a una función es:

```
nombreFunción[(parámetro1[, parámetro2,...])]
```

Si una función no precisa parámetros (como **SYSDATE**) no hace falta colocar los paréntesis.

En realidad, hay dos tipos de funciones:

- Funciones que operan con datos de la misma fila
- Funciones que operan con datos de varias filas diferentes (**funciones de agrupación**).

En este apartado se tratan las funciones del primer tipo (más adelante se comentan las de agrupación).

**Nota: tabla DUAL (Oracle)**

Oracle proporciona una tabla llamada dual con la que se permiten hacer pruebas. Esa tabla tiene un solo campo (llamado **DUMMY**) y una sola fila de modo que es posible hacer pruebas. Por ejemplo la consulta:

```
SELECT SQRT(5) FROM DUAL;
```

Muestra una tabla con el contenido de ese cálculo (la raíz cuadrada de 5). DUAL es una tabla interesante para hacer pruebas.

En los siguientes apartados se describen algunas de las funciones más interesantes, las más importantes son las remarcadas con un fondo naranja más intenso.



## (1.5.2) funciones numéricas

### redondeos

Función	Descripción
<b>ROUND</b> (n,decimales)	Redondea el número al siguiente número con el número de decimales indicado más cercano. <b>ROUND</b> (8.239,2) devuelve 8.24
<b>TRUNC</b> (n,decimales)	Los decimales del número se cortan para que sólo aparezca el número de decimales indicado

### matemáticas

Función	Descripción
<b>MOD</b> (n1,n2)	Devuelve el resto resultado de dividir <b>n1</b> entre <b>n2</b>
<b>POWER</b> (valor,exponente)	Eleva el valor al exponente indicado
<b>SQRT</b> (n)	Calcula la raíz cuadrada de n
<b>SIGN</b> (n)	Devuelve 1 si <b>n</b> es positivo, cero si vale cero y -1 si es negativo
<b>ABS</b> (n)	Calcula el valor absoluto de <b>n</b>
<b>EXP</b> (n)	Calcula $e^n$ , es decir el exponente en base <b>e</b> del número n
<b>LN</b> (n)	Logaritmo neperiano de <b>n</b>
<b>LOG</b> (n)	Logaritmo en base 10 de <b>n</b>
<b>SIN</b> (n)	Calcula el seno de <b>n</b> ( <b>n</b> tiene que estar en radianes)
<b>COS</b> (n)	Calcula el coseno de <b>n</b> ( <b>n</b> tiene que estar en radianes)
<b>TAN</b> (n)	Calcula la tangente de <b>n</b> ( <b>n</b> tiene que estar en radianes)
<b>ACOS</b> (n)	Devuelve en radianes el arco coseno de <b>n</b>
<b>ASIN</b> (n)	Devuelve en radianes el arco seno de <b>n</b>
<b>ATAN</b> (n)	Devuelve en radianes el arco tangente de <b>n</b>
<b>SINH</b> (n)	Devuelve el seno hiperbólico de <b>n</b>
<b>COSH</b> (n)	Devuelve el coseno hiperbólico de <b>n</b>
<b>TANH</b> (n)	Devuelve la tangente hiperbólica de <b>n</b>

### (1.5.3) funciones de caracteres

conversión del texto a mayúsculas y minúsculas

Función	Descripción
<b>LOWER</b> (texto)	Convierte el texto a minúsculas (funciona con los caracteres españoles)
<b>UPPER</b> (texto)	Convierte el texto a mayúsculas
<b>INITCAP</b> (texto)	Coloca la primera letra de cada palabra en mayúsculas

funciones de transformación

Función	Descripción
<b>RTRIM</b> (texto)	Elimina los espacios a la derecha del texto
<b>LTRIM</b> (texto)	Elimina los espacios a la izquierda que posea el texto
<b>TRIM</b> (texto)	Elimina los espacios en blanco a la izquierda y la derecha del texto y los espacios dobles del interior.
<b>TRIM</b> (caracteres <b>FROM</b> texto)	Elimina del texto los caracteres indicados. Por ejemplo <b>TRIM</b> ('h' <b>FROM</b> nombre) elimina las haches de la columna <b>nombre</b> que estén a la izquierda y a la derecha
<b>SUBSTR</b> (texto,n[,m])	Obtiene los <b>m</b> siguientes caracteres del texto a partir de la posición <b>n</b> (si <b>m</b> no se indica se cogen desde <b>n</b> hasta el final).
<b>LENGTH</b> (texto)	Obtiene el tamaño del texto
<b>INSTR</b> (texto, textoBuscado [,posInicial [, nAparición]])	Obtiene la posición en la que se encuentra el texto buscado en el texto inicial. Se puede empezar a buscar a partir de una posición inicial concreta e incluso indicar el número de aparición del texto buscado.  Ejemplo, si buscamos la letra <b>a</b> y ponemos 2 en <b>nAparición</b> , devuelve la posición de la segunda letra a del texto).  Si no lo encuentra devuelve 0
<b>REPLACE</b> (texto, textoABuscar, [textoReemplazo])	Buscar el texto a buscar en un determinado texto y lo cambia por el indicado como texto de reemplazo.  Si no se indica texto de reemplazo, entonces esta función elimina el texto a buscar



Función	Descripción
<b>TRANSLATE</b> (texto, caracteresACambiar, caracteresSustitutivos)	<p>Potentísima función que permite transformar caracteres. Los <b>caracteresACambiar</b> son los caracteres que se van a cambiar, los <b>caracteresSustitutivos</b> son los caracteres que reemplazan a los anteriores. De tal modo que el primer carácter a cambiar se cambia por el primer carácter sustitutivo, el segundo por el segundo y así sucesivamente. Ejemplo:</p> <pre>SELECT TRANSLATE('prueba','ue','wx') FROM DUAL;</pre> <p>El resultado sería el texto <b>prwxba</b>, de tal forma que la <b>u</b> se cambia por la <b>w</b> y la <b>e</b> por la <b>x</b>.</p> <p>Si la segunda cadena es más corta, los caracteres de la primera que no encuentran sustituto, se eliminan. Ejemplo:</p> <pre>SELECT TRANSLATE('prueba','ue','w') FROM DUAL;</pre> <p>Da como resultado <b>prwba</b></p>
<b>LPAD</b> (texto, anchuraMáxima, [caracterDeRelleno]) <b>RPAD</b> (texto, anchuraMáxima, [caracterDeRelleno])	<p>Rellena el texto a la izquierda (LPAD) o a la derecha (RPAD) con el carácter indicado para ocupar la anchura indicada.</p> <p>Si el texto es más grande que la anchura indicada, el texto se recorta.</p> <p>Si no se indica carácter de relleno se rellenará el espacio marcado con espacios en blanco.</p> <p>Ejemplo:</p> <pre>LPAD('Hola',10,'-')</pre> <p>da como resultado</p> <p><b>-----Hola</b></p>
<b>REVERSE</b> (texto)	Invierte el texto (le da la vuelta)

otras funciones de caracteres

Función	Descripción
<b>ASCII</b> (carácter)	Devuelve el código ASCII del carácter indicado
<b>CHR</b> (número)	Devuelve el carácter correspondiente al código ASCII indicado

Función	Descripción
<b>SOUNDEX</b> (texto)	<p>Devuelve el valor fonético del texto. Es una función muy interesante para buscar textos de los que se no se sabe con exactitud su escritura. Por ejemplo:</p> <pre>SELECT * FROM personas WHERE SOUNDEX(apellido1)=SOUNDEX('Smith')</pre> <p>En el ejemplo se busca a las personas cuyo primer apellido suena como <i>Smith</i></p>

### (1.5.4) funciones de trabajo con nulos

Permiten definir valores a utilizar en el caso de que las expresiones tomen el valor nulo.

Función	Descripción
<b>COALESCE</b> (listaExpresiones)	<p>Devuelve la primera de las expresiones que no es nula. Ejemplo<sup>2</sup>:</p> <pre>CREATE TABLE test ( col1 VARCHAR2(1), col2 VARCHAR2(1), col3 VARCHAR2(1));  INSERT INTO test VALUES (NULL, 'B', 'C'); INSERT INTO test VALUES ('A', NULL, 'C'); INSERT INTO test VALUES (NULL, NULL, 'C'); INSERT INTO test VALUES ('A', 'B', 'C');  SELECT COALESCE(col1, col2, col3) FROM test;</pre> <p>El resultado es:</p> <pre>B A C A</pre>
<b>NVL</b> (valor,sustituto)	<p>Si el <i>valor</i> es NULL, devuelve el valor <i>sustituto</i>; de otro modo, devuelve valor</p>

---

<sup>2</sup> Ejemplo tomado de [http://www.psoug.org/reference/string\\_func.html](http://www.psoug.org/reference/string_func.html)

Función	Descripción
<b>NVL2</b> (valor,sustituto1,sustituto2)	Variante de la anterior, devuelve el valor <i>sustituto1</i> si <i>valor</i> no es nulo. Si <i>valor</i> es nulo devuelve el <i>sustituto2</i>
<b>NULLIF</b> (valor1,valor2)	Devuelve nulo si <i>valor1</i> es igual a <i>valor2</i> . De otro modo devuelve <i>valor1</i>

## (1.5.5) funciones de expresiones regulares

### expresiones regulares

Las expresiones regulares son una parte cada vez más importante de los lenguajes de programación. Han existido desde que aparecieron los lenguajes de programación de alto nivel y cada vez tienen más importancia, hasta el punto de que lenguajes como **Perl** no tendrían cabida sin las expresiones regulares.

En el caso de las bases de datos, también tienen una enorme importancia, en especial para crear restricciones de validación (**CHECK**) que imponen una forma muy concreta que debe de cumplir el texto. Es lo que comúnmente se conoce como patrón o máscara de entrada.

Por ejemplo, si nuestra base de datos almacena DNIs, éstos se forman por ocho números y una letra. Si no queremos permitir otra posibilidad (por ejemplo, aceptar que se pueda introducir como *dni* al texto *abcdefgrjdddkjdkj*), entonces lo ideal es indicarlo mediante un check con una expresión regular, utilizando la función **REGEXP\_LIKE** por ejemplo.

Una expresión regular está formada por un texto que contiene símbolos especiales que son los que permiten indicar claramente la expresión. Tradicionalmente hay dos normas de expresiones regulares:

- **POSIX**. *Portable Operating System Interface*, es el formato que procede del sistema **Unix** (ahí la equis en sus siglas) y que ha sido aceptado por el organismo **IEEE** (*Institute of Electrical and Electronics Engineers*) de estándares. Oracle es totalmente compatible con el estándar POSIX correspondiente y exactamente al el borrador estándar 1003.2/D11.2.
- **PCRE** (*Perl Compatible Regular Expressions*). Proceden del lenguaje Perl (precisamente famoso por su uso de las expresiones regulares) y es un formato en claro ascenso. Oracle admite muchas de las expresiones correspondiente a esta especificación (las más populares)

Además, Oracle ha incluido la especificación que compatibiliza las expresiones regulares con las lenguas incluidas en Unicode, lo que evita problemas habituales con las expresiones regulares en otros lenguajes.

## caracteres de uso habitual en las expresiones regulares

símbolo	significado
<b>c</b>	Si <b>c</b> es un carácter cualquiera (por ejemplo <b>a</b> , <b>H</b> , <b>ñ</b> , <i>etc.</i> ) indica, donde aparezca dentro de la expresión, que en esa posición debe aparecer dicho carácter para que la expresión sea válida. Es decir, si nuestra expresión regular es simplemente ' <b>a</b> ' será válido para esa expresión cualquier texto que contenga la letra <b>a</b> .
<b>cde</b>	Siendo <b>c</b> , <b>d</b> , y <b>e</b> caracteres, indica que esos caracteres deben aparecer de esa manera en la expresión.
<b>^x</b>	Comenzar por. Indica el texto debe empezar por la expresión <b>x</b> . Por ejemplo: ' <b>^Ja</b> ' es una expresión que da por válido cualquier texto que empiece por <b>jota</b> y luego lleve una <b>a</b> .
<b>x\$</b>	Finalizar por. Indica que el String texto terminar con la expresión <b>x</b> .
<b>[cde]</b>	Opción, son válidos uno de estos caracteres: <b>c</b> , <b>d</b> ó <b>e</b>
<b>c-d</b>	Cumplen esa expresión los caracteres que, en orden ASCII, vayan del carácter <b>c</b> al carácter <b>d</b> . Por ejemplo <b>a-z</b> representa todas las letras minúsculas del alfabeto inglés.
<b>[^x]</b>	No es válido ninguno de los caracteres que cumplan la expresión <b>x</b> . Por ejemplo <b>[^dft]</b> indica que no son válidos los caracteres <b>d</b> , <b>f</b> ó <b>t</b> .
<b>.</b>	Cualquier carácter. El punto indica que en esa posición puede ir cualquier carácter
<b>x+</b>	La expresión a la izquierda de este símbolo se puede repetir una o más veces
<b>x*</b>	la expresión a la izquierda de este símbolo se puede repetir cero o más veces
<b>x?</b>	El carácter a la izquierda de este símbolo se puede repetir cero o una vez
<b>x{n}</b>	Significa que la expresión <b>x</b> aparecerá <b>n</b> veces, siendo <b>n</b> un número entero positivo.
<b>x{n,}</b>	Significa que la expresión <b>x</b> aparecerá <b>n</b> o más veces
<b>x{m,n}</b>	Significa que la expresión <b>x</b> aparecerá de <b>m</b> a <b>n</b> veces.
<b>(x)</b>	Permite indicar un subpatrón dentro del paréntesis. Ayuda a formar expresiones regulares complejas.
<b>x y</b>	La barra indica que las expresiones <b>x</b> e <b>y</b> son opcionales, se puede cumplir una u otra.

## caracteres especiales

Nos permiten indicar caracteres que de otra forma no se pueden escribir (como el salto de línea, por ejemplo), a se les llama también secuencias de escape y proceden del lenguaje **C**.

carácter	significado
<code>\t</code>	Tabulador
<code>\n</code>	Salto de línea
<code>\r</code>	Salto de carro
<code>\v</code>	Tabulador vertical
<code>\\</code>	La propia barra

También se utiliza si necesitamos poner como literales algunos de los símbolos especiales de las expresiones regulares. Por ejemplo, el símbolo punto (.) significa cualquier carácter, pero si indicamos `\.` entonces es el propio carácter punto. Por ejemplo si nuestra expresión verifica que un texto contiene una dirección de correo electrónico, pondríamos: `.*@.*\..*` y eso daría válido cualquier texto que empiece con lo que sea (.\*), que luego tenga el símbolo @, luego lo que sea (.\*), luego, obligatoriamente, un punto (\.) y luego lo que sea.

Lo mismo para otros símbolos: `\$`, `\[`, `\]`,...

## símbolos procedentes del modelo POSIX

Deben utilizarse siempre dentro de los corchetes, así por ejemplo si nuestra expresión se refiere a una letra, se pondría `[:alpha:]` y si es cualquier carácter excepto letra, se usa `[^[:alpha:]]` y si fuera una letra o el número5, sería: `[:alpha:]5`

clase	significado
<code>[:alpha:]</code>	Letra
<code>[:alnum:]</code>	Letra o número
<code>[:blank:]</code>	Espacio o tabulador
<code>[:cntrl:]</code>	Carácter de control
<code>[:digit:]</code>	Número
<code>[:graph:]</code>	Carácter visible (no valen ni espacios ni caracteres de control)
<code>[:lower:]</code>	Letra minúscula
<code>[:print:]</code>	Carácter visible o el espacio en blanco
<code>[:punct:]</code>	Caracteres de puntuación
<code>[:space:]</code>	Espacios en blanco y símbolos que añaden espacios (como el salto de línea, tabulador,...)
<code>[:upper:]</code>	Letra mayúscula
<code>[:xdigit:]</code>	Dígito hexadecimal



símbolos procedentes del modelo Perl de expresión regular (y válidos en Oracle)

clases	significado
<code>\d</code>	Dígito, vale cualquier dígito numérico
<code>\D</code>	Todo menos dígito
<code>\s</code>	Espacio en blanco
<code>\S</code>	Cualquier carácter salvo el espacio en blanco
<code>\w</code>	<i>Word</i> , carácter válido dentro de los que PHP considera para identificar variables. Es decir, letras, números o el guion bajo.
<code>\W</code>	Todo lo que no sea un carácter de tipo <i>Word</i> .

### ejemplos de expresiones regulares

**DNI** (ocho números y una letra).

```
'^[0-9]{8}[A-Z]$'
```

Se delimita entre `^` y `$` para que exactamente el contenido del texto que cumpla la expresión sea así. De otro modo este texto cumpliría la expresión:

```
holhola17672612Cholahola
```

*No es válido, gracias a `^y $`*

**Nº de serie** con una o dos letras mayúsculas, tres números, seguido de una letra un guión y dos números:

```
'^[A-Z]{1,2}[0-9]{3}[A-Z]-[0-9]{2}$'
```

**Código postal**, números del 01000 al 52999 (se hacen tres grupos: del 10000 al 19999; del 01000 al 09000; y del 51000 al 52999; de otro modo dejaríamos algún caso de más)

```
'^([1-4][0-9]{4}) | (0[1-9][0-9]{3})| (5[1-2][0-9]{3})$'
```

**Correo electrónico** (al menos una letra con posibilidad de punto (y si lo hay habría letra a sus dos lados y al menos un punto tras el símbolo de arroba, al final un punto y nombre de dominio de dos o tres letras; `\w` significa **palabra**)

```
'^[\\w]+(\\.\\w+)*@[\\w]+(\\.\\w+)*\\.([a-z]{2,3})$'
```

## funciones de Oracle para uso con expresiones regulares

Función	Descripción
<b>REGEXP_LIKE</b> (texto, expresionRegular [,flag])	<p>Devuelve verdadero si el texto cumple la expresión regular.</p> <p>El <b>flag</b> que se puede indicar como tercer parámetro es una de estas letras (entrecomilladas):</p> <ul style="list-style-type: none"> <li>• <b>i</b>. No distingue entre mayúsculas y minúsculas</li> <li>• <b>c</b>. Distingue entre mayúsculas y minúsculas</li> <li>• <b>n</b>. Hace que el punto represente a todos los caracteres excepto al salto de línea (<b>\n</b>).</li> <li>• <b>x</b>. Ignora los espacios en blanco</li> </ul>
<b>REGEXP_COUNT</b> (texto, expresionRegular [,posición [,flag]])	<p>Cuenta las veces que aparece la expresión regular en el texto. Se puede especificar la <b>posición</b> en el texto desde la que empezamos a buscar y el parámetro <b>flag</b> comentado anteriormente.</p>
<b>REGEXP_INSTR</b> (texto, expresionRegular [,posInicial [,nAparición [,modoPos [,flag [,subexpr]]]]])	<p>Es una versión de la función <b>INSTR</b>, pero con más potencia. Busca la expresión regular en el texto y devuelve su posición (o cero si no la encuentra).</p> <p>Se puede empezar a buscar desde una posición (indicando un número en el parámetro <b>posInicial</b>) e incluso indicar que buscamos no la primera aparición (que es como funciona por defecto) sino que buscamos el número de aparición que indiquemos en el cuarto parámetro.</p> <p>El quinto parámetro vale cero (valor por defecto) si la posición devuelta indica el primer carácter en el que aparece la expresión; y vale uno si la posición devuelta es la del siguiente carácter respecto al texto que cumple la expresión regular.</p> <p><b>flag</b> es el parámetro comentado en la función <b>REGEXP_LIKE</b>.</p>
<b>REGEXP_REPLACE</b> (texto, expresionRegular [, nuevoTexto [,posInicial [,nAparición [,flag]]]])	<p>Busca la expresión regular en el texto y reemplaza todo el texto que la cumple por el nuevo texto indicado.</p> <p>Si no hay <b>nuevoTexto</b>, entonces se eliminan los caracteres que cumplan la expresión regular</p>

Función	Descripción
<b>REGEXP_SUBSTR</b> (texto, expresionRegular [,posInicial [,nAparición [,flag [,subexpr]]]])	Busca la expresión regular en el texto y extrae los caracteres que cumplen la expresión. Es, por lo tanto, una versión más potente de <b>SUBSTR</b> .

## (1.5.6) funciones de fecha y manejo de fechas e intervalos

Las fechas se utilizan muchísimo en todas las bases de datos. MySQL proporciona dos tipos de datos para manejar fechas, los tipos **DATE** y **TIMESTAMP**. En el primer caso se almacena una fecha concreta (que incluso puede contener la hora), en el segundo caso se almacena un instante de tiempo más concreto que puede incluir incluso fracciones de segundo.

Hay que tener en cuenta que a los valores de tipo fecha se les pueden sumar números y se entendería que esta suma es de días. Si tiene decimales entonces se suman días, horas, minutos y segundos. La diferencia entre dos fechas también obtiene un número de días.

### intervalos

Los intervalos son datos relacionados con las fechas en sí, pero que no son fechas. Hay dos tipos de intervalos el **INTERVAL DAY TO SECOND** que sirve para representar días, horas, minutos y segundos; y el **INTERVAL YEAR TO MONTH** que representa años y meses.

Para los intervalos de año a mes los valores se pueden indicar de estas formas:

```
/* 123 años y seis meses */
INTERVAL '123-6' YEAR(4) TO MONTH
/* 123 años */
INTERVAL '123' YEAR(4) TO MONTH
/* 6 meses */
INTERVAL '6' MONTH(3) TO MONTH
```

La precisión en el caso de indicar tanto años como meses, se indica sólo en el año. En intervalos de días a segundos los intervalos se pueden indicar como:

```
/* 4 días 10 horas 12 minutos y 7 con 352 segundos */
INTERVAL '4 10:12:7,352' DAY TO SECOND(3)
/* 4 días 10 horas 12 minutos */
INTERVAL '4 10:12' DAY TO MINUTE
/* 4 días 10 horas */
INTERVAL '4 10' DAY TO HOUR
/* 4 días */
INTERVAL '4' DAY
/* 10 horas */
INTERVAL '10' HOUR
/* 25 horas */
INTERVAL '253' HOUR
/* 12 minutos */
```

```

INTERVAL '12' MINUTE
/*30 segundos */
INTERVAL '30' SECOND
/*8 horas y 50 minutos */
INTERVAL '8:50' HOUR TO MINUTE;
/*7 minutos 6 segundos*/
INTERVAL '7:06' MINUTE TO SECOND;
/*8 horas 7 minutos 6 segundos*/
INTERVAL '8:07:06' HOUR TO SECOND;

```

Esos intervalos se pueden sumar a valores de tipo **DATE** o **TIMESTAMP** para hacer cálculos. Gracias a ello se permiten sumar horas o minutos por ejemplo a los datos de tipo **TIMESTAMP**.

obtener la fecha y hora actual

Función	Descripción
<b>SYSDATE</b>	Obtiene la fecha y hora actuales
<b>SYSTIMESTAMP</b>	Obtiene la fecha y hora actuales en formato <b>TIMESTAMP</b>

calcular fechas

Función	Descripción
<b>ADD_MONTHS</b> (fecha,n)	Añade a la fecha el número de meses indicado por <i>n</i>
<b>MONTHS_BETWEEN</b> (fecha1, fecha2)	Obtiene la diferencia en meses entre las dos fechas (puede ser decimal)
<b>NEXT_DAY</b> (fecha,día)	Indica cual es el día que corresponde a añadir a la fecha el día indicado. El día puede ser el texto ' <i>Lunes</i> ', ' <i>Martes</i> ', ' <i>Miércoles</i> '... (si la configuración está en español) o el número de día de la semana (1=lunes, 2=martes...)
<b>LAST_DAY</b> (fecha)	Obtiene el último día del mes al que pertenece la fecha. Devuelve un valor DATE
<b>EXTRACT</b> (valor <b>FROM</b> fecha)	Extrae un valor de una fecha concreta. El valor puede ser <b>day</b> (día), <b>month</b> (mes), <b>year</b> (año), etc.
<b>GREATEST</b> (fecha1, fecha2,..)	Devuelve la fecha más moderna la lista
<b>LEAST</b> (fecha1, fecha2,..)	Devuelve la fecha más antigua la lista

Función	Descripción
<b>ROUND</b> (fecha [, 'formato'])	Redondea la fecha al valor de aplicar el formato a la fecha. El formato puede ser: ' <b>YEAR</b> ' Hace que la fecha refleje el año completo ' <b>MONTH</b> ' Hace que la fecha refleje el mes completo más cercano a la fecha ' <b>HH24</b> ' Redondea la hora a las 00:00 más cercanas ' <b>DAY</b> ' Redondea al día más cercano
<b>TRUNC</b> (fecha [formato])	Igual que el anterior pero trunca la fecha en lugar de redondearla.

### (1.5.7) funciones de conversión

Oracle es capaz de convertir datos automáticamente a fin de que la expresión final tenga sentido. En ese sentido son fáciles las conversiones de texto a número y viceversa. Ejemplo:

```
SELECT 5+'3' FROM DUAL /*El resultado es 8 */
SELECT 5 || '3' FROM DUAL /* El resultado es 53 */
```

También ocurre eso con la conversión de textos a fechas. De hecho, es forma habitual de asignar fechas.

Pero en diversas ocasiones querremos realizar conversiones explícitas.

#### TO\_CHAR

Obtiene un texto a partir de un número o una fecha. En especial se utiliza con fechas (ya que de número a texto se suele utilizar de forma implícita).

#### fechas

En el caso de las fechas se indica el formato de conversión, que es una cadena que puede incluir estos símbolos (en una cadena de texto):

Símbolo	Significado
<b>YY</b>	Año en formato de dos cifras
<b>YYYY</b>	Año en formato de cuatro cifras
<b>MM</b>	Mes en formato de dos cifras
<b>MON</b>	Las tres primeras letras del mes
<b>MONTH</b>	Nombre completo del mes
<b>DY</b>	Día de la semana en tres letras
<b>DAY</b>	Día completo de la semana
<b>D</b>	Día de la semana (del 1 al 7)

Símbolo	Significado
<b>DD</b>	Día del mes en formato de dos cifras (del 1 al 31)
<b>DDD</b>	Día del año
<b>Q</b>	Semestre
<b>WW</b>	Semana del año
<b>AM</b>	Indicador AM
<b>PM</b>	Indicador PM
<b>HH12</b>	Hora de 1 a 12
<b>HH24</b>	Hora de 0 a 23
<b>MI</b>	Minutos (0 a 59)
<b>SS</b>	Segundos (0 a 59)
<b>SSSS</b>	Segundos desde medianoche
<b>/ . , :: ' ,</b>	Posición de los separadores, donde se pongan estos símbolos aparecerán en el resultado

Ejemplos:

```
SELECT TO_CHAR(SYSDATE, 'DD/MONTH/YYYY, DAY HH:MI:SS')  
FROM DUAL ;  
/* Sale : 16/AGOSTO /2004, LUNES 08:35:15, por ejemplo*/
```

## números

Para convertir números a textos se usa esta función cuando se desean características especiales. En ese caso en el formato se pueden utilizar estos símbolos:

Símbolo	Significado
<b>9</b>	Posición del número
<b>0</b>	Posición del número (muestra ceros)
<b>\$</b>	Formato dólar
<b>L</b>	Símbolo local de la moneda
<b>S</b>	Hace que aparezca el símbolo del signo
<b>D</b>	Posición del símbolo decimal (en español, la coma)
<b>G</b>	Posición del separador de grupo (en español el punto)

## TO\_NUMBER

Convierte textos en números. Se indica el formato de la conversión (utilizando los mismos símbolos que los comentados anteriormente).

## TO\_DATE

Convierte textos en fechas. Como segundo parámetro se utilizan los códigos de formato de fechas comentados anteriormente.

## CAST

Función muy versátil que permite convertir el resultado a un tipo concreto. Sintaxis:

```
CAST(expresión AS tipoDatos)
```

Ejemplo:

```
SELECT CAST(2.34567 AS NUMBER(7,6)) FROM DUAL;
```

Lo interesante es que puede convertir de un tipo a otro. Por ejemplo, imaginemos que tenemos una columna en una tabla mal planteada en la que el precio de las cosas se ha escrito en euros. Los datos son (se muestra sólo la columna precio:

precio
25.2 €
2.8 €
123.65 €
.78 €
.123 €
20 €

Imaginemos que queremos doblar el precio, no podremos porque la columna es de tipo texto, por ello debemos tomar sólo la parte numérica y convertirla a número, después podremos mostrar los precios multiplicados por dos:

```
SELECT 2 * CAST(SUBSTR(precio,1,INSTR(precio,'€')-2) AS NUMBER)  
FROM precios;
```

La combinación de SUBSTR e INSTR es para obtener sólo los números. Incluso es posible que haya que utilizar REPLACE para cambiar los puntos por comas (para utilizar el separador decimal del idioma español).

## (1.5.8) función DECODE

Función que permite realizar condiciones en una consulta. Se evalúa una expresión y se colocan a continuación pares valor, resultado de forma que si se la expresión equivale al valor, se obtiene el resultado indicado. Se puede indicar un último parámetro con el resultado a efectuar en caso de no encontrar ninguno de los valores indicados.

Sintaxis:

```
DECODE(expresión, valor1, resultado1  
[valor2, resultado2,...]  
[valorPordefecto])
```

Ejemplo:

```
SELECT  
DECODE(cotizacion,1, salario*0.85,  
        2,salario * 0.93,  
        3,salario * 0.96,  
        salario)  
FROM empleados;
```

En el ejemplo dependiendo de la cotización se muestra rebajado el salario: un 85% si la cotización es uno, un 93 si es dos y un 96 si es tres. Si la cotización no es ni uno ni dos ni tres, sencillamente se muestra el salario sin más.

## (1.5) obtener datos de múltiples tablas

Es más que habitual necesitar en una consulta datos que se encuentran distribuidos en varias tablas. Las bases de datos relacionales se basan en que los datos se distribuyen en tablas que se pueden relacionar mediante un campo. Ese campo es el que permite integrar los datos de las tablas.

Por ejemplo, si disponemos de una tabla de empleados cuya clave es el *dni* y otra tabla de tareas que se refiere a tareas realizadas por los empleados, es seguro (si el diseño está bien hecho) que en la tabla de tareas aparecerá el dni del empleado para saber qué empleado realizó la tarea.

### (1.5.1) producto cruzado o cartesiano de tablas

En el ejemplo anterior si quiere obtener una lista de los datos de las tareas y los empleados, se podría hacer de esta forma:

```
SELECT cod_tarea, descripcion_tarea,  
       dni_empleado, nombre_empleado  
FROM tareas,empleados;
```



La sintaxis es correcta ya que, efectivamente, en el apartado **FROM** se pueden indicar varias tareas separadas por comas. Pero eso produce un producto cruzado, aparecerán todos los registros de las tareas relacionados con todos los registros de empleados..

El producto cartesiano a veces es útil para realizar consultas complejas, pero en el caso normal no lo es. necesitamos discriminar ese producto para que sólo aparezcan los registros de las tareas relacionadas con sus empleados correspondientes. A eso se le llama asociar (*join*) tablas

### (1.5.2) asociando tablas

La forma de realizar correctamente la consulta anterior (asociado las tareas con los empleados que la realizaron sería:

```
SELECT cod_tarea, descripcion_tarea, dni_empleado,  
nombre_empleado FROM tareas,empleados  
WHERE tareas.dni_empleado = empleados.dni;
```

Nótese que se utiliza la notación *tabla.columna* para evitar la ambigüedad, ya que el mismo nombre de campo se puede repetir en ambas tablas. Para evitar repetir continuamente el nombre de la tabla, se puede utilizar un alias de tabla:

```
SELECT a.cod_tarea, a.descripcion_tarea, b.dni_empleado,  
b.nombre_empleado FROM tareas a,empleados b  
WHERE a.dni_empleado = b.dni;
```

Al apartado WHERE se le pueden añadir condiciones encadenándolas con el operador AND. Ejemplo:

```
SELECT a.cod_tarea, a.descripcion_tarea  
FROM tareas a,empleados b  
WHERE a.dni_empleado = b.dni AND  
b.nombre_empleado='Javier';
```

Finalmente indicar que se pueden enlazar más de dos tablas a través de sus campos relacionados. Ejemplo:

```
SELECT a.cod_tarea, a.descripcion_tarea,  
b.nombre_empleado, c.nombre_utilisilio  
FROM tareas a,empleados b, utensilios_utilizados c  
WHERE a.dni_empleado = b.dni AND a.cod_tarea=c.cod_tarea;
```

### (1.5.3) relaciones sin igualdad

A las relaciones descritas anteriormente se las llama relaciones en igualdad (*equijoins*), ya que las tablas se relacionan a través de campos que contienen valores iguales en dos tablas.

Sin embargo, no siempre las tablas tienen ese tipo de relación, por ejemplo:

EMPLEADOS		
Empleado	Sueldo	
Antonio	18000	
Marta	21000	
Sonia	15000	

CATEGORIAS		
categoría	Sueldo mínimo	Sueldo máximo
D	6000	11999
C	12000	17999
B	18000	20999
A	20999	80000

En el ejemplo anterior podríamos averiguar la categoría a la que pertenece cada empleado, pero estas tablas poseen una relación que ya no es de igualdad.

La forma sería:

```
SELECT a.empleado, a.sueldo, b.categoria
FROM empleados a, categorias b
WHERE a.sueldo BETWEEN b.sueldo_minimo AND
      b.sueldo_maximo;
```

#### (1.5.4) sintaxis SQL 1999

En la versión SQL de 1999 se ideó una nueva sintaxis para consultar varias tablas. La razón fue separar las condiciones de asociación respecto de las condiciones de selección de registros. Oracle incorpora totalmente esta normativa.

La sintaxis completa es:

```
SELECT tabla1.columna1, tabla1.columna2,...
      tabla2.columna1, tabla2.columna2,... FROM tabla1
[CROSS JOIN tabla2]|
[NATURAL JOIN tabla2]|
[JOIN tabla2 USING(columna)]|
[JOIN tabla2 ON (tabla1.columa=tabla2.columa)]|
[LEFT|RIGHT|FULL OUTER JOIN tabla2 ON
      (tabla1.columa=tabla2.columa)]
```

Se describen sus posibilidades en los siguientes apartados.

## CROSS JOIN

Utilizando la opción **CROSS JOIN** se realiza un producto cruzado entre las tablas indicadas. Eso significa que cada tupla de la primera tabla se combina con cada tupla de la segunda tabla. Es decir si la primera tabla tiene 10 filas y la segunda otras 10, como resultado se obtienen 100 filas, resultado de combinar todas entre sí. Ejemplo:

```
SELECT * FROM piezas CROSS JOIN existencias;
```

No es una operación muy utilizada, aunque posibilita resolver consultas extremadamente complicadas.

## NATURAL JOIN

Establece una relación de igualdad entre las tablas a través de los campos que tengan el mismo nombre en ambas tablas:

```
SELECT * FROM piezas  
NATURAL JOIN existencias;
```

En ese ejemplo se obtienen los registros de piezas relacionados en existencias a través de los campos que tengan el mismo nombre en ambas tablas.

Hay que asegurarse de que sólo son las claves principales y secundarias de las tablas relacionadas, las columnas en las que el nombre coincide, de otro modo fallaría la asociación y la consulta no funcionaría.

## JOIN USING

Permite establecer relaciones indicando qué columna (o columnas) común a las dos tablas hay que utilizar:

```
SELECT * FROM piezas  
JOIN existencias USING(tipo,modelo);
```

Las columnas deben de tener exactamente el mismo nombre en ambas tablas-

## JOIN ON

Permite establecer relaciones cuya condición se establece manualmente, lo que permite realizar asociaciones más complejas o bien asociaciones cuyos campos en las tablas no tienen el mismo nombre:

```
SELECT * FROM piezas  
JOIN existencias ON(piezas.tipo=existencias.tipo AND  
piezas.modelo=existencias.modelo);
```

## relaciones externas

Utilizando las formas anteriores de relacionar tablas (incluidas las explicadas con la sintaxis SQL 92), sólo aparecen en el resultado de la consulta filas presentes en las tablas relacionadas. Es decir en esta consulta:

```
SELECT * FROM piezas  
JOIN existencias  
ON(piezas.tipo=existencias.tipo AND
```

```
piezas.modelo=existencias.modelo);
```

Sólo aparecen piezas presentes en la tabla de existencias. Si hay piezas que no están en existencias, éstas no aparecen (y si hay existencias que no están en la tabla de piezas, tampoco salen).

Por ello se permite utilizar relaciones laterales o externas (*outer join*). Su sintaxis es:

```
...  
{LEFT | RIGHT | FULL} OUTER JOIN tabla  
{ON(condición) | USING (expresion)}  
...
```

Así esta consulta:

```
SELECT * FROM piezas  
LEFT OUTER JOIN existencias  
ON(piezas.tipo=existencias.tipo AND  
piezas.modelo=existencias.modelo);
```

Obtiene los datos de las piezas estén o no relacionadas con datos de la tabla de existencias (la tabla **LEFT** sería piezas por que es la que está a la izquierda del **JOIN**).

En la consulta anterior, si el **LEFT** lo cambiamos por un **RIGHT**, aparecerán las existencias no presentes en la tabla piezas (además de las relacionadas en ambas tablas).

La condición **FULL OUTER JOIN** produciría un resultado en el que aparecen los registros no relacionados de ambas tablas (piezas sin existencias relacionadas y viceversa).

#### consultas de no coincidentes

Un caso típico de uso de las relaciones externas es el uso de consultas de no coincidentes. Estas consultas sustituyen de forma más eficiente al operador **NOT IN** (se explica más adelante) ya que permiten comprobar filas de una tabla que no están presentes en una segunda tabla que se relaciona con la primera a través de alguna clave secundaria.

Por ejemplo, supongamos que tenemos una base de datos que posee una tabla de empresas y otra de trabajadores de las empresas. Ambas tablas supongamos que se relacionan a través del CIF de la empresa. Si existen empresas de las que no tenemos ningún trabajador podremos saberlo mediante esta consulta:

```
SELECT e.nombre FROM empresas e  
LEFT OUTER JOIN trabajadores t ON (t.cif=e.cif)  
WHERE t.dni IS NULL;
```

En la consulta anterior gracias a utilizar el operador **LEFT OUTER**, obtenemos la lista completa de empresas (tengan o no trabajadores). Si eliminamos aquellas filas con el **dni** de los trabajadores nulo, entonces salen las empresas sin trabajadores relacionados (sólo las empresas sin trabajadores mostrarán valores nulos en los datos de los trabajadores).

## (1.6) agrupaciones

Es muy común utilizar consultas en las que se desee agrupar los datos a fin de realizar cálculos en vertical, es decir calculados a partir de datos de distintos registros.

Para ello se utiliza la cláusula GROUP BY que permite indicar en base a qué registros se realiza la agrupación. Con GROUP BY la instrucción SELECT queda de esta forma:

```
SELECT listaDeExpresiones
FROM listaDeTablas
[JOIN tablasRelacionadasYCondicionesDeRelación]
[WHERE condiciones]
[GROUP BY grupos]
[HAVING condicionesDeGrupo]
[ORDER BY columnas];
```

En el apartado GROUP BY, se indican las columnas por las que se agrupa. La función de este apartado es crear un único registro por cada valor distinto en las columnas del grupo. Si por ejemplo agrupamos en base a las columnas *tipo* y *modelo* en una tabla de *existencias*, se creará un único registro por cada tipo y modelo distintos:

```
SELECT tipo,modelo
FROM existencias
GROUP BY tipo,modelo;
```

Si la tabla de existencias sin agrupar es:

TI	MODELO	N_ALMACEN	CANTIDAD
AR	6	1	2500
AR	6	2	5600
AR	6	3	2430
AR	9	1	250
AR	9	2	4000
AR	9	3	678
AR	15	1	5667
AR	20	3	43
BI	10	2	340
BI	10	3	23
BI	38	1	1100
BI	38	2	540
BI	38	3	

La consulta anterior creará esta salida:

TI	MODELO
AR	6
AR	9
AR	15
AR	20
BI	10
BI	38

Es decir es un resumen de los datos anteriores. Los datos *n\_almacén* y *cantidad* no están disponibles directamente ya que son distintos en los registros del mismo grupo. Sólo se pueden utilizar desde funciones (como se verá ahora). Es decir esta consulta es errónea:

```
SELECT tipo,modelo, cantidad
FROM existencias
GROUP BY tipo,modelo;

SELECT tipo,modelo, cantidad
*
```

**ERROR en línea 1:**  
**ORA-00979: no es una expresión GROUP BY**

### (1.6.1) funciones de cálculo con grupos

Lo interesante de la creación de grupos es las posibilidades de cálculo que ofrece. Para ello se utilizan funciones que permiten trabajar con los registros de un grupo son:

Función	Significado
<b>COUNT</b> (*)	Cuenta los elementos de un grupo. Se utiliza el asterisco para no tener que indicar un nombre de columna concreto, el resultado es el mismo para cualquier columna
<b>SUM</b> (expresión)	Suma los valores de la expresión
<b>AVG</b> (expresión)	Calcula la media aritmética sobre la expresión indicada
<b>MIN</b> (expresión)	Mínimo valor que toma la expresión indicada
<b>MAX</b> (expresión)	Máximo valor que toma la expresión indicada
<b>STDDEV</b> (expresión)	Calcula la desviación estándar
<b>VARIANCE</b> (expresión)	Calcula la varianza

Todas las funciones de la tabla anterior se calculan para cada elemento del grupo, así la expresión:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo,modelo;
```

Obtiene este resultado:

TI	MODELO	SUM(CANTIDAD)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI	10	363
BI	38	1740

Se suman las cantidades para cada grupo

### (1.6.2) condiciones HAVING

A veces se desea restringir el resultado de una expresión agrupada, por ejemplo, con:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
WHERE SUM(Cantidad)>500
GROUP BY tipo,modelo;
```

Pero Oracle devolvería este error:

```
WHERE SUM(Cantidad)>500
*
ERROR en línea 3:
ORA-00934: función de grupo no permitida aquí
```

La razón es que Oracle calcula primero el WHERE y luego los grupos; por lo que esa condición no la puede realizar al no estar establecidos los grupos.

Por ello se utiliza la cláusula **HAVING**, que se ejecuta una vez realizados los grupos. Se usaría de esta forma:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo,modelo
HAVING SUM(Cantidad)>500;
```

Eso no implica que no se pueda usar WHERE. Esta expresión sí es válida:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
```

```
FROM existencias
WHERE tipo!='AR'
GROUP BY tipo,modelo
HAVING SUM(Cantidad)>500;
```

En definitiva, el orden de ejecución de la consulta marca lo que se puede utilizar con WHERE y lo que se puede utilizar con HAVING:

Para evitar problemas estos podrían ser los pasos en la ejecución de una instrucción de agrupación por parte del gestor de bases de datos:

- (4) Seleccionar las filas deseadas utilizando **WHERE**. Esta cláusula eliminará columnas en base a la condición indicada
- (5) Se establecen los grupos indicados en la cláusula **GROUP BY**
- (6) Se calculan los valores de las funciones de totales (**COUNT**, **SUM**, **AVG**,...)
- (7) Se filtran los registros que cumplen la cláusula **HAVING**
- (8) El resultado se ordena en base al apartado **ORDER BY**.

## (1.7) subconsultas

### (1.7.1) uso de subconsultas simples

Se trata de una técnica que permite utilizar el resultado de una tabla SELECT en otra consulta SELECT. Permite solucionar consultas que requieren para funcionar el resultado previo de otra consulta.

La sintaxis es:

```
SELECT listaExpresiones
FROM tabla
WHERE expresión OPERADOR
      (SELECT listaExpresiones
FROM tabla);
```

Se puede colocar el SELECT dentro de las cláusulas **WHERE**, **HAVING** o **FROM**. El operador puede ser **>**, **<**, **>=**, **<=**, **!=**, **=** o **IN**.

Ejemplo:

```
SELECT
nombre_empleado, paga
FROM empleados
WHERE paga <
(SELECT paga FROM empleados
WHERE
nombre_empleado='Martina')
;
```

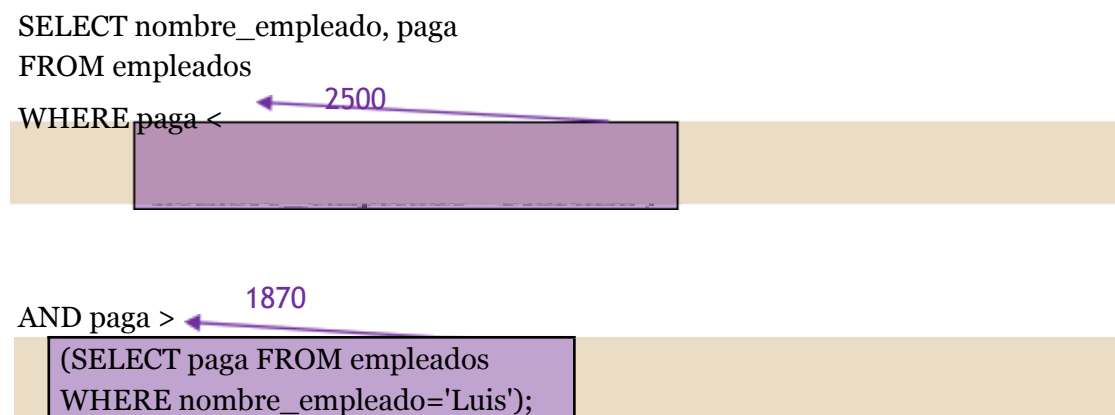


Lógicamente el resultado de la subconsulta debe incluir el campo que estamos analizando. Se pueden realizar esas subconsultas las veces que haga falta:

```
SELECT
nombre_empleado, paga
FROM empleados
WHERE paga <
(SELECT paga FROM empleados
WHERE
nombre_empleado='Martina')
AND paga >
(SELECT paga FROM empleados WHERE
nombre_empleado='Luis');
```

En realidad lo primero que hace la base de datos es calcular el resultado de la subconsulta:

```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga <
AND paga >
(SELECT paga FROM empleados
WHERE nombre_empleado='Luis');
```



La última consulta obtiene los empleados cuyas pagas estén entre lo que gana Luís (1870 euros) y lo que gana Martina (2500).

Las subconsultas siempre se deben encerrar entre paréntesis y se debería colocar a la derecha del operador relacional. Una subconsulta que utilice los valores >, <, >=, ... tiene que devolver un único valor, **de otro modo ocurre un error**. Además, tienen que tener el mismo tipo de columna para relacionar la subconsulta con la consulta que la utiliza (no puede ocurrir que la subconsulta tenga dos columnas y ese resultado se compare usando una sola columna en la consulta general).

### (1.7.2) uso de subconsultas de múltiples filas

En el apartado anterior se comentaba que las subconsultas sólo pueden devolver una fila. Pero a veces se necesitan consultas del tipo: *mostrar el sueldo y nombre de los empleados cuyo sueldo supera al de cualquier empleado del departamento de ventas*.

La subconsulta necesaria para ese resultado mostraría **todos** los sueldos del departamento de ventas. Pero no podremos utilizar un operador de comparación directamente ya que esa subconsulta devuelve más de una fila. La solución a esto es utilizar instrucciones especiales entre el operador y la consulta, que permiten el uso de subconsultas de varias filas.

Esas instrucciones son:

Instrucción	Significado
ANY	Compara con cualquier registro de la subconsulta. La instrucción es válida si hay un registro en la subconsulta que permite que la comparación sea cierta
ALL	Compara con todos los registros de la consulta. La instrucción resulta cierta si es cierta toda comparación con los registros de la subconsulta
IN	No usa comparador, ya que sirve para comprobar si un valor se encuentra en el resultado de la subconsulta
NOT IN	Comprueba si un valor no se encuentra en una subconsulta

Ejemplo:

```
SELECT nombre, sueldo
FROM empleados
WHERE sueldo >= ALL (SELECT sueldo FROM empleados);
```

La consulta anterior obtiene el empleado que más cobra. Otro ejemplo:

```
SELECT nombre FROM empleados
WHERE dni IN (SELECT dni FROM directivos);
```

En ese caso se obtienen los nombres de los empleados cuyos *dni* están en la tabla de directivos. Si se necesita comprobar dos columnas en una consulta IN, se hace:

```
SELECT nombre FROM empleados
WHERE (cod1,cod2) IN (SELECT cod1,cod2 FROM directivos);
```

### (1.7.3) consultas EXISTS

Este operador devuelve verdadero si la consulta que le sigue devuelve algún valor. Si no, devuelve falso. Se utiliza sobre todo en consultas correlacionadas. Ejemplo:

```
SELECT tipo,modelo, precio_venta
FROM piezas p
WHERE EXISTS (
    SELECT tipo,modelo FROM existencias
    WHERE tipo=p.tipo AND modelo=p.modelo);
```

Esta consulta devuelve las piezas que se encuentran en la tabla de existencias (es igual al ejemplo comentado en el apartado subconsultas sobre múltiples valores). La consulta contraria es :

```
SELECT tipo,modelo, precio_venta
FROM piezas p
WHERE NOT EXISTS (
    SELECT tipo,modelo FROM existencias
    WHERE tipo=p.tipo AND modelo=p.modelo);
```

Normalmente las consultas EXISTS se pueden realizar de alguna otra forma con los operadores ya comentados.

## (1.8) combinaciones especiales

### (1.8.1) uniones

La palabra **UNION** permite añadir el resultado de un SELECT a otro SELECT. Para ello ambas instrucciones tienen que utilizar el mismo número y tipo de columnas. Ejemplo:

```
SELECT nombre FROM provincias
UNION
SELECT nombre FROM comunidades
```

El resultado es una tabla que contendrá nombres de provincia y de comunidades. Es decir, UNION crea una sola tabla con registros que estén presentes en cualquiera de las consultas. Si están repetidas sólo aparecen una vez, para mostrar los duplicados se utiliza **UNION ALL** en lugar de la palabra **UNION**.

Es muy importante señalar que tanto esta cláusula como el resto de combinaciones especiales, requieren en los dos SELECT que unen el mismo tipo de columnas (y en el mismo orden).

### (1.8.2) intersecciones

De la misma forma, la palabra **INTERSECT** permite unir dos consultas SELECT de modo que el resultado serán las filas que estén presentes en ambas consultas.

Ejemplo; tipos y modelos de piezas que se encuentren sólo en los almacenes 1 y 2:

```
SELECT tipo,modelo FROM existencias
WHERE n_almacen=1
INTERSECT
SELECT tipo,modelo FROM existencias
WHERE n_almacen=2
```

### (1.8.3) diferencia

Con **MINUS** también se combinan dos consultas SELECT de forma que aparecerán los registros del primer SELECT que no estén presentes en el segundo.

Ejemplo; tipos y modelos de piezas que se encuentren el almacén 1 y no en el 2

```
(SELECT tipo,modelo FROM existencias
WHERE n_almacen=1)
MINUS(SELECT tipo,modelo FROM existencias
WHERE n_almacen=2)
```

Se podrían hacer varias combinaciones anidadas (una unión cuyo resultado se intersectará con otro SELECT por ejemplo), en ese caso es conveniente utilizar paréntesis para indicar qué combinación se hace primero:

```
(SELECT....
....
UNION
SELECT....
...
)
MINUS
SELECT /* Primero se hace la unión y luego la diferencia*/
```

## (1.9) consultas avanzadas

### (1.9.1) consultas con ROWNUM

La función **ROWNUM** devuelve el número de la fila de una consulta. Por ejemplo en:

```
SELECT ROWNUM, edad, nombre FROM clientes
```

Aparece el número de cada fila en la posición de la tabla. Esa función actualiza sus valores usando subconsultas de modo que la consulta:

```
SELECT ROWNUM AS ranking, edad, nombre FROM clientes
FROM (SELECT edad, nombre FROM clientes ORDER BY edad DESC)
```

Puesto que la consulta `SELECT edad, nombre FROM clientes ORDER BY edad DESC`, obtiene una lista de los clientes ordenada por edad, el SELECT superior obtendrá esa lista, pero mostrando el orden de las filas en esa consulta. Eso permite hacer consultas el tipo top-n, (los n más).

Por ejemplo, para sacar el top-10 de la edad de los clientes (los 10 clientes más mayores):

```
SELECT ROWNUM AS ranking, edad, nombre FROM clientes
FROM (SELECT edad, nombre FROM clientes ORDER
BY edad DESC) WHERE ROWNUM<=10
```

### (1.9.2) consultas sobre estructuras jerárquicas

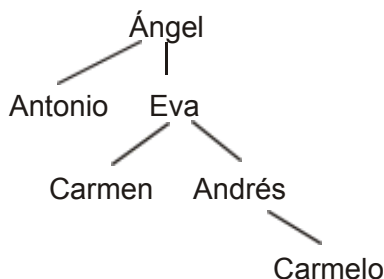
Imaginemos una tabla de empleados definida por un código de empleado, nombre del mismo y el código del jefe. Este último código está relacionado con el código de empleado que posee el jefe en cuestión. Así definido, una consulta que muestre el nombre de un empleado y el nombre de su jefe directo, sería:

```
SELECT e.nombre AS empleado, j.nombre AS jefe
FROM empleados e
JOIN empleados j ON (e.cod_jefe=j.cod_empleado);
```

Saldría, por ejemplo:

EMPLEADO	JEFE
Antonio	Ángel
Ángel	
Eva	Ángel
Carmen	Eva
Andrés	Eva
Carmelo	Andrés

En el ejemplo se observa como un jefe puede tener otro jefe, generando una estructura jerárquica:



En este tipo de estructuras, a veces se requieren consultas que muestren todos los empleados de un jefe, mostrando los mandos intermedios. Se trata de una consulta que recorre ese árbol. Este tipo de consultas posee esta sintaxis:

```
SELECT [LEVEL,] listaDeColumnasYExpresiones
FROM tabla(s)...
[WHERE condiciones...]
[START WITH condiciones]
CONNECT BY [NOCYCLE] [PRIOR] expresion1=[PRIOR] expresion2
```

El apartado **CONNECT** permite indicar qué relación hay que seguir para recorrer el árbol. La palabra **PRIOR** indica hacia dónde se dirige el recorrido. Finalmente, el apartado **START** indica la condición de inicio del recorrido (normalmente la condición

que permita buscar el nodo del árbol por el que comenzamos el recorrido, es decir sirve para indicar desde donde comenzamos. Ejemplo:

```
SELECT nombre FROM empleados
START WITH nombre='Andrés'
CONNECT BY PRIOR n_jefe=n_empleado;
```

Resultado:

NOMBRE
Andrés
Eva
Ángel

Sin embargo:

```
SELECT nombre FROM empleados
START WITH nombre='Andrés'
CONNECT BY n_jefe= PRIOR n_empleado;
```

Devuelve:

NOMBRE
Andrés
Carmelo

Si en lugar de Andrés en esa consulta buscáramos desde Ángel, saldría:

NOMBRE
Ángel
Antonio
Eva
Carmen
Andrés
Carmelo

El modificador LEVEL permite mostrar el nivel en el árbol jerárquico de cada elemento:

```
SELECT LEVEL, nombre FROM empleados
START WITH nombre='Ángel'
CONNECT BY n_jefe= PRIOR n_empleado;
```

Resultado:

LEVEL	NOMBRE
1	Ángel
2	Antonio
2	Eva
3	Carmen
3	Andrés
4	Carmelo

Para eliminar recorridos, se utilizan condiciones en WHERE o en el propio CONNECT. De modo que :

```
SELECT LEVEL, nombre FROM empleados
WHERE nombre!='Eva'
START WITH nombre='Ángel'
CONNECT BY n_jefe= PRIOR n_empleado;
```

En ese ejemplo, Eva no sale en los resultados. En este otro:

```
SELECT LEVEL, nombre FROM empleados
START WITH nombre='Ángel'
CONNECT BY n_jefe= PRIOR n_empleado AND nombre!='Eva';
```

No sale ni Eva ni sus empleados (se corta la rama entera).

Cuando los datos forman un árbol jerárquico (como ocurre con los ejemplos) no suele haber ningún problema. Pero hay diseños en los que hay padres de hijos que a su vez pueden ser sus padres. Con los padres e hijos no ocurre esta situación evidentemente. Pero por ejemplo si tuviéramos un diseño de redes sociales donde se apunta el nombre del usuario y el nombre de sus amigos, entonces resulta que yo estaré apuntado como amigo de una persona que, a su vez, es mi amigo.

En ese caso resultaría un bucle infinito cuando se hace esta consulta:

```
SELECT amigo FROM contactos
START WITH nombre='Ángel'
CONNECT BY PRIOR amigo=nombre;
-- Ocurre un bucle imposible de solucionar
```

Para esos casos disponemos de la cláusula **NOCYCLE** que controla los resultados repetidos y evita esos caminos. En cualquier caso, consultas en datos no jerarquizados en forma de árbol sino en forma de grafo (como el comentado ejemplo de las redes sociales), pueden ocupar muchísimo tiempo a Oracle por lo que no es mala idea pensar en otra solución con ayuda de PL/SQL (lenguaje que se comenta en estos mismos apuntes).

```
SELECT DISTINCT amigo FROM contactos
START WITH nombre='Ángel'
CONNECT BY NOCYCLE PRIOR amigo=nombre; -- Ahora sí
```

Por cierto, el DISTINCT impide que aparezcan muchas veces las mismas personas. Ya que pueden ser amigos de muchos de mis amigos (e incluso de muchos de los amigos de mis amigos).

## (1.10) consultas de agrupación avanzada

### (1.10.1) ROLLUP

Esta expresión en una consulta de agrupación (GROUP BY) permite obtener los totales de la función utilizada para calcular en esa consulta. Ejemplo:

```
SELECT tipo, modelo, SUM(cantidad)
FROM existencias
GROUP BY tipo, modelo;
```

Esta consulta suma las cantidades de la tabla existencias por cada tipo y modelo distinto. Si añadimos:

```
SELECT tipo, modelo, SUM(cantidad)
FROM existencias
GROUP BY ROLLUP (tipo, modelo);
```

Entonces nos añade un registro para cada tipo en el que aparece la suma del total para ese tipo. Al final mostrará un registro con el total absoluto. Es decir el resultado de esa consulta es:

TI	MODELO	SUM(CANTIDAD)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
AR		21168
BI	10	363
BI	38	1740
BI	57	1638
BI		3741
CL	12	7000
CL	15	3068
CL	18	6612
CL		16680
EM	21	257
EM	42	534
EM		791



TI	MODELO	SUM(CANTIDAD)
PU	5	12420
PU	9	7682
PU		20102
TO	6	464
TO	9	756
TO	10	987
TO	12	7740
TO	16	356
TO		10303
TU	6	277
TU	9	876
TU	10	1023
TU	12	234
TU	16	654
TU		3064
		75849

Se pueden unir varias columnas entre paréntesis para tratarlas como si fueran una unidad:

```
SELECT tipo, modelo, n_almacen, SUM(cantidad)
FROM existencias
GROUP BY ROLLUP ((tipo,modelo), (n_almacen));
```

La diferencia respecto a la anterior es que el total mostado por ROLLUP se referirá al tipo y al modelo.

### (1.10.2) CUBE

Es muy similar al anterior, sólo que este calcula todos los subtotales relativos a la consulta. Ejemplo:

```
SELECT tipo, modelo, SUM(cantidad)
FROM existencias
GROUP BY CUBE (tipo,modelo);
```

Resulta:

TI	MODELO	SUM(CANTIDAD)
		75849
	5	12420
	6	11271

TI	MODELO	SUM(CANTIDAD)
	9	14242
	10	2373
	12	14974
	15	8735
	16	1010
	18	6612
	20	43
	21	257
	38	1740
	42	534
	57	1638
AR		21168
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI		3741
BI	10	363
BI	38	1740
BI	57	1638
CL		16680
CL	12	7000
CL	15	3068
CL	18	6612
EM		791
EM	21	257
EM	42	534
PU		20102
PU	5	12420
PU	9	7682
TO		10303
TO	6	464
TO	9	756
TO	10	987
TO	12	7740
TO	16	356

TI	MODELO	SUM(CANTIDAD)
TU		3064
TU	6	277
TU	9	876
TU	10	1023
TU	12	234
TU	16	

Es decir, calcula totales por tipo, por modelo y el total absoluto.

### (1.10.3) GROUPING

Se trata de una función que funciona con **ROLLUP** y **CUBE** y que recibe uno o más campos e indica si la fila muestra un subtotal referido a los campos en cuestión. Si la fila es un subtotal de esos campos pone **1**, sino lo marca con **0**. Ejemplo:

```
SELECT tipo, modelo, SUM(cantidad),
       GROUPING(tipo), GROUPING(modelo)
FROM existencias
GROUP BY CUBE (tipo,modelo);
```

Sale:

TIPO	MODELO	SUM(CANTIDAD)	GROUPING(TIPO)	GROUPING(MODELO)
		75849	1	1
	5	12420	1	0
	6	11271	1	0
	9	14242	1	0
	10	2373	1	0
	12	14974	1	0
	15	8735	1	0
	16	1010	1	0
	18	6612	1	0
	20	43	1	0
	21	257	1	0
	38	1740	1	0
	42	534	1	0
	57	1638	1	0
AR		21168	0	1
AR	6	10530	0	0
AR	9	4928	0	0

TIPO	MODELO	SUM(CANTIDAD)	GROUPING(TIPO)	GROUPING(MODELO)
AR	15	5667	0	0
AR	20	43	0	0
BI		3741	0	1
BI	10	363	0	0
BI	38	1740	0	0
BI	57	1638	0	0
CL		16680	0	1
CL	12	7000	0	0
CL	15	3068	0	0
CL	18	6612	0	0
EM		791	0	1
EM	21	257	0	0
EM	42	534	0	0
PU		20102	0	1
PU	5	12420	0	0
PU	9	7682	0	0
TO		10303	0	1
TO	6	464	0	0
TO	9	756	0	0
TO	10	987	0	0
TO	12	7740	0	0
TO	16	356	0	0
TU		3064	0	1
TU	6	277	0	0
TU	9	876	0	0
TU	10	1023	0	0
TU	12	234	0	0
TU	16	654	0	0

Se utiliza sobre todo para preparar un consulta para la creación de informes.

#### (1.10.4) GROUPING SETS

Se trata de una mejora de Oracle 9i que permite realizar varias agrupaciones para la misma consulta. Sintaxis:

```
SELECT...  
...  
GROUP BY GROUPING SETS (listaDeCampos1) [,(lista2)...]
```

Las listas indican los campos por los que se realiza la agrupación. Ejemplo:

```
SELECT tipo, modelo, n_almacen, SUM(cantidad)  
FROM existencias  
GROUP BY GROUPING SETS ((tipo,modelo), (n_almacen));
```

De esa consulta se obtiene:

TI	MODELO	N_ALMACEN	SUM(CANTIDAD)
AR	6		10530
AR	9		4928
AR	15		5667
AR	20		43
BI	10		363
BI	38		1740
BI	57		1638
CL	12		7000
CL	15		3068
CL	18		6612
EM	21		257
EM	42		534
PU	5		12420
PU	9		7682
TO	6		464
TO	9		756
TO	10		987
TO	12		7740
TO	16		356
TU	6		277
TU	9		876
TU	10		1023

TI	MODELO	N_ALMACEN	SUM(CANTIDAD)
TU	12		234
TU	16		654
		1	30256
		2	40112
		3	5481

Se trata de dos consultas de totales unidades

### (1.10.5) conjuntos de agrupaciones combinadas

Se pueden combinar agrupaciones de diversas formas creando consultas como:

```
SELECT tipo, modelo, n_almacen, SUM(cantidad)
FROM existencias
GROUP BY tipo, ROLLUP(modelo), CUBE(n_almacen)
```

Que mostraría un informe espectacular sobre las tablas anteriores. Así como:

```
SELECT tipo, modelo, n_almacen, SUM(cantidad)
FROM existencias
GROUP BY GROUPING SETS(tipo, modelo), GROUPING
SETS(tipo, n_almacen)
```

## (1.11) DQL en instrucciones DML

A pesar del poco ilustrativo título de este apartado, la idea es sencilla. Se trata de cómo utilizar instrucciones SELECT dentro de las instrucciones DML (**INSERT**, **DELETE** o **UPDATE**), ello permite dar más potencia a dichas instrucciones.

### (1.11.1) relleno de registros a partir de filas de una consulta

Hay un tipo de consulta, llamada de adición de datos, que permite rellenar datos de una tabla copiando el resultado de una consulta. Se hace mediante la instrucción **INSERT** y, en definitiva, permite copiar datos de una consulta a otra.

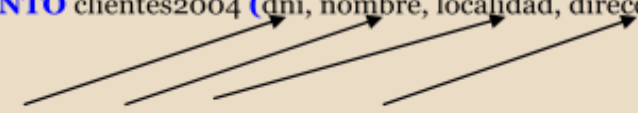
Ese relleno se basa en una consulta SELECT que poseerá los datos a añadir. Lógicamente el orden de esos campos debe de coincidir con la lista de campos indicada en la instrucción INSERT.

Sintaxis:

```
INSERT INTO tabla (campo1, campo2,...)
SELECT campoCompatibleCampo1, campoCompatibleCampo2,...
FROM lista DeTablas
[...otras cláusulas del SELECT...]
```

Ejemplo:

```
INSERT INTO clientes2004 (dni, nombre, localidad, direccion)
SELECT dni, nombre, localidad, direccion
FROM clientes
WHERE problemas=0;
```



Lógicamente las columnas del **SELECT** se tienen que corresponder con las columnas a rellenar mediante **INSERT** (observar las flechas).

### (1.11.2) subconsultas en la instrucción **UPDATE**

La instrucción **UPDATE** permite modificar filas. Es muy habitual el uso de la cláusula **WHERE** para indicar las filas que se modificarán. Esta cláusula se puede utilizar con las mismas posibilidades que en el caso del **SELECT**, por lo que es posible utilizar subconsultas. Por ejemplo:

```
UPDATE empleados
SET sueldo=sueldo*1.10
WHERE id_seccion =(SELECT id_seccion FROM secciones
                     WHERE nom_seccion='Producción');
```

Esta instrucción aumenta un 10% el sueldo de los empleados de la sección llamada *Producción*. También podemos utilizar subconsultas en la cláusula **SET** de la instrucción **UPDATE**.

Ejemplo:

```
UPDATE empleados
SET puesto_trabajo=(SELECT puesto_trabajo
                     FROM empleados
                     WHERE id_empleado=12)
WHERE seccion=23;
```

Esta instrucción coloca a todos los empleados de la sección 23 el mismo puesto de trabajo que el empleado número 12. Este tipo de actualizaciones sólo son válidas si el *subselect* devuelve un único valor, que además debe de ser compatible con la columna que se actualiza.

Hay que tener en cuenta que las actualizaciones no pueden saltarse las reglas de integridad que posean las tablas.

### (1.11.3) subconsultas en la instrucción DELETE

Al igual que en el caso de las instrucciones **INSERT** o **SELECT**, **DELETE** dispone de cláusula **WHERE** y en dicha cláusulas podemos utilizar subconsultas. Por ejemplo:

```
DELETE empleados
WHERE id_empleado IN
(SELECT id_empleado FROM errores_graves);
```

En este caso se trata de una subconsulta creada con el operador **IN**, se eliminarán los empleados cuyo identificador esté dentro de la tabla de *errores graves*.

## (1.12) vistas

### (1.12.1) introducción

Una vista no es más que una consulta almacenada a fin de utilizarla tantas veces como se desee. Una vista no contiene datos sino la instrucción **SELECT** necesaria para crear la vista, eso asegura que los datos sean coherentes al utilizar los datos almacenados en las tablas. Por todo ello, las vistas gastan muy poco espacio de disco.

Las vistas se emplean para:

- Realizar consultas complejas más fácilmente, ya que permiten dividir la consulta en varias partes
- Proporcionar tablas con datos completos
- Utilizar visiones especiales de los datos
- Ser utilizadas como tablas que resumen todos los datos
- Ser utilizadas como cursores de datos en los lenguajes procedimentales (como PL/SQL)

Hay dos tipos de vistas:

- **Simples.** Las forman una sola tabla y no contienen funciones de agrupación. Su ventaja es que permiten siempre realizar operaciones DML sobre ellas.
- **Complejas.** Obtienen datos de varias tablas, pueden utilizar funciones de agrupación. No siempre permiten operaciones DML.



## (1.12.2) creación de vistas

Sintaxis:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW vista
      [(alias[, alias2...])]
AS consultaSELECT
[WITH CHECK OPTION [CONSTRAINT restricción]]
[WITH READ ONLY [CONSTRAINT restricción]]
```

- **OR REPLACE.** Si la vista ya existía, la cambia por la actual
- **FORCE.** Crea la vista aunque los datos de la consulta SELECT no existan
- ***vista*.** Nombre que se le da a la vista
- ***alias*.** Lista de alias que se establecen para las columnas devueltas por la consulta SELECT en la que se basa esta vista. El número de alias debe coincidir con el número de columnas devueltas por SELECT.
- **WITH CHECK OPTION.** Hace que sólo las filas que se muestran en la vista puedan ser añadidas (**INSERT**) o modificadas (**UPDATE**). La ***restricción*** que sigue a esta sección es el nombre que se le da a esta restricción de tipo **CHECK OPTION**.
- **WITH READ ONLY.** Hace que la vista sea de sólo lectura. Permite grabar un nombre para esta restricción.

Lo bueno de las vistas es que tras su creación se utilizan como si fueran una tabla.  
Ejemplo:

```
CREATE VIEW resumen
/* alias */
(id_localidad, localidad, poblacion, n_provincia,
provincia,superficie, capital_provincia,
id_comunidad, comunidad, capital_comunidad)
AS
( SELECT l.id_localidad, l.nombre,
      l.poblacion, n_provincia, p.nombre,
      p.superficie, l2.nombre,
      id_comunidad, c.nombre, l3.nombre
FROM localidades l
JOIN provincias p USING
(n_provincia) JOIN comunidades c
USING (id_comunidad)
JOIN localidades l2 ON
(p.id_capital=l2.id_localidad) JOIN
localidades l3 ON
(c.id_capital=l3.id_localidad)
);

SELECT DISTINCT (comunidad, capital_comunidad)
FROM resumen; /* La vista pasa a usarse como una tabla normal*/
```

La creación de la vista del ejemplo es compleja ya que hay relaciones complicadas, pero una vez creada la vista, se le pueden hacer consultas como si se tratara de una tabla normal. Incluso se puede utilizar el comando **DESCRIBE** sobre la vista para mostrar la estructura de los campos que forman la vista o utilizarse como subconsulta en los comandos UPDATE o DELETE.

### (1.12.3) mostrar la lista de vistas

La vista del diccionario de datos de Oracle **USER\_VIEWS** permite mostrar una lista de todas las vistas que posee el usuario actual. Es decir, para saber qué vistas hay disponibles se usa:

```
SELECT * FROM USER_VIEWS;
```

La columna **TEXT** de esa vista contiene la sentencia SQL que se utilizó para crear la vista (sentencia que es ejecutada cada vez que se invoca a la vista).

### (1.12.4) borrar vistas

Se utiliza el comando DROP VIEW:

```
DROP VIEW nombreDeVista;
```