

# Introducción

## ¿Qué es el SQL ?

El **SQL** (Structured query language), **lenguaje de consulta estructurado**, es un lenguaje surgido de un proyecto de investigación de IBM para el acceso a bases de datos relacionales. Actualmente se ha convertido en un **estándar** de lenguaje de bases de datos, y la mayoría de los sistemas de bases de datos lo soportan, desde sistemas para ordenadores personales, hasta grandes ordenadores.

Por supuesto, a partir del estándar cada sistema ha desarrollado su propio SQL que puede variar de un sistema a otro, pero con cambios que no suponen ninguna complicación para alguien que conozca un SQL concreto, como el que vamos a ver aquí correspondiente al Access2000.

Como su nombre indica, el SQL nos **permite** realizar **consultas a la base de datos**. Pero el nombre se queda corto ya que SQL además realiza funciones de **definición, control y gestión de la base de datos**. Las sentencias SQL se clasifican según su finalidad dando origen a tres 'lenguajes' o mejor dicho sublenguajes:

- el **DDL** (Data Description Language), **lenguaje de definición** de datos, incluye órdenes para definir, modificar o borrar las tablas en las que se almacenan los datos y de las relaciones entre estas. (Es el que más varía de un sistema a otro)

- el **DCL** (Data Control Language), **lenguaje de control** de datos, contiene elementos útiles para trabajar en un entorno multiusuario, en el que es importante la protección de los datos, la seguridad de las tablas y el establecimiento de restricciones en el acceso, así como elementos para coordinar la compartición de datos por parte de usuarios concurrentes, asegurando que no interfieren unos con otros.

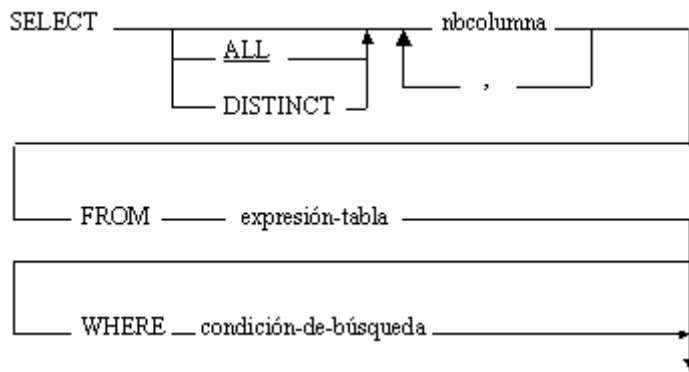
- el **DML** (Data Manipulation Language), **lenguaje de manipulación** de datos, nos permite recuperar los datos almacenados en la base de datos y también incluye órdenes para permitir al usuario actualizar la base de datos añadiendo nuevos datos, suprimiendo datos antiguos o modificando datos previamente almacenados.

## Características del lenguaje

Una sentencia SQL es como una **frase** (escrita en **inglés**) con la que decimos **lo que queremos obtener y de dónde obtenerlo**.

Todas las sentencias empiezan con un **verbo** (palabra reservada que indica la acción a realizar), seguido del resto de **cláusulas**, algunas **obligatorias** y otras **opcionales** que completan la frase. Todas las sentencias siguen una **sintaxis** para que se puedan ejecutar correctamente, para describir esa sintaxis utilizaremos un **diagrama sintáctico** como el que se muestra a continuación.

## Cómo interpretar un diagrama sintáctico



Las palabras que aparecen en mayúsculas son palabras reservadas. Se tienen que poner tal cual y no se pueden utilizar para otro fin, por ejemplo, en el diagrama de la figura tenemos las palabras reservadas **SELECT**, **ALL**, **DISTINCT**, **FROM**, **WHERE**.

Las palabras en minúsculas son variables que el usuario deberá sustituir por un dato concreto. En el diagrama tenemos *nbcolumna*, *expresión-tabla* y *condición-de-búsqueda*.

Una sentencia válida se construye siguiendo la línea a través del diagrama hasta el punto que marca el final. Las líneas se siguen de **izquierda a derecha y de arriba abajo**. Cuando se quiere alterar el orden normal se indica con una **flecha**.

¿Cómo se interpretaría el diagrama sintáctico de la figura?

Hay que empezar por la palabra **SELECT**, después puedes poner **ALL** o bien **DISTINCT** o nada, a continuación un nombre de columna, o varios separados por comas, a continuación la palabra **FROM** y una expresión-tabla, y por último de forma opcional puedes incluir la cláusula **WHERE** con una condición-de-búsqueda.

Por ejemplo:

**SELECT ALL col1,col2,col3 FROM mitabla**

**SELECT col1,col2,col3 FROM mitabla**

**SELECT DISTINCT col1 FROM mitabla**

**SELECT col1,col2 FROM mitabla WHERE col2 = 0**

Todas estas sentencias se podrían escribir y no darían lugar a errores sintácticos.

Cuando una palabra opcional está **subrayada**, esto indica que ese es el **valor por defecto** (el valor que se asume si no se pone nada). En el ejemplo anterior las dos primeras sentencias son equivalentes (en el diagrama **ALL** aparece subrayada).

## Cómo se crea una sentencia SQL en ACCESS

Este manual está basado en el SQL del motor de base de datos que utiliza el Access2000, el **Microsoft Jet 4.x**, para que los ejemplos y ejercicios se puedan ejecutar y probar.

Para crear y después ejecutar una sentencia SQL en Access, lo fácil es utilizar **la ventana SQL de las consultas**.


Para **crear una consulta de selección**, seguir los siguientes pasos:

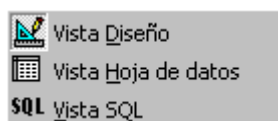
Abrir la base de datos donde se encuentra la consulta a crear.

Hacer clic sobre el objeto **Consulta** que se encuentra a la izquierda de la ventana de la base de datos.


Hacer clic sobre el botón **Nuevo**  de la ventana de la base de datos.

Seleccionar **Vista Diseño**. Como no queremos utilizar el generador de consultas sino escribir nuestras propias sentencias SQL, no agregamos ninguna tabla. Aparecerá la ventana de diseño de consultas.

Hacer clic sobre el botón , este botón es el que permite elegir la vista de la consulta, puede adoptar una de estas tres formas



Una vez escrita sólo nos queda ver si está bien hecha.

Hacer clic sobre el botón  de la barra de herramientas para **ejecutar** la sentencia.

Si nos hemos equivocado a la hora de escribir la sintaxis, Access nos saca un **mensaje de error** y muchas veces el cursor se queda posicionado en la palabra donde ha saltado el error. Ojo, a veces el error está antes o después de donde se ha quedado el cursor.

***Si no saca ningún mensaje de error, esto quiere decir que la sentencia respeta la sintaxis definida, pero esto no quiere decir que la sentencia esté bien, puede que no obtenga lo que nosotros queremos, en este caso habrá que rectificar la sentencia.***

# Las consultas simples

## Objetivo

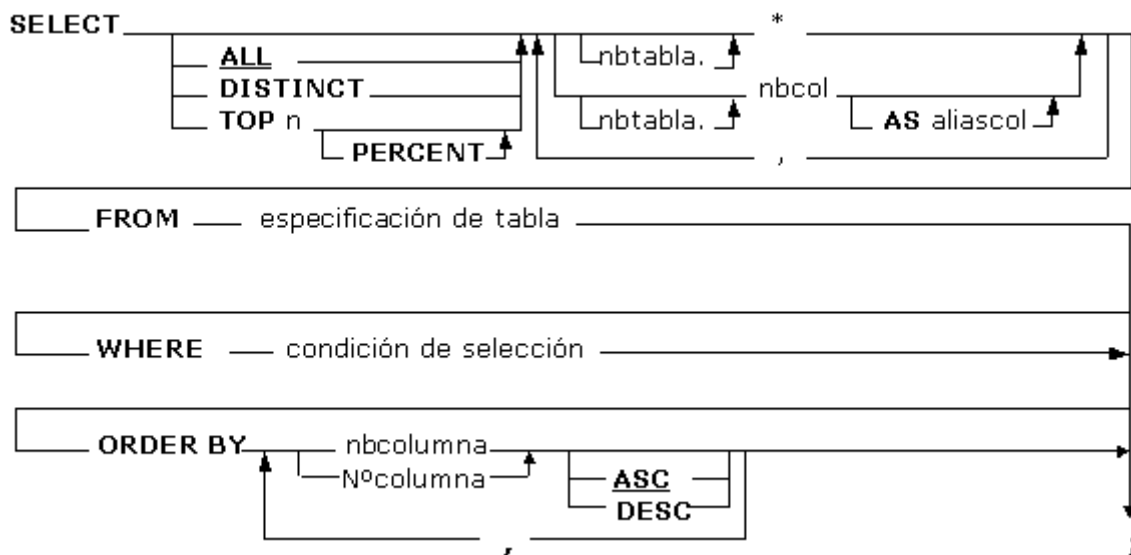
Empezaremos por estudiar la sentencia **SELECT**, que permite **recuperar datos de** una o varias **tablas**. La sentencia SELECT es con mucho la más compleja y potente de las sentencias SQL. Empezaremos por ver las **consultas más simples, basadas en una sola tabla**.

Esta sentencia forma parte del DML (lenguaje de manipulación de datos), en este tema veremos cómo **seleccionar columnas** de una tabla, cómo **seleccionar filas** y cómo obtener las **filas ordenadas** por el criterio que queramos.

El resultado de la consulta es una **tabla lógica**, porque no se guarda en el disco sino que está en memoria y cada vez que ejecutamos la consulta se vuelve a calcular.

Cuando ejecutamos la consulta se visualiza el resultado en forma de tabla con columnas y filas, pues en la SELECT tenemos que indicar qué columnas queremos que tenga el resultado y qué filas queremos seleccionar de la tabla origen.

## Sintaxis de la sentencia SELECT (consultas simples)

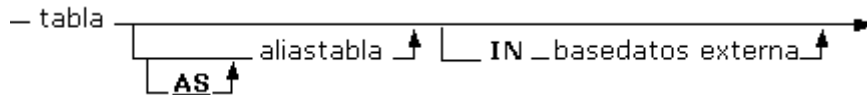


## La tabla origen - FROM -

Con la cláusula **FROM** indicamos **en qué tabla** tiene que **buscar la información**. En este capítulo de consultas simples el resultado se obtiene de una única tabla. La sintaxis de la cláusula es:

### FROM especificación de tabla

Una especificación de tabla puede ser el nombre de una consulta guardada (las que aparecen en la ventana de base de datos), o el nombre de una tabla que a su vez puede tener el siguiente formato:



● **Aliastabla** es un nombre de **alias**, es como un **segundo nombre** que asignamos a la **tabla**. Si en una consulta definimos un alias para la tabla, esta se deberá nombrar utilizando ese nombre y no su nombre real, además **ese nombre sólo** es **válido en la consulta** donde se define. El alias se suele emplear en consultas basadas en más de una tabla que veremos en el tema siguiente. La palabra **AS** que se puede poner delante del nombre de alias es opcional y es el valor por defecto por lo que no tienen ningún efecto.

Ejemplo: **SELECT .....FROM oficinas ofi** ; es equivalente a **SELECT .....FROM oficinas AS ofi** esta sentencia me indica que se van a buscar los datos en la tabla *oficinas* que queda renombrada en esta consulta con *ofi*.

● En una **SELECT** podemos utilizar tablas que no están definidas en la base de datos (siempre que tengamos los permisos adecuados claro), si la tabla no está en la base de datos activa, debemos indicar en qué base de datos se encuentra con la cláusula **IN**.

En la cláusula **IN** el nombre de la base de datos debe incluir el **camino completo**, la **extensión** (.mdb), y estar entre **comillas simples**.

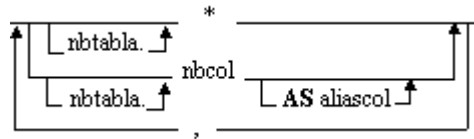
Supongamos que la tabla empleados estuviese en otra base de datos llamada *otra* en la carpeta *c:\mis documentos\*, habría que indicarlo así:

**SELECT \***  
**FROM empleados IN 'c:\mis documentos\otra.mdb'**

Generalmente tenemos las tablas en la misma base de datos y no hay que utilizar la cláusula **IN**.

## Selección de columnas

La lista de **columnas** que queremos que **aparezcan** en el **resultado** es lo que llamamos **lista de selección** y se especifica delante de la cláusula **FROM**.



### ● Utilización del \*

Se utiliza el asterisco \* en la lista de selección para indicar '**todas las columnas de la tabla**'.

Tiene dos **ventajas**:

- 1.- Evitar nombrar las columnas una a una (es más corto).
  - 2.- Si añadimos una columna nueva en la tabla, esta nueva columna saldrá sin tener que modificar la consulta.
- Se puede combinar el \* con el nombre de una tabla (ej. oficinas.\*), pero esto se utiliza más cuando el origen de la consulta son dos tablas.

**SELECT \* FROM oficinas**

o bien

**SELECT oficinas.\* FROM oficinas**

Lista todos los datos de las oficinas

### ● Columnas de la tabla origen

Las columnas se pueden especificar mediante su **nombre simple** (nbcol) o su **nombre cualificado** (nbtbla.nbcol, el nombre de la columna precedido del nombre de la tabla que contiene la columna y separados por un punto). El nombre cualificado se puede emplear siempre que queramos y es obligatorio en algunos casos que veremos más adelante.

Cuando el **nombre** de la **columna** o de la tabla **contiene espacios en blanco**, hay que poner el nombre **entre corchetes** [ ] y además el número de espacios en blanco debe coincidir. Por ejemplo [codigo de cliente] no es lo mismo que [ codigo de cliente] (el segundo lleva un espacio en blanco delante de código)

Ejemplos :

**SELECT nombre, oficina,  
contrato  
FROM ofiventas**

Lista el nombre, oficina, y fecha de contrato de todos los empleados.

**SELECT idfab, idproducto,  
descripcion, precio  
FROM productos**

Lista una tarifa de productos

### ● Alias de columna.

Cuando se visualiza el resultado de la consulta, normalmente las columnas toman el nombre que tiene la columna en la tabla, si queremos cambiar ese nombre lo podemos hacer definiendo un alias de columna mediante la cláusula **AS** será el nombre que aparecerá como **título de la columna**.

Ejemplo:

```
SELECT idfab AS fabricante,  
idproducto, descripcion  
FROM productos
```

Como título de la primera columna aparecerá fabricante en vez de idfab

### ● Columnas calculadas.

Además de las columnas que provienen directamente de la tabla origen, una consulta SQL puede incluir **columnas calculadas** cuyos valores se calculan a partir de los valores de los datos almacenados. Para solicitar una columna calculada, se especifica en la lista de selección una **expresión** en vez de un nombre de columna. La expresión puede contener sumas, restas, multiplicaciones y divisiones, concatenación & , paréntesis y también funciones predefinidas).

Ejemplos:

```
SELECT ciudad, región,  
(ventas-objetivo) AS  
superavit  
FROM oficinas
```

Lista la ciudad, región y el superavit de cada oficina.

```
SELECT idfab, idproducto,  
descripcion, (existencias * precio) AS  
valoracion  
FROM productos
```

De cada producto obtiene su fabricante, idproducto, su descripción y el valor del inventario

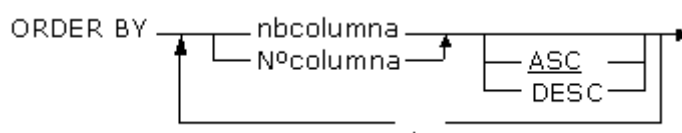
```
SELECT nombre,  
MONTH(contrato),  
YEAR(contrato)  
FROM repventas
```

Lista el nombre, mes y año del contrato de cada vendedor.  
La función MONTH() devuelve el mes de una fecha  
La función YEAR() devuelve el año de una fecha

```
SELECT oficina, 'tiene ventas de ', ventas  
FROM oficinas
```

Lista las ventas en cada oficina con el formato: 22 tiene ventas de 186,042.00

## Ordenación de las filas - ORDER BY -



Para **ordenar** las **filas** del resultado de la consulta, tenemos la cláusula **ORDER BY**.

Con esta cláusula se altera el orden de visualización de las filas de la tabla pero en ningún caso se modifica el orden de las filas dentro de la tabla. La tabla no se modifica.

● Podemos indicar la columna por la que queremos ordenar utilizando su **nombre de columna** (nbcolumna) o utilizando su **número de orden** que ocupa en la lista de selección (Nºcolumna).

Ejemplo:

```
SELECT nombre, oficina, contrato  
FROM empleados  
ORDER BY oficina
```

es equivalente a

```
SELECT nombre, oficina, contrato  
FROM empleados  
ORDER BY 2
```

● Por defecto el **orden** será **ascendente (ASC)** (de menor a mayor si el campo es numérico, por orden alfabético si el campo es de tipo texto, de anterior a posterior si el campo es de tipo fecha/hora, etc.

Ejemplos:

```
SELECT nombre,  
numemp, oficinarep  
FROM empleados  
ORDER BY nombre
```

Obtiene un listado alfabético de los empleados.

```
SELECT nombre,  
numemp, contrato  
FROM empleados  
ORDER BY contrato
```

Obtiene un listado de los empleados por orden de antigüedad en la empresa (los de más antigüedad aparecen primero).

```
SELECT nombre,  
numemp,ventas  
FROM empleados  
ORDER BY ventas
```

Obtiene un listado de los empleados ordenados por volumen de ventas sacando los de menores ventas primero.

● Si queremos podemos alterar ese orden utilizando la cláusula **DESC** (DESCendente), en este caso el orden será el inverso al ASC.

Ejemplos:

```
SELECT nombre, numemp,  
contrato  
FROM empleados  
ORDER BY contrato DESC
```

Obtiene un listado de los empleados por orden de antigüedad en la empresa empezando por los más recientemente incorporados.

```
SELECT nombre, numemp,ventas  
FROM empleados  
ORDER BY ventas des
```

Obtiene un listado de los empleados ordenados por volumen de ventas sacando primero los de mayores ventas.



🟡 También podemos **ordenar** por **varias columnas**, en este caso se indican las columnas separadas por comas.  
Se ordenan las filas por la primera columna de ordenación, para un mismo valor de la primera columna, se ordenan por la segunda columna, y así sucesivamente.

La cláusula **DESC** o **ASC** se puede indicar para cada columna y así utilizar una ordenación distinta para cada columna. Por ejemplo ascendente por la primera columna y dentro de la primera columna, descendente por la segunda columna.

Ejemplos:

```
SELECT region, ciudad,  
ventas  
FROM oficinas  
ORDER BY region, ciudad
```

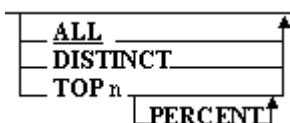
Muestra las ventas de cada oficina ,  
ordenadas por orden alfabético de región y  
dentro de cada región por ciudad.

```
SELECT region, ciudad,  
(ventas - objetivo) AS  
superavit  
FROM oficinas  
ORDER BY region, 3 DESC
```

Lista las oficinas clasificadas por región y  
dentro de cada región por superavit de modo  
que las de mayor superavit aparezcan las  
primeras.

## Selección de filas

A continuación veremos las cláusulas que nos permiten indicar **qué filas** queremos **visualizar**.



**WHERE** — condición-de-selección —

## Las cláusulas **DISTINCT** / **ALL**

Al incluir la cláusula **DISTINCT** en la **SELECT**, se **eliminan** del resultado las **repeticiones** de filas. Si por el contrario queremos que aparezcan todas las filas **incluidas las duplicadas**, podemos incluir la cláusula **ALL** o nada, ya que **ALL** es el valor que **SQL** asume por defecto.

Por ejemplo queremos saber los códigos de los directores de oficina.

```
SELECT dir FROM oficinas  
SELECT ALL dir FROM oficinas
```

Lista los códigos de los directores de las  
oficinas. El director 108 aparece en cuatro  
oficinas, por lo tanto aparecerá cuatro  
veces en el resultado de la consulta.

**SELECT DISTINCT dir FROM  
oficinas**

En este caso el valor 108 aparecerá una sola vez ya que le decimos que liste los distintos valores de directores.

## La cláusula TOP

La cláusula **TOP** permite **sacar** las **n primeras "filas"** de la tabla origen. No diferencia filas que sean iguales. Si pido los 25 primeros valores pero, el que hace la posición 26 es el mismo valor que el de la 25, entonces devolverá 26 registros en vez de 25. Siempre se guía por la columna de ordenación, la que aparece en la cláusula ORDER BY o en su defecto la clave principal de la tabla.

Por ejemplo queremos saber los dos empleados más antiguos de la empresa.

**SELECT TOP 2 numemp,  
nombre  
FROM empleado  
ORDER BY contrato**

Lista el código y nombre de los empleados ordenándolos por fecha de contrato, sacando únicamente los dos primeros (serán los dos más antiguos).

**SELECT TOP 3 numemp,  
nombre  
FROM empleado  
ORDER BY contrato**

En este caso tiene que sacar los tres primeros, pero si nos fijamos en las fechas de contrato tenemos 20/10/86, 10/12/86, 01/03/87, 01/03/87, la tercera fecha es igual que la cuarta, en este caso sacará estas cuatro filas en vez de tres, y sacaría todas las filas que tuviesen el mismo valor que la tercera fecha de contrato.

El número de filas que queremos visualizar se puede expresar con un número entero o como un **porcentaje** sobre el **número total de filas** que se recuperarían sin la cláusula TOP. En este último caso utilizaremos la cláusula **TOP n PERCENT** (porcentaje en inglés).

**SELECT TOP 20 PERCENT  
nombre  
FROM empleado  
ORDER BY contrato**

Lista el nombre de los empleados ordenándolos por fecha de contrato, sacando únicamente un 20% del total de empleados. Como tenemos 10 empleados, sacará los dos primeros, si tuviésemos 100 empleados sacaría los 20 primeros.

## La cláusula WHERE

**WHERE** — condición-de-selección —

La cláusula **WHERE** selecciona únicamente las **filas** que **cumplan** la **condición de selección** especificada.

En la consulta sólo aparecerán las filas para las cuales la condición es verdadera (TRUE). Los valores nulos (NULL) no se incluyen en las filas del resultado. La **condición de selección** puede ser cualquier **condición válida** o **combinación de condiciones** utilizando los operadores **NOT** (no) **AND** (y) y **OR** (ó). En ACCESS2000 una cláusula WHERE puede contener hasta 40 expresiones vinculadas por operadores lógicos AND y OR.

Para empezar veamos un ejemplo sencillo:

```
SELECT nombre  
FROM empleados  
WHERE oficina = 12
```

Lista el nombre de los empleados de la oficina 12.

```
SELECT nombre  
FROM empleados  
WHERE oficina = 12 AND  
edad > 30
```

Lista el nombre de los empleados de la oficina 12 que tengan más de 30 años.  
(oficina igual a 12 **y** edad mayor que 30)

## Condiciones de selección

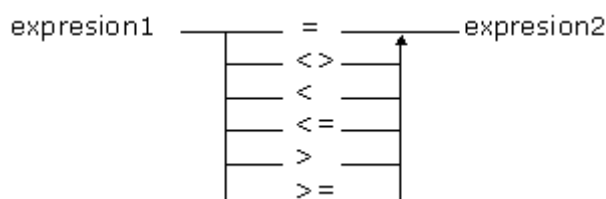
Las **condiciones de selección** son las condiciones que pueden aparecer en la cláusula **WHERE**.

En SQL tenemos cinco condiciones básicas:

- el **test de comparación**
- el **test de rango**
- el **test de pertenencia a un conjunto**
- el **test de valor nulo**
- el **test de correspondencia con patrón**.

### 🟡 El **test de comparación**.

Compara el valor de una expresión con el valor de otra.  
La sintaxis es la siguiente:



- = igual que
- <> distinto de
- < menor que
- <= menor o igual
- > mayor que
- >= mayor o igual

```
SELECT numemp, nombre  
FROM empleados  
WHERE ventas > cuota
```

Lista los empleados cuyas ventas superan su cuota

```
SELECT numemp, nombre
FROM empleados
WHERE contrato <
#01/01/1988#
```

Lista los empleados contratados antes del año 88 (cuya fecha de contrato sea anterior al 1 de enero de 1988).

¡¡Ojo!!, las fechas entre almohadillas # # deben estar con el formato mes,dia,año aunque tengamos definido otro formato para nuestras fechas.

```
SELECT numemp, nombre
FROM empleados
WHERE YEAR(contrato) <
1988
```

Este ejemplo obtiene lo mismo que el anterior pero utiliza la función `year()`. Obtiene los empleados cuyo año de la fecha de contrato sea menor que 1988.

```
SELECT oficina
FROM oficinas
WHERE ventas < objetivo *
0.8
```

Lista las oficinas cuyas ventas estén por debajo del 80% de su objetivo.  
Hay que utilizar siempre el punto decimal aunque tengamos definida la coma como separador de decimales.

```
SELECT oficina
FROM oficinas
WHERE dir = 108
```

Lista las oficinas dirigidas por el empleado 108.

● **Test de rango (BETWEEN).**

Examina si el **valor** de la expresión está **comprendido entre** los **dos valores** definidos por exp1 y exp2.

Tiene la siguiente sintaxis:

expression            BETWEEN — exp1 —AND— exp2→  
                  └─NOT─┐

```
SELECT numemp, nombre
FROM empleados
WHERE ventas BETWEEN
100000 AND 500000
```

Lista los empleados cuyas ventas estén comprendidas entre 100.000 y 500.00

```
SELECT numemp, nombre  
FROM empleados  
WHERE (ventas >= 100000)  
AND (ventas <= 500000)
```

Obtenemos lo mismo que en el ejemplo anterior. Los paréntesis son opcionales.

### 🟡 Test de pertenencia a conjunto (IN)

Examina si el **valor** de la expresión es uno de los valores **incluidos en la lista de valores**.

Tiene la siguiente sintaxis:

Expresion — IN — (valor) →  
                                  ↑  
                                  ,    

**SELECT numemp, nombre,  
oficina  
FROM empleados  
WHERE oficina IN (12,14,16)**

Lista los empleados de las oficinas 12, 14 y 16

**SELECT numemp, nombre  
FROM empleados  
WHERE (oficina = 12) OR  
(oficina = 14) OR (oficina = 16)**

Obtenemos lo mismo que en el ejemplo anterior. Los paréntesis son opcionales.

### 🟡 Test de valor nulo (IS NULL)

Una condición de selección puede dar como resultado el valor verdadero TRUE, falso FALSE o nulo NULL.

Cuando una **columna** que interviene en una condición de selección **contiene** el **valor nulo**, el **resultado** de la condición no es verdadero ni falso, sino **nulo**, **sea cual sea el test** que se haya utilizado. Por eso si queremos listar las filas que tienen valor en una determinada columna, no podemos utilizar el test de comparación, ya que la condición oficina = null devuelve el valor nulo sea cual sea el valor contenido en oficina. Si queremos preguntar si una columna contiene el valor nulo debemos utilizar un **test especial**, el test de valor nulo. Tiene la siguiente sintaxis:

nbcolumna — IS — NOT — NULL —→  
                                  ↑  
                                  NOT

Ejemplos:

**SELECT oficina, ciudad  
FROM oficinas  
WHERE dir IS NULL**

Lista las oficinas que no tienen director.

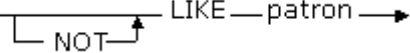
**SELECT numemp, nombre  
FROM empleados  
WHERE oficina IS NOT NULL**

Lista los empleados asignados a alguna oficina (los que tienen un valor en la columna oficina).

## 🟡 Test de correspondencia con patrón (LIKE)

Se utiliza cuando queremos **utilizar caracteres comodines** para formar el valor con el comparar.

Tiene la siguiente sintaxis:

nbcolumna 

Los comodines más usados son los siguientes:

**?** representa un carácter cualquiera

**\*** representa cero o más caracteres

**#** representa un dígito cualquiera (0-9)

Ejemplos:

**SELECT numemp, nombre**  
**FROM empleados**  
**WHERE nombre LIKE 'Luis\*'**

Lista los empleados cuyo nombre empiece por Luis (Luis seguido de cero o más caracteres).

**SELECT numemp, nombre**  
**FROM empleados**  
**WHERE nombre LIKE '\*Luis\*'**

Lista los empleados cuyo nombre contiene Luis, en este caso también saldría los empleados José Luis (cero o más caracteres seguidos de LUIS y seguido de cero o más caracteres).

**SELECT numemp, nombre**  
**FROM empleados**  
**WHERE nombre LIKE '??a\*'**

Lista los empleados cuyo nombre contenga una a como tercera letra (dos caracteres, la letra a, y cero o más caracteres).

# Las consultas multitabla

## Introducción

En este tema vamos a estudiar las **consultas multitabla** llamadas así porque están **basadas** en **más de una tabla**.

El SQL de Microsoft Jet 4.x soporta dos grupos de consultas multitabla:

- la **unión** de tablas
- la **composición** de tablas

## La unión de tablas

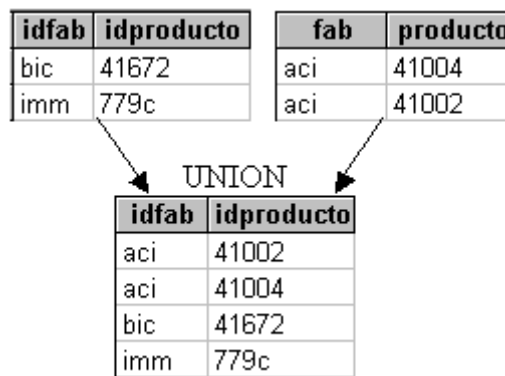
Esta operación se utiliza cuando tenemos **dos tablas** con las **mismas columnas** y queremos obtener una **nueva tabla** con las **filas de la primera y las filas de la segunda**. En este caso la tabla resultante tiene las mismas columnas que la primera tabla (que son las mismas que las de la segunda tabla).

Por ejemplo, tenemos una tabla de libros nuevos y una tabla de libros antiguos, y queremos una lista con todos los libros que tenemos. En este caso las dos tablas tienen las mismas columnas, lo único que varía son las filas, además queremos obtener una lista de libros (las columnas de una de las tablas) con las filas que están tanto en libros nuevos como las que están en libros antiguos, en este caso utilizaremos este tipo de operación.

Cuando hablamos de tablas, pueden ser **tablas reales** almacenadas en la base de datos, **o tablas lógicas** (resultados de una consulta). Esto nos permite utilizar la operación con más frecuencia, ya que pocas veces tenemos en una base de datos tablas idénticas en cuanto a columnas. El **resultado** es siempre una **tabla lógica**.

Por ejemplo, queremos en un sólo listado los productos cuyas existencias sean iguales a cero, y también los productos que aparecen en pedidos del año 90. En este caso tenemos unos productos en la tabla de productos y los otros en la tabla de pedidos, las tablas no tienen las mismas columnas y por tanto no se puede hacer una unión de ellas, pero lo que interesa realmente es el identificador del producto (idfab,idproducto), luego por una parte sacamos los códigos de los productos con existencias cero (con una consulta), por otra parte los códigos de los productos que aparecen en pedidos del año 90 (con otra consulta), y luego unimos estas dos tablas lógicas.

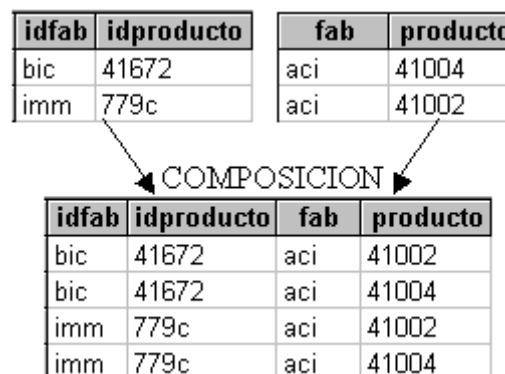
El operador que permite realizar esta operación es el operador **UNION**.



## La composición de tablas

La composición de tablas consiste en **concatenar** filas de una tabla con filas de otra. En este caso obtenemos una tabla con las **columnas** de la **primera tabla unidas** a las **columnas** de la **segunda tabla**, y las filas de la tabla resultante son **concatenaciones** de **filas** de la **primera tabla con** filas de la **segunda tabla**.

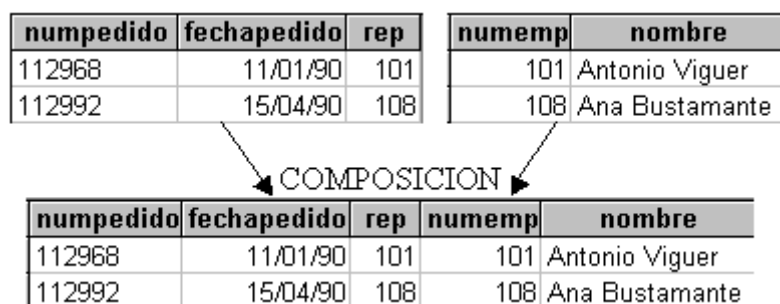
El ejemplo anterior quedaría de la siguiente forma con la composición:



A diferencia de la unión, la composición permite obtener una fila con datos de las dos tablas, esto es muy útil cuando queremos visualizar filas cuyos datos se encuentran en dos tablas.

Por ejemplo queremos listar los pedidos con el nombre del representante que ha hecho el pedido, pues los datos del pedido los tenemos en la tabla de pedidos pero el nombre del representante está en la tabla de empleados y además queremos que aparezcan en la misma línea; en este caso necesitamos componer las dos tablas (Nota: en el ejemplo expuesto a continuación, hemos seleccionado las filas que nos interesan).





Existen distintos tipos de composición, aprenderemos a utilizarlos todos y a elegir el tipo más apropiado a cada caso.

Los **tipos de composición** de tablas son:

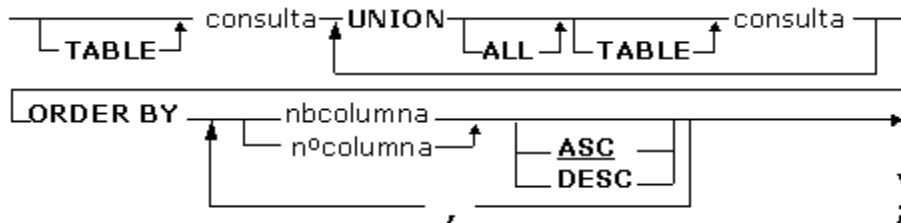
- . El **producto cartesiano**
- . El **INNER JOIN**
- . El **LEFT / RIGHT JOIN**

## El operador UNION



Como ya hemos visto en la página anterior, el operador **UNION** sirve para obtener a partir de dos **tablas** con las **mismas columnas**, una nueva tabla con las **filas** de la **primera** y las **filas** de la **segunda**.

La sintaxis es la siguiente:



● **Consulta** puede ser un **nombre de tabla**, un **nombre de consulta** (en estos dos casos el nombre debe estar precedido de la palabra **TABLE**), o una **sentencia SELECT** completa (en este caso no se puede poner **TABLE**). La sentencia **SELECT** puede ser cualquier sentencia **SELECT** con la única restricción de que no puede contener la cláusula **ORDER BY**.

Después de la primera consulta viene la palabra **UNION** y a continuación la segunda consulta. La segunda consulta sigue las mismas reglas que la primera consulta.

● Las dos consultas deben tener el **mismo número** de **columnas** pero las columnas pueden **llamarse** de **diferente** forma. No deben ser de **tipos** de datos **distintos**.

● Las **columnas** del **resultado** se **llaman** como las de la **primera consulta**.

● Por **defecto** la unión **no incluye filas repetidas**, si alguna fila está en las dos tablas, sólo aparece una vez en el resultado.

Si **queremos** que aparezcan todas las filas incluso las **repeticiones** de **filas**, incluimos la palabra **ALL** (*todo* en inglés).

El empleo de **ALL** tienen una **ventaja**, la consulta **se ejecutará más rápidamente**. Puede que la diferencia no se note con tablas pequeñas, pero si tenemos tablas con muchos registros (filas) la diferencia puede ser notable.

● Se puede **unir más** de **dos** tablas, para ello después de la segunda consulta **repetimos** la palabra **UNION**... y así sucesivamente.

● También podemos indicar que queremos el **resultado ordenado** por algún criterio, en este caso se incluye la cláusula **ORDER BY** que ya vimos en el tema anterior. La cláusula **ORDER BY** se escribe **después** de la **última consulta**, al final de la sentencia; para indicar las **columnas de ordenación** podemos utilizar su **número de orden** o el **nombre de la columna**, en este último caso se deben de utilizar los nombres de columna de la **primera consulta** ya que son los que se van a utilizar para nombrar las columnas del resultado.

Para ilustrar la operación vamos a realizar el ejercicio visto en la página anterior, vamos a obtener los códigos de los productos que tienen existencias iguales a cero o que aparezcan en pedidos del año 90.

```
SELECT idfab,idproducto
FROM productos
WHERE existencias = 0
UNION ALL
SELECT fab,producto
FROM pedidos
WHERE year(fechapedido) = 1990
ORDER BY idproducto
```

-----Primera consulta(tabla 1) o bien Segunda consulta(tabla 2)

```
TABLE [existencias
cero]
UNION ALL
TABLE [pedidos 90]
ORDER BY idproducto
```

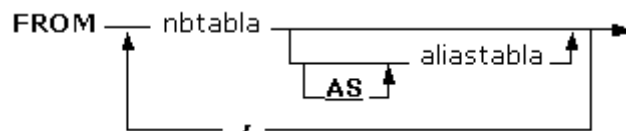
Se ha incluido la cláusula **ALL** porque no nos importa que salgan filas repetidas.

Se ha incluido **ORDER BY** para que el resultado salga ordenado por idproducto, observar que hemos utilizado el nombre de la columna de la primera **SELECT**, también podíamos haber puesto **ORDER BY 2** pero no **ORDER BY producto** (es el nombre de la columna de la segunda tabla).

Para el 2º caso hemos creado una consulta llamada *existencias cero* con la primera **SELECT**, y una consulta llamada *pedidos 90* con la segunda **SELECT**. Observar que los nombres de las consultas están entre corchetes porque contienen espacios en blanco, y que en este caso hay que utilizar **TABLE**.

## El producto cartesiano

El producto cartesiano es un tipo de composición de tablas. Aplicando el producto cartesiano a dos tablas se obtiene una tabla con las **columnas** de la **primera tabla unidas** a las **columnas** de la **segunda tabla**, y las filas de la tabla resultante son **todas las posibles concatenaciones** de **filas** de la **primera tabla** con **filas** de la **segunda tabla**. La sintaxis es la siguiente:



● El **producto cartesiano** se indica **poniendo** en la **FROM** las **tablas** que queremos componer **separadas por comas**, podemos obtener así el producto cartesiano de dos, tres, o más tablas.

● **nbtabela** puede ser un **nombre de tabla** o un **nombre de consulta**. Si todas las tablas están en una base de datos externa, añadiremos la cláusula **IN basedatosexterna después** de la **última tabla**. Pero para mejorar el rendimiento y facilitar el uso, se recomienda utilizar una tabla vinculada en lugar de la cláusula IN.

● Hay que tener en cuenta que el producto cartesiano **obtiene** todas las **posibles combinaciones** de filas, por lo tanto, si tenemos dos tablas de 100 registros cada una, el resultado tendrá 100x100 filas. Si el producto lo hacemos de estas dos tablas con una tercera de 20 filas, el resultado tendrá 200.000 filas (100x100x20) y estamos hablando de tablas pequeñas. Se ve claramente que el producto cartesiano es una **operación costosa**, sobre todo si operamos con más de dos tablas o con tablas voluminosas.

● Se puede **componer** una **tabla consigo misma**. En este caso es **obligatorio** utilizar un nombre de **alias**, por lo menos para una de las dos. Por ejemplo: **SELECT \* FROM empleados, empleados emp**

En este ejemplo obtenemos el producto cartesiano de la tabla de empleados con ella misma. Todas las posibles combinaciones de empleados con empleados.

● Para ver cómo funciona el producto cartesiano cogemos las consultas [existencias cero] y [pedidos 90] creadas en la página anterior, y creamos una consulta que halle el producto cartesiano de las dos.

**SELECT \***  
**FROM [existencias**  
**cero],[pedidos 90]**

obtenemos  
la  
siguiente  
tabla:

idfab	idproducto	fab	producto
bic	41672	aci	41002
bic	41672	aci	41004
imm	779c	aci	41002
imm	779c	aci	41004

Se observa que tenemos las dos filas de la primera consulta combinadas con las dos filas de la segunda.

● Esta operación **no** es de las **más utilizadas**. Normalmente cuando queremos componer dos tablas, es para añadir a las filas de una tabla una fila de la otra tabla. Por ejemplo, añadir a los pedidos los datos del cliente correspondiente, o los datos del representante. Esto equivaldría a un producto cartesiano con una selección de filas:

```
SELECT *  
FROM pedidos,clientes  
WHERE pedidos.clie=clientes.numclie
```

*Combinamos todos los pedidos con todos los clientes pero luego seleccionamos los que cumplan que el código de cliente de la tabla de pedidos sea igual al código de cliente de la tabla de clientes, por lo tanto nos quedamos con los pedidos combinados con los datos del cliente correspondiente.*

● Las columnas que aparecen en la cláusula WHERE de nuestra consulta anterior se denominan **columnas de emparejamiento** ya que permiten emparejar las filas de las dos tablas. Las columnas de emparejamiento no tienen porqué estar incluidas en la lista de selección.

● Normalmente emparejamos tablas que están relacionadas entre sí y una de las columnas de emparejamiento será la clave principal. En este caso, cuando **una** de las **columnas de emparejamiento** tiene un **índice** definido, es más eficiente utilizar otro tipo de composición, el **INNER JOIN**.

## EL INNER JOIN

El **INNER JOIN** es otro tipo de composición de tablas. Permite **emparejar filas** de distintas tablas de forma **más eficiente** que con el producto cartesiano **cuando** una de las **columnas de emparejamiento** está **indexada**. Ya que en vez de hacer el producto cartesiano completo y luego seleccionar la filas que cumplen la condición de emparejamiento, para cada fila de una de las tablas **busca directamente** en la otra tabla **las filas que** cumplen la condición, con lo cual **se emparejan** sólo las filas que luego aparecen en el resultado.

La sintaxis es la siguiente:

```
FROM – tabla1 – INNER JOIN – tabla2 – ON – tabla1.col1 – comp – tabla2.col2
```

Ejemplo:

```
SELECT *  
FROM pedidos INNER JOIN clientes ON pedidos.clie = clientes.numclie
```

● *tabla1* y *tabla2* son **especificaciones de tabla** (nombre de tabla con alias o no, nombre de consulta guardada), de las tablas cuyos registros se van a combinar.

Pueden ser las dos la **misma tabla**, en este caso es **obligatorio** definir al menos un **alias** de tabla.

● *col1*, *col2* son las **columnas de emparejamiento**.

Observar que dentro de la cláusula **ON** los nombres de **columna** deben ser **nombres cualificados** (lleva delante el nombre de la tabla y un punto).

● Las **columnas de emparejamiento** deben contener la **misma clase de datos**, las dos de tipo texto, de tipo fecha etc. los campos numéricos deben ser de tipos similares. Por ejemplo, se puede combinar campos Auto Numérico y Long puesto que son tipos similares, sin embargo, no se puede combinar campos de tipo Simple y Doble. Además las columnas no pueden ser de tipo Memo ni OLE.

● **comp** representa cualquier operador de **comparación** ( =, <, >, <=, >=, o <> ) y se utiliza para establecer la condición de emparejamiento.

Se pueden definir **varias condiciones** de emparejamiento **unidas** por los operadores **AND** y **OR** poniendo cada condición entre **paréntesis**.

Ejemplo:

```
SELECT *  
FROM pedidos INNER JOIN productos ON (pedidos.fab =  
productos.idfab) AND (pedidos.producto = productos.idproducto)
```

● Se pueden **combinar más de dos tablas**  
En este caso hay que **sustituir** en la sintaxis una **tabla** por un **INNER JOIN completo**.

Por ejemplo:

```
SELECT *  
FROM (pedidos INNER JOIN clientes ON pedidos.clie =  
clientes.numclie) INNER JOIN empleados ON pedidos.rep =  
empleados.numemp
```

-> Todo lo que esta dentro del parentesis funciona como si fuese una única tabla

En vez de *tabla1* hemos escrito un INNER JOIN completo, también podemos escribir:

```
SELECT *  
FROM clientes INNER JOIN (pedidos INNER JOIN empleados ON  
pedidos.rep = empleados.numemp) ON pedidos.clie = clientes.numclie
```

En este caso hemos sustituido *tabla2* por un INNER JOIN completo.

## EL LEFT JOIN y RIGHT JOIN

El **LEFT JOIN** y **RIGHT JOIN** son otro tipo de composición de tablas, también denominada **composición externa**. Son una extensión del **INNER JOIN**.

Las composiciones vistas hasta ahora (el **producto cartesiano** y el **INNER JOIN**) son **composiciones internas**, ya que todos los valores de las filas del resultado son valores que están en las tablas que se combinan.

Con una composición interna sólo se obtienen las filas que tienen al menos una fila de la otra tabla que cumpla la condición, veamos un ejemplo:

Queremos combinar los empleados con las oficinas para saber la ciudad de la oficina donde trabaja cada empleado, si utilizamos un producto cartesiano tenemos:

```
SELECT empleados.*, ciudad  
FROM empleados, oficinas  
WHERE empleados.oficina = oficinas.oficina
```

*Observar que hemos cualificado el nombre de columna oficina ya que ese nombre aparece en las dos tablas de la FROM.*

Con esta sentencia los **empleados** que **no tienen** una **oficina** asignada (un valor nulo en el campo oficina de la tabla empleados) **no aparecen en el resultado** ya que la condición **empleados.oficina = oficinas.oficina** será siempre nula para esos empleados.

Si utilizamos el **INNER JOIN**:

```
SELECT empleados.*, ciudad  
FROM empleados INNER JOIN oficinas ON empleados.oficina =  
oficinas.oficina
```

Nos pasa lo mismo, el empleado 110 tiene un valor nulo en el campo oficina y no aparecerá en el resultado.

En los casos en que **queremos** que **también aparezcan** las **filas que no tienen una fila coincidente** en la otra tabla, **utilizaremos** el **LEFT** o **RIGHT JOIN**.

🟡 La sintaxis del **LEFT JOIN** es la siguiente:

```
FROM— tabla1 — LEFT JOIN— tabla2 — ON—tabla1.col1—comp—tabla2.col2
```

La descripción de la sintaxis es la **misma** que la del **INNER JOIN** (ver página anterior), lo único que cambia es la palabra **INNER** por **LEFT** (**izquierda** en inglés).

Esta operación consiste en **añadir al resultado** del **INNER JOIN** las **filas** de la **tabla** de la **izquierda** que **no tienen correspondencia** en la otra tabla, y **rellenar** en esas filas los **campos** de la **tabla** de la **derecha** con **valores nulos**.



Ejemplo:

```
SELECT *  
FROM empleados LEFT JOIN oficinas ON empleados.oficina =  
oficinas.oficina
```

Con este ejemplo obtenemos una lista de los empleados con los datos de su oficina, y el empleado 110 que no tiene oficina aparece con sus datos normales y los datos de su oficina a nulos.

● La sintaxis del **RIGHT JOIN** es la siguiente:

```
FROM – tabla1 – RIGHT JOIN – tabla2 – ON – tabla1.col1 – comp – tabla2.col2
```

La sintaxis es la misma que la del **INNER JOIN** (ver página anterior), lo único que cambia es la palabra **INNER** por **RIGHT** (**derecha** en inglés).

Esta operación consiste en **añadir al resultado** del **INNER JOIN** las **filas** de la **tabla** de la **derecha** que **no tienen correspondencia** en la otra tabla, y **rellenar** en esas filas los **campos** de la **tabla** de la **izquierda** con **valores nulos**.

Ejemplo:

```
SELECT *  
FROM empleados RIGHT JOIN oficinas ON empleados.oficina =  
oficinas.oficina
```

Ahora obtenemos una lista de los empleados con los datos de su oficina, y además aparece una fila por cada oficina que no está asignada a ningún empleado con los datos del empleado a nulos.

● Una operación **LEFT JOIN** o **RIGHT JOIN** **se puede anidar** dentro de una operación **INNER JOIN**, pero una operación **INNER JOIN** **no** se puede **anidar dentro** de **LEFT JOIN** o **RIGHT JOIN**. Los anidamientos de **JOIN** de distinta naturaleza no funcionan siempre, a veces depende del orden en que colocamos las tablas, en estos casos lo mejor es probar y si no permite el anudamiento, cambiar el orden de las tablas (y por tanto de los **JOINS**) dentro de la cláusula **FROM**.

Por ejemplo podemos tener:

```
SELECT *  
FROM clientes INNER JOIN (empleados LEFT JOIN oficinas ON  
empleados.oficina = oficinas.oficina) ON clientes.repclie =  
empleados.numclie
```

Combinamos empleados con oficinas para obtener los datos de la oficina de cada empleado, y luego añadimos los clientes de cada representante, así obtenemos los clientes que tienen un representante asignado y los datos de la oficina del representante asignado.

Si hubiéramos puesto **INNER** en vez de **LEFT** no saldrían los clientes que

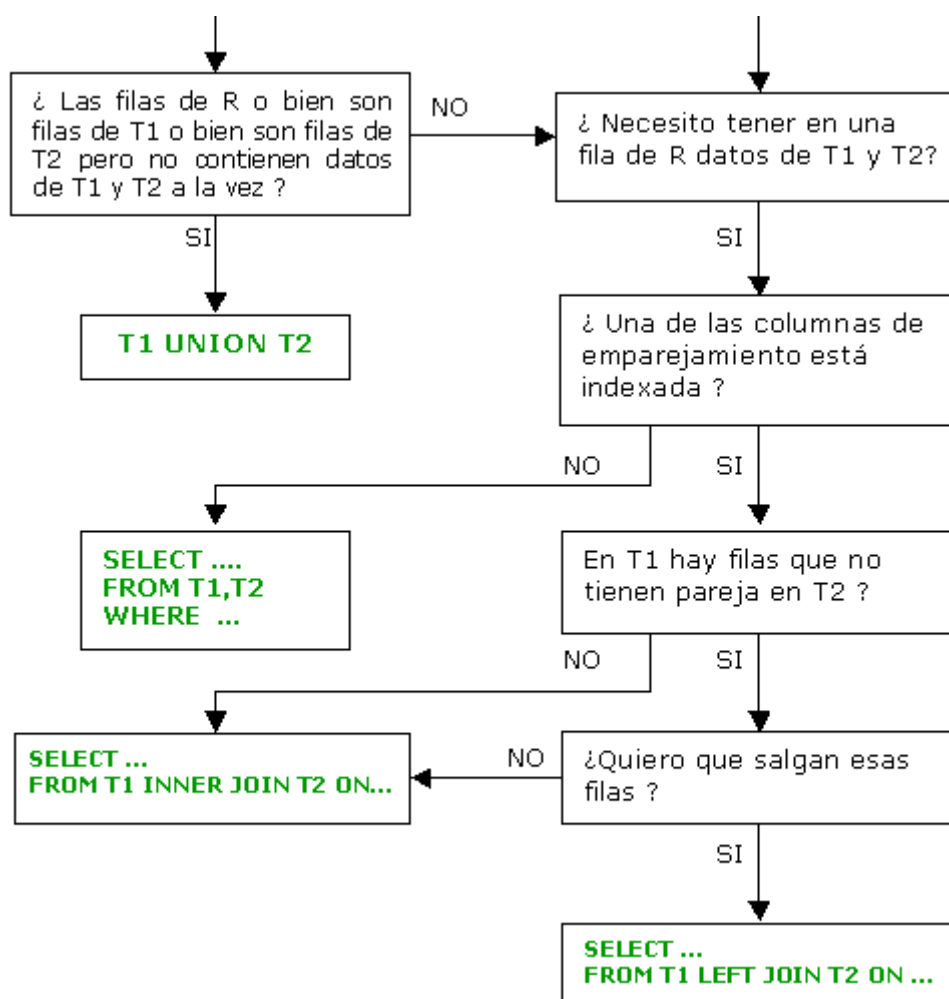
tienen el empleado 110 (porque no tiene oficina y por tanto no aparece en el resultado del **LEFT JOIN** y por tanto no entrará en el cálculo del **INNER JOIN** con clientes).

## Resumen de cuándo utilizar cada operación.

Para saber en cada caso qué tipo de operación se debe utilizar, a continuación tienes un gráfico que indica qué preguntas se tienen que hacer y según la respuesta, qué operación utilizar.

Para resumir hemos llamado T1 y T2 las tablas de las que queremos sacar los datos y R la tabla lógica que representa el resultado de consulta. T1 y T2 podrían ser tablas guardadas o consultas.

En la última parte cuando se pregunta "En T1 hay filas que no tienen pareja en T2", la pregunta se debe de interpretar como "en alguna de las tablas hay filas que no tienen pareja".



## Las consultas de resumen

### Introducción

En SQL de Microsoft Jet 4.x y en la mayoría de los motores de bases de datos relacionales, podemos definir un **tipo de consultas** cuyas filas resultantes **son un resumen de las filas de la tabla origen**, por eso las denominamos **consultas de resumen**, también se conocen como consultas sumarias.

Es importante entender que las filas del resultado de una consulta de resumen tienen una **naturaleza distinta** a las filas de las demás tablas resultantes de consultas, ya que corresponden a varias filas de la tabla origen. Para simplificar, veamos el caso de una consulta basada en una sola tabla, una fila de una consulta 'no resumen' corresponde a una fila de la tabla origen ya que contiene datos que se encuentran en una sola fila del origen, mientras que **una fila de una consulta de resumen corresponde a un resumen de varias filas de la tabla origen**, los datos no se encuentran en una fila concreta de la tabla origen. Esta diferencia es lo que va a originar una serie de restricciones que sufren las consultas de resumen y que veremos a lo largo del tema.

En el ejemplo que viene a continuación tienes un ejemplo de consulta normal en la que se visualizan las filas de la tabla oficinas ordenadas por región, en este caso cada fila del resultado se corresponde con una sola fila de la tabla oficinas. La segunda consulta es una consulta resumen. Cada fila del resultado se corresponde con una o varias filas de la tabla oficinas.

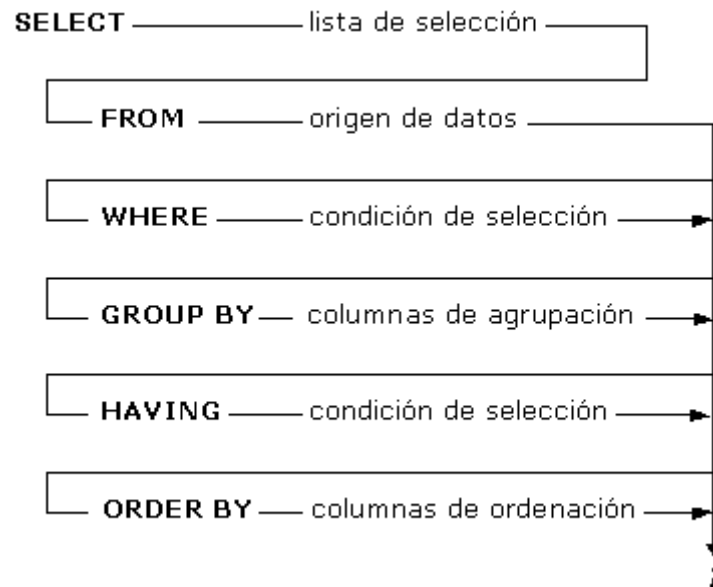
```
SELECT oficina,region,ventas
FROM oficinas
ORDER BY region
```

oficina	region	ventas
24	centro	150.000 Pts
23	centro	
28	este	0 Pts
13	este	368.000 Pts
12	este	735.000 Pts
11	este	693.000 Pts
26	norte	
22	oeste	186.000 Pts
21	oeste	836.000 Pts

```
SELECT region,SUM(ventas)
FROM oficinas
GROUP BY region
```

region	SumaDeventas
centro	150000
este	1796000
norte	
oeste	1022000

Las consultas de resumen introducen dos nuevas cláusulas a la sentencia SELECT: la cláusula GROUP BY y la cláusula HAVING. Son cláusulas que sólo se pueden utilizar en una consulta de resumen, se tienen que escribir entre la cláusula WHERE y la cláusula ORDER BY y tienen la siguiente sintaxis:



Las detallaremos en la página siguiente del tema, primero vamos a introducir otro concepto relacionado con las consultas de resumen, las funciones de columna.

## Funciones de columna

En la lista de selección de una consulta de resumen aparecen [funciones de columna](#) también denominadas funciones de dominio agregadas. Una función de columna [se aplica a una columna](#) y obtiene un [valor que resume el contenido de la columna](#).

Tenemos las siguientes funciones de columna:

SUM ( expresión )	MIN ( expresión )
AVG ( expresión )	MAX ( expresión )
STDEV ( expresión )	COUNT ( nbcolumna )
STDEVP ( expresión )	COUNT ( * )

El argumento de la función indica con qué valores se tiene que operar, por eso *expresión* suele ser un nombre de columna, columna que contiene los valores a resumir; pero también puede ser cualquier expresión válida que devuelva una lista de valores.

🟡 La función SUM() calcula la suma de los valores indicados en el argumento. Los datos que se suman deben ser de tipo numérico (entero, decimal, coma flotante o monetario...). El resultado será del mismo tipo aunque puede tener una precisión mayor.

Ejemplo:

```
SELECT  
SUM(ventas)  
FROM oficinas
```

Obtiene una sola fila con el resultado de sumar todos los valores de la columna ventas de la tabla oficinas.

● La función AVG() calcula el promedio (la media aritmética) de los valores indicados en el argumento, también se aplica a datos numéricos, y en este caso el tipo de dato del resultado puede cambiar según las necesidades del sistema para representar el valor del resultado.

- StDev() y StDevP() calculan la desviación estándar de una población o de una muestra de la población representada por los valores contenidos en la columna indicada en el argumento. Si la consulta base (el origen) tiene menos de dos registros, el resultado es nulo.

*Es interesante destacar que el valor nulo no equivale al valor 0, las funciones de columna no consideran los valores nulos mientras que consideran el valor 0 como un valor, por lo tanto en las funciones AVG(), STDEV(), STDEVP() los resultados no serán los mismos con valores 0 que con valores nulos.*

Veámoslo con un ejemplo:

Si tenemos esta tabla:

Tabla1 : Tabla	
col1	
10	
5	
0	
3	
6	
0	

La consulta

```
SELECT  
AVG(col1) AS  
media  
FROM tabla1
```

devuelve:

media
4

En este caso los ceros entran en la media por lo que sale igual a 4  
 $(10+5+0+3+6+0)/6 = 4$


Con esta otra tabla:

Tabla2 : Tabla	
col1	
10	
5	
3	
6	

```
SELECT  
AVG(col1) AS  
media  
FROM tabla2
```

devuelve:

media
6

En este caso los ceros se han sustituido por valores nulos y no entran en el cálculo por lo que la media sale igual a 6  
 $(10+5+3+6)/4 = 4$  

● Las funciones MIN() y MAX() determinan los valores menores y mayores respectivamente. Los valores de la columna pueden ser de tipo numérico, texto o fecha. El resultado de la función tendrá el mismo tipo de dato que la columna. Si la columna es de tipo numérico MIN() devuelve el valor menor contenido en la columna, si la columna es de tipo texto MIN() devuelve el primer valor en orden alfabético, y si la columna es de tipo fecha, MIN() devuelve la fecha más antigua y MAX() la fecha más reciente.

● La función COUNT(nbcolumna) cuenta el número de **valores no nulos** que hay en la columna. Los datos de la columna pueden ser de cualquier tipo, y la función siempre devuelve un número entero. Si la columna contiene valores nulos esos valores no se cuentan, si en la columna aparece un valor repetido, lo cuenta varias veces.

● COUNT(\*) permite contar filas en vez de valores. Si la columna no contiene ningún valor nulo, COUNT(nbcolumna) y COUNT(\*) devuelven el mismo resultado, mientras que si hay valores nulos en la columna, COUNT(\*) cuenta también esos valores mientras que COUNT(nbcolumna) no los cuenta.

Ejemplo:  
¿Cuántos empleados tenemos?

```
SELECT COUNT(numemp)  
FROM empleados
```

o bien

```
SELECT COUNT(*)  
FROM empleados
```

En este caso las dos sentencias devuelven el mismo resultado ya que la columna numemp no contiene valores nulos (es la clave principal de la tabla empleados).

¿Cuántos empleados tienen una oficina asignada?

```
SELECT COUNT(oficina)  
FROM empleados
```

Esta sentencia por el contrario, nos devuelve el **número de valores no nulos** que se encuentran en la columna oficina de la tabla empleados, por lo tanto nos dice **cuántos empleados tienen una oficina asignada.**

● Se pueden combinar varias funciones de columna en una expresión pero, no se pueden anidar, es decir:

```
SELECT (AVG(ventas) * 3) +  
SUM(cuota)  
FROM ...  
es correcto
```

```
SELECT AVG(SUM(ventas))  
FROM ...  
NO es correcto; no se puede incluir  
una función de columna dentro de  
una función de columna
```

## Selección en el origen de datos.

Si queremos eliminar del origen de datos algunas filas, basta incluir la cláusula WHERE que ya conocemos

Ejemplo: Queremos saber el acumulado de ventas de los empleados de la oficina 12.

```
SELECT SUM(ventas)
FROM empleados
WHERE oficina = 12
```

## Origen múltiple.

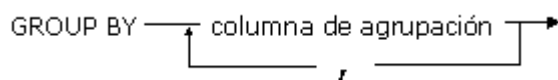
Si los **datos** que necesitamos utilizar para obtener nuestro resumen **se encuentran** en **varias tablas**, formamos el origen de datos adecuado en la cláusula **FROM** como si fuera una consulta **multitabla** normal.

Ejemplo: Queremos obtener el importe total de ventas de todos los empleados y el mayor objetivo de las oficinas asignadas a los empleados:

```
SELECT SUM(empleados.ventas), MAX(objetivo)
FROM empleados LEFT JOIN oficinas ON
empleados.oficina=oficinas.oficina
```

NOTA: combinamos empleados con oficinas por un **LEFT JOIN** para que aparezcan en el origen de datos todos los empleados incluso los que no tengan una oficina asignada, así el origen de datos estará formado por una tabla con tantas filas como empleados haya en la tabla empleados, con los datos de cada empleado y de la oficina a la que está asignado. De esta tabla sacamos la suma del campo ventas (importe total de ventas de todos los empleados) y el objetivo máximo. Observar que el origen de datos no incluye las oficinas que no tienen empleados asignados, por lo que esas oficinas no entran a la hora de calcular el valor máximo del objetivo.

## La cláusula GROUP BY



Hasta ahora las consultas de resumen que hemos visto utilizan todas las filas de la tabla y producen una única fila resultado.

🟡 Se pueden obtener **subtotales** con la cláusula **GROUP BY**. Una consulta con una cláusula **GROUP BY** se denomina **consulta agrupada** ya que agrupa los datos de la tabla origen y **produce una única fila resumen por cada grupo formado**. Las columnas indicadas en el **GROUP BY** se llaman **columnas de agrupación**.

Ejemplo:

```
SELECT  
SUM(ventas)  
FROM Empleados
```

Obtiene la suma de las ventas de todos los empleados.

```
SELECT  
SUM(ventas)  
FROM Empleados  
GROUP BY oficina
```

Se forma un grupo para cada oficina, con las filas de la oficina, y la suma se calcula sobre las filas de cada grupo. Este ejemplo obtiene una lista con la suma de las ventas de los empleados de cada oficina.

La consulta quedaría mejor incluyendo en la lista de selección la oficina para saber a qué oficina corresponde la suma de ventas:

```
SELECT oficina,SUM(ventas)  
FROM Empleados  
GROUP BY oficina
```

- Un columna de agrupación no puede ser de tipo memo u OLE.

● La **columna de agrupación** se puede indicar mediante un **nombre de columna** o cualquier expresión válida basada en una columna, pero no se pueden utilizar los alias de campo.

Ejemplo:

```
SELECT importe/cant ,  
SUM(importe)  
FROM pedidos  
GROUP BY importe/cant
```

Está permitido; equivaldría a agrupar las líneas de pedido por precio unitario y sacar de cada precio unitario el importe total vendido.

```
SELECT importe/cant AS precio,  
SUM(importe)  
FROM pedidos  
GROUP BY precio
```



No está permitido; no se puede utilizar un alias campo.

● En la lista de selección sólo pueden aparecer:  
valores constantes  
funciones de columna  
columnas de agrupación (columnas que aparecen en la cláusula **GROUP BY**)  
o cualquier expresión basada en las anteriores.

```
SELECT  
SUM(importe),rep*10  
FROM pedidos  
GROUP BY rep*10
```

```
SELECT SUM(importe),rep  
FROM pedidos  
GROUP BY rep*10
```

Está permitido

No está permitido; rep es una columna simple que no está encerrada en una función de columna, ni está en la lista de columnas de agrupación.



● Se **pueden agrupar** las filas **por varias columnas**, en este caso se indican las columnas separadas por una coma y en el **orden de mayor a menor agrupación**. Se permite incluir en la lista de agrupación hasta 10 columnas.

Ejemplo: Queremos obtener la suma de las ventas de las oficinas agrupadas por region y ciudad:

```
SELECT SUM(ventas)  
FROM oficinas  
GROUP BY region,ciudad
```

Se agrupa primero por región, y dentro de cada región por ciudad.

● **Todas las filas** que tienen **valor nulo** en el campo de agrupación, pasan a formar **un único grupo**. Es decir, considera el valor nulo como un valor cualquiera a efectos de agrupación.

Ejemplo:

```
SELECT oficina,SUM(ventas) AS ventas_totales  
FROM repventas  
GROUP BY oficina
```

En el resultado aparece una fila con el campo oficina sin valor y a continuación una cantidad en el campo ventas\_totales, esta cantidad corresponde a la suma de las ventas de los empleados que no tienen oficina asignada (campo *oficina* igual a nulo).

## La cláusula HAVING

HAVING ——— condición de selección ———→

● La cláusula **HAVING** nos permite **seleccionar filas** de la tabla resultante **de una consulta de resumen**.

● Para la condición de selección se pueden utilizar los mismos tests de comparación descritos en la cláusula **WHERE**, también se pueden escribir condiciones compuestas (unidas por los operadores **OR**, **AND**, **NOT**), pero existe una restricción:

**En la condición de selección sólo pueden aparecer:**  
**valores constantes**

**funciones de columna**

**columnas de agrupación** (columnas que aparecen en la cláusula GROUP BY)

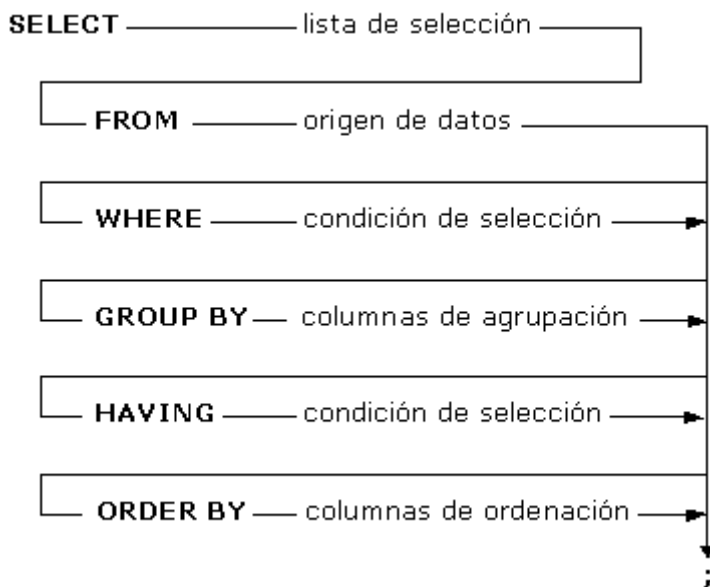
**o cualquier expresión basada en las anteriores.**

Ejemplo: Queremos saber las oficinas con un promedio de ventas de sus empleados mayor que 500.000 ptas.

```
SELECT oficina  
FROM empleados  
GROUP BY oficina  
HAVING AVG(ventas) > 500000
```

NOTA: Para obtener lo que se pide hay que calcular el promedio de ventas de los empleados de cada oficina, por lo que hay que utilizar la tabla empleados. Tenemos que agrupar los empleados por oficina y calcular el promedio para cada oficina, por último nos queda seleccionar del resultado las filas que tengan un promedio superior a 500.000 ptas.

## Resumen del tema



● ¿Cómo se ejecuta internamente una consulta de resumen?

- Primero se forma la tabla origen de datos según la cláusula **FROM**,
  - se seleccionan del origen de datos las filas según la cláusula **WHERE**,
  - se forman los grupos de filas según la cláusula **GROUP BY**,
  - por cada grupo se obtiene una fila en la tabla resultante con los valores que aparecen en las cláusulas **GROUP BY, HAVING** y en la lista de selección,
  - se seleccionan de la tabla resultante las filas según la cláusula **HAVING**,
  - se eliminan de la tabla resultante las columnas que no aparecen en la lista de selección,
  - se ordenan las filas de la tabla resultante según la cláusula **ORDER BY**

● Una consulta se convierte en consulta de resumen en cuanto aparece **GROUP BY, HAVING** o una función de columna.

● En una consulta de resumen, la lista de selección y la cláusula **HAVING** sólo pueden contener:  
valores constantes  
funciones de columna  
columnas de agrupación (columnas que aparecen en la cláusula **GROUP BY**)  
o cualquier expresión basada en las anteriores.

# Las subconsultas

## Definiciones

Una **subconsulta es una sentencia SELECT que aparece dentro de otra sentencia SELECT** que llamaremos **consulta principal**.

Se puede encontrar:

- **en la lista de selección,**
- **en la cláusula WHERE o**
- **en la cláusula HAVING** de la consulta principal.

Una subconsulta tiene la misma sintaxis que una sentencia SELECT normal exceptuando que aparece **encerrada entre paréntesis**, no puede contener la cláusula **ORDER BY**, ni puede ser la **UNION** de varias sentencias **SELECT**. Además tiene algunas restricciones en cuanto a número de columnas según el lugar donde aparece en la consulta principal. Estas restricciones las iremos describiendo en cada caso.

Cuando se ejecuta una consulta que contiene una subconsulta, **la subconsulta se ejecuta por cada fila de la consulta principal**.

Se aconseja no utilizar campos calculados en las subconsultas, ralentizan la consulta.

Las consultas que utilizan subconsultas suelen ser **más fáciles de interpretar por el usuario**.

## Referencias externas

A menudo, es necesario, dentro del cuerpo de una subconsulta, hacer referencia al valor de una columna en la fila actual de la consulta principal, ese nombre de columna se denomina referencia externa. Una **referencia externa** es un nombre de columna que estando en la subconsulta, no se refiere a ninguna columna de las tablas designadas en la **FROM** de la subconsulta sino a una **columna de las tablas designadas en la FROM de la consulta principal**. Como la subconsulta se ejecuta por cada fila de la consulta principal, el valor de la referencia externa irá cambiando.

Ejemplo:

```
SELECT numemp, nombre, (SELECT MIN(fechapedido) FROM pedidos  
WHERE rep = numemp)  
FROM empleados;
```

En este ejemplo la consulta principal es:

```
SELECT... FROM empleados.
```

La subconsulta es:

```
(SELECT MIN(fechapedido) FROM pedidos WHERE rep = numemp)
```

En la subconsulta tenemos una **referencia externa** (*numemp*); es un campo de la tabla empleados (origen de la consulta principal).

### ¿Qué pasa cuando se ejecuta la consulta principal?

- se coge el primer empleado y se calcula la subconsulta sustituyendo numemp por el valor que tiene en el primer empleado. La subconsulta obtiene la fecha más antigua en los pedidos del rep = 101,

- se coge el segundo empleado y se calcula la subconsulta con numemp = 102 (numemp del segundo empleado)... y así sucesivamente hasta llegar al último empleado.

Al final obtenemos una lista con el número, nombre y fecha del primer pedido de cada empleado.

Si quitamos la cláusula **WHERE** de la subconsulta obtenemos la fecha del primer pedido de todos los pedidos no del empleado correspondiente.

## Anidar subconsultas

Las subconsultas pueden **anidarse** de forma que **una subconsulta aparezca en la cláusula WHERE** (por ejemplo) **de otra subconsulta** que a su vez forma parte de otra consulta principal. *En la práctica, una consulta consume mucho más tiempo y memoria cuando se incrementa el número de niveles de anidamiento. La consulta resulta también más difícil de leer, comprender y mantener, cuando contiene más de uno o dos niveles de subconsultas.*

Ejemplo:

```
SELECT numemp, nombre  
FROM empleados  
WHERE numemp = (SELECT rep FROM pedidos WHERE clie = (SELECT  
numclie FROM clientes WHERE nombre = 'Julia Antequera'))
```

En este ejemplo, por cada línea de pedido se calcula la subconsulta de clientes, y esto se repite por cada empleado, en el caso de tener 10 filas de empleados y 200 filas de pedidos (tablas realmente pequeñas), la subconsulta más interna se ejecutaría **2000 veces (10 x 200)!!**.

## Subconsulta en la lista de selección

Cuando la subconsulta aparece **en la lista de selección** de la consulta principal, en este caso la subconsulta, **no puede devolver varias filas ni varias columnas**, de lo contrario se da un mensaje de error.

Muchos SQLs no permiten que una subconsulta aparezca en la lista de selección de la consulta principal, pero eso no es ningún problema, ya que normalmente se puede obtener lo mismo utilizando como origen de datos las dos tablas. Así el ejemplo primero que hemos visto se puede obtener de la siguiente forma:

```
SELECT numemp, nombre, MIN(fechapedido)
FROM empleados LEFT JOIN pedidos ON empleados.numemp =
pedidos.rep
GROUP BY numemp, nombre
```

## Subconsulta en la cláusula FROM

En la cláusula FROM se puede encontrar una sentencia SELECT encerrada entre paréntesis pero **más que subconsulta sería una consulta** ya que no se ejecuta para cada fila de la tabla origen, sino que se ejecuta una sola vez al principio, su resultado se combina con las filas de la otra tabla para formar las filas origen de la SELECT primera y no admite referencias externas.

En la cláusula FROM vimos que se podía poner un nombre de tabla o un nombre de consulta, pues en vez de poner un nombre de consulta se puede poner directamente la sentencia SELECT correspondiente a esa consulta encerrada entre paréntesis.

## Subconsulta en las cláusulas WHERE y HAVING

Se suele utilizar subconsultas en las cláusulas WHERE o HAVING cuando los datos que queremos visualizar están en una tabla pero para seleccionar las filas de esa tabla necesitamos un dato que está en otra tabla.

Ejemplo:

```
SELECT numemp, nombre
FROM empleados
WHERE contrato = (SELECT MIN(fechapedido) FROM pedidos)
```

En este ejemplo listamos el número y nombre de los empleados cuya fecha de contrato sea igual a la primera fecha de todos los pedidos de la empresa.

En una cláusula **WHERE** / **HAVING** tenemos siempre una condición y la subconsulta actúa de operando dentro de esa condición. En el ejemplo anterior se compara contrato con el resultado de la subconsulta.

Hasta ahora las condiciones estudiadas tenían como operandos valores simples (el valor contenido en una columna de una fila de la tabla, el resultado de una operación aritmética...). Ahora la subconsulta puede devolver una columna entera por lo que es necesario definir otro tipo de **condiciones especiales** para cuando se utilizan con subconsultas.

Estas nuevas condiciones se describen en la página siguiente...

## Condiciones de selección con subconsultas

Las **condiciones de selección** son las condiciones que pueden aparecer en la cláusula **WHERE** o **HAVING**. La mayoría se han visto en el tema 2 pero ahora incluiremos las condiciones que utilizan una subconsulta como operando.

En SQL tenemos cuatro nuevas condiciones:

- el **test de comparación con subconsulta**
- el **test de comparación cuantificada**
- el **test de pertenencia a un conjunto**
- el **test de existencia**

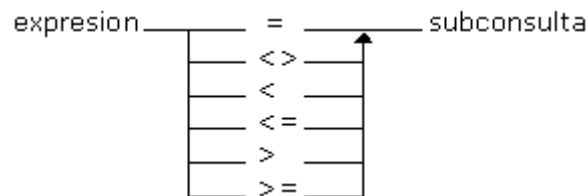
En todos los tests estudiados a continuación *expresion* puede ser cualquier nombre de columna de la consulta principal o una expresión válida como ya vimos en el tema 2.

### ● El **test de comparación con subconsulta**.

Es el **equivalente al test de comparación simple**. Se utiliza para comparar un valor de la fila que se está examinado con un único valor producido por la subconsulta. La subconsulta debe devolver una **única columna**, sino se produce un error.

Si la subconsulta no produce **ninguna fila** o devuelve el valor **nulo**, el test devuelve el **valor nulo**, si la subconsulta produce **varias filas**, SQL devuelve una **condición de error**.

La sintaxis es la siguiente:



```
SELECT oficina, ciudad  
FROM oficinas  
WHERE objetivo > (SELECT  
SUM(ventas) FROM  
empleados WHERE  
empleados.oficina =  
oficinas.oficina)
```

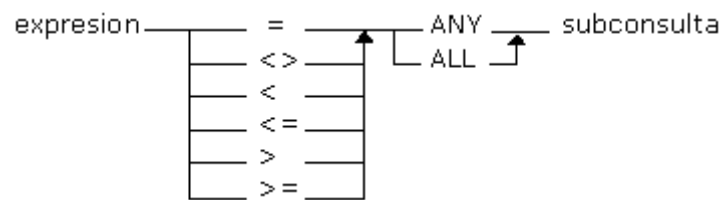
Lista las oficinas cuyo objetivo sea superior a la suma de las ventas de sus empleados. En este caso la subconsulta devuelve una única columna y una única fila (es un consulta de resumen sin **GROUP BY**)

### ● El **test de comparación cuantificada**.

Este test es una extensión del test de comparación y del test de conjunto. **Compara el valor** de la expresión **con cada uno de los valores** producidos por la subconsulta. La subconsulta debe devolver una **única columna** sino se produce un error.

Tenemos el test **ANY** (*algún, alguno* en inglés) y el test **ALL** (*todos* en inglés).

La sintaxis es la siguiente:



El **test ANY**:

La subconsulta debe devolver una **única columna**, si no, se produce un error. Se evalúa la comparación con cada valor devuelto por la subconsulta. Si **alguna de las comparaciones individuales produce el resultado verdadero, el test ANY devuelve el resultado verdadero**.

- Si la subconsulta no devuelve **ningún valor**, el test **ANY** devuelve **falso**.
- Si el test de comparación es **falso** para **todos los valores** de la columna, **ANY** devuelve **falso**.
- Si el test de comparación **no es verdadero** para ningún valor de la columna, **y es nulo** para al menos alguno de los valores, **ANY** devuelve **nulo**.

```
SELECT oficina, ciudad
FROM oficinas
WHERE objetivo > ANY
(SELECT SUM(cuota) FROM
empleados GROUP BY
oficina)
```

En este caso la subconsulta devuelve una única columna con las sumas de las cuotas de los empleados de cada oficina. La consulta lista las oficinas cuyo objetivo sea superior a alguna de las sumas obtenidas.

El **test ALL**:

La subconsulta debe devolver una única columna sino se produce un error. Se evalúa la comparación con cada valor devuelto por la subconsulta. **Si todas las comparaciones individuales, producen un resultado verdadero, el test devuelve el valor verdadero**.

- Si la subconsulta no devuelve **ningún valor** el test **ALL** devuelve el valor **verdadero**. (¡Ojo con esto!)
- Si el test de comparación es **falso** para algún valor de la columna, el resultado es **falso**.
- Si el test de comparación **no es falso** para ningún valor de la columna, pero es **nulo** para alguno de esos valores, el test **ALL** devuelve valor **nulo**.

```
SELECT oficina, ciudad
FROM oficinas
WHERE objetivo > ALL
(SELECT SUM(cuota) FROM
empleados GROUP BY
oficina)
```

En este caso se listan las oficinas cuyo objetivo sea superior a todas las sumas.

### ● Test de pertenencia a conjunto (IN).

Examina si el **valor** de la expresión es uno de los valores **incluidos en la lista de valores producida por la subconsulta**.

La subconsulta debe generar una única columna y las filas que sean.

Si la subconsulta no produce ninguna fila, el test da falso.

Tiene la siguiente sintaxis:

expresion — IN — subconsulta

```
SELECT numemp, nombre,
oficina
FROM empleados
WHERE oficina IN (SELECT
oficina FROM oficinas WHERE
region = 'este')
```

Con la subconsulta se obtiene la lista de los números de oficina del **este** y la consulta principal obtiene los empleados cuyo número de oficina sea uno de los números de oficina del **este**. Por lo tanto, lista los empleados de las oficinas del **este**.

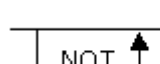
### ● El test de existencia EXISTS.

Examina si la subconsulta produce alguna fila de resultados. **Si la subconsulta contiene filas**, el test adopta el valor **verdadero**, si la subconsulta no contiene ninguna fila, el test toma el valor falso, nunca puede tomar el valor nulo.

Con este test la subconsulta **puede tener varias columnas**. No importa, ya que el test se fija, no en los valores devueltos sino, en si hay o no fila en la tabla resultado de la subconsulta.

Cuando se utiliza el test de existencia, en la mayoría de los casos, habrá que utilizar una referencia externa. Si no se utiliza una referencia externa la subconsulta devuelta siempre será la misma para todas las filas de la consulta principal, y en este caso se seleccionan todas las filas de la consulta principal (si la subconsulta genera filas) o ninguna (si la subconsulta no devuelve ninguna fila)

La sintaxis es la siguiente:


 EXISTS — subconsulta



```
SELECT numemp, nombre,  
oficina  
FROM empleados  
WHERE EXISTS (SELECT *  
FROM oficinas WHERE region  
= 'este' AND  
empleados.oficina =  
oficinas.oficina)
```

Este ejemplo obtiene lo mismo que el ejemplo del test **IN**.  
Observa que delante de **EXISTS** no va ningún nombre de columna.  
En la subconsulta se pueden poner las columnas que queramos en la lista de selección (hemos utilizado el \*).  
Hemos añadido una condición adicional al **WHERE**, la de la referencia externa para que la oficina que se compare sea la oficina del empleado.

NOTA. Cuando se trabaja con tablas muy voluminosas el test **EXISTS** suele dar mejor rendimiento que el test **IN**.

## Resumen del tema

- Una subconsulta es una sentencia **SELECT** que aparece en la lista de selección, o en las cláusulas **WHERE** o **HAVING** de otra sentencia **SELECT**.
- La subconsulta se ejecuta por cada fila de la consulta principal.
- Dentro de una consulta se puede utilizar una columna del origen de la consulta principal, una referencia externa.
- Aunque se puedan anidar subconsultas no es aconsejado más de un nivel de anidamiento.
- La subconsulta sufre una serie de restricciones según el lugar donde se encuentre.
- Las condiciones asociadas a las subconsultas son las siguientes:
  - el test de comparación con subconsulta
  - el test **ANY**
  - el test **ALL**
  - el test **IN**
  - el test **EXISTS**



● Cuando la tabla tiene una **columna de tipo contador** (AutoNumber), lo normal es **no asignar valor** a esa columna para que el sistema le asigne el valor que le toque según el contador, si por el contrario queremos que la columna tenga un valor concreto, lo indicamos en la lista de valores.

● Cuando no se indica **ninguna lista de columnas** después del destino, se asume por defecto **todas las columnas de la tabla**, en este caso, los valores se tienen que especificar en el **mismo orden** en que aparecen las columnas en la ventana de **diseño de dicha tabla**, y se tiene que utilizar el valor **NULL** para rellenar las columnas de las cuales no tenemos valores.

Ejemplo:

### **INSERT INTO empleados**

**VALUES (200, 'Juan López', 30, NULL, 'rep ventas', #06/23/01#, NULL, 350000, 0)**

Observar en el ejemplo que los valores de tipo texto se encierran entre comillas simples ' ' (también se pueden emplear las comillas dobles " ") y que la fecha de contrato se encierra entre almohadillas # # con el formato mes/día/año. Como no tenemos valor para los campos oficina y director (a este nuevo empleado todavía no se le ha asignado director ni oficina) utilizamos la palabra reservada **NULL**. Los valores numéricos se escriben tal cual, para separar la parte entera de la parte decimal hay que utilizar **siempre el punto** independientemente de la configuración que tengamos.

● Cuando indicamos **nombres de columnas**, estos corresponden a nombres de **columna de la tabla**, pero no tienen por qué estar en el **orden** en que aparecen en la ventana diseño de la tabla, también **se pueden omitir algunas columnas**, la columnas que no se nombran tendrán **por defecto el valor NULL o el valor predeterminado** indicado en la ventana de diseño de tabla.

El ejemplo anterior se podría escribir de la siguiente forma:

### **INSERT INTO empleados (numemp,oficina, nombre, titulo,cuota, contrato, ventas)**

**VALUES (200, 30, 'Juan López', 'rep ventas',350000, #06/23/01#,0)**

Observar que ahora hemos variado el orden de los valores y los nombres de columna no siguen el mismo orden que en la tabla origen, no importa, lo importante es poner los valores en el mismo orden que las columnas que enunciamos. Como no enunciamos las columnas oficina y director se rellenarán con el valor nulo (porque es el valor que tienen esas columnas como valor predeterminado).

● Utilizar la opción de **poner una lista de columnas** podría parecer peor ya que se tiene que escribir más pero realmente, **tiene ventajas**, sobre todo **cuando la sentencia la vamos a almacenar y reutilizar**:

● la sentencia queda **más fácil de interpretar** (*leyéndola vemos qué valor asignamos a qué columna*),

● de paso nos **aseguramos** que el valor lo asignamos a la columna que queremos,

● **si** por lo que sea **cambia el orden de las columnas en la tabla** en el diseño, no pasaría nada mientras que de la otra forma intentaría asignar los valores a otra columna; esto produciría errores de *'tipo no corresponde'* y lo que es peor, podría asignar valores erróneos sin que nos demos cuenta,

● otra ventaja es que **si se añade una nueva columna a la tabla** en el diseño, la primera sentencia INSERT daría error ya que el número de valores no corresponde con el número de columnas de la tabla, mientras que la segunda **INSERT** no daría error y en la nueva columna se insertaría el valor predeterminado.

#### ● **Errores que se pueden producir** cuando se ejecuta la sentencia **INSERT INTO**:

● **Si la tabla de destino tiene clave principal** y en ese campo intentamos no asignar valor, asignar el valor nulo o un valor que ya existe en la tabla, el motor de base de datos Microsoft Jet no añade la fila y da un mensaje de error de *'infracciones de clave'*.

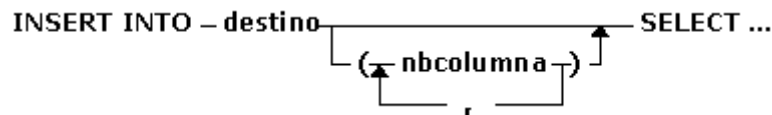
● **Si tenemos** definido **un índice único** (sin duplicados) e intentamos asignar un valor que ya existe en la tabla también devuelve el mismo error.

● **Si la tabla está relacionada con otra**, se seguirán las **reglas de integridad referencial**.

## Insertar varias filas INSERT INTO...SELECT

Podemos **insertar en una tabla varias filas** con una sola sentencia **SELECT INTO** si los valores a insertar se pueden obtener como resultado de una consulta, en este caso sustituimos la cláusula **VALUES lista de valores** por una sentencia **SELECT** como las que hemos visto hasta ahora. **Cada fila resultado** de la **SELECT forma una lista de valores** que son los que se insertan en una nueva fila de la tabla destino. Es como si tuviésemos una **INSERT...VALUES** por cada fila resultado de la sentencia **SELECT**.

La sintaxis es la siguiente:



- El **origen** de la **SELECT** puede ser el **nombre de una consulta guardada, un nombre de tabla o una composición de varias tablas** (mediante **INNER JOIN, LEFT JOIN, RIGHT JOIN** o producto cartesiano).
  - Cada fila devuelta por la **SELECT** actúa como la **lista de valores** que vimos con la **INSERT...VALUES** por lo que tiene las **mismas restricciones** en cuanto a tipo de dato, etc. La **asignación de valores se realiza por posición** por lo que la **SELECT** debe devolver el **mismo número de columnas** que las de la tabla destino y en el mismo orden, o el mismo número de columnas que indicamos en la lista de columnas después de destino.
  - Las columnas de la **SELECT** **no tienen porqué llamarse igual que en la tabla destino**, ya que el sistema sólo se fija en los valores devueltos por la **SELECT**.
  - Si **no queremos asignar valores a todas las columnas** entonces tenemos que indicar **entre paréntesis la lista de columnas** a rellenar **después del nombre del destino**.
  - El **estándar ANSI/ISO** especifica **varias restricciones sobre la consulta** que aparece dentro de la sentencia **INSERT**:
    - la consulta no puede tener una cláusula **ORDER BY**,
    - la tabla destino de la sentencia **INSERT** no puede aparecer en la cláusula **FROM** de la consulta o de ninguna subconsulta que ésta tenga. Esto prohíbe insertar parte de una tabla en sí misma,
    - la consulta no puede ser la **UNION** de varias sentencias **SELECT** diferentes,
    - el resultado de la consulta debe contener el mismo número de columnas que las indicadas para insertar y los tipos de datos deben ser compatibles columna a columna.
- Sin embargo en **SQL de Microsoft Jet**,
- **se puede incluir la cláusula ORDER BY aunque no tiene mucho sentido.**
  - **se puede poner** en la cláusula **FROM** de la consulta, **la tabla** en la que

vamos a insertar, pero **no** podemos utilizar una **UNION**.

Ejemplo: Supongamos que tenemos una tabla llamada *repres* con la misma estructura que la tabla *empleados*, y queremos insertar en esa tabla los empleados que tengan como título *rep ventas*

**INSERT INTO repres SELECT \* FROM empleados WHERE titulo = 'rep ventas'**

Con la **SELECT** obtenemos las filas correspondientes a los empleados con título *rep ventas*, y las insertamos en la tabla *repres*. Como las tablas tienen la misma estructura no hace falta poner la lista de columnas y podemos emplear **\*** en la lista de selección de la **SELECT**.

Ejemplo: Supongamos ahora que la tabla *repres* tuviese las siguientes columnas *numemp*, *oficinarep*, *nombrerep*. En este caso no podríamos utilizar el asterisco, tendríamos que poner:

**INSERT INTO repres SELECT numemp, oficina, nombre FROM empleados WHERE titulo = 'rep ventas'**

O bien:

**INSERT INTO repres (numemp, oficinarep, nombrerep) SELECT numemp, oficina, nombre FROM empleados WHERE titulo = 'rep ventas'**

## Insertar filas en una nueva tabla **SELECT ... INTO**

Esta sentencia **inserta filas creando** en ese momento **la tabla donde se insertan** las filas. Se suele utilizar **para guardar en una tabla el resultado de una SELECT**.

La **sintaxis** es la siguiente:

```
SELECT — lista selección — INTO — nuevatabla —  
                                     IN — bdexterna  
FROM ...
```

● Las **columnas de la nueva tabla** tendrán el **mismo tipo y tamaño que las columnas origen**, y **se llamarán con el nombre de alias** de la columna origen o **en su defecto con el nombre de la columna origen**, pero **no se transfiere ninguna otra propiedad** del campo o de la tabla **como por ejemplo las claves e índices**.

● La sentencia **SELECT** puede ser **cualquier sentencia SELECT sin ninguna restricción**, puede ser una consulta multitabla, una consulta de resumen, una **UNION** ...

Ejemplo:

**SELECT \* INTO t2 FROM t1**

Esta sentencia genera una nueva tabla t2 con todas las filas de la tabla t1. Las columnas se llamarán igual que en t1 pero t2 no será una copia exacta de t1 ya que no tendrá clave principal, ni relaciones con las otras tablas, ni índices si los tuviese t1 etc...

● Si en la base de datos hay ya una tabla del mismo nombre, el sistema **nos avisa** y nos pregunta si la queremos borrar. Si le contestamos que no, la **SELECT** no se ejecuta.

● Para formar una sentencia **SELECT INTO** lo mejor es **escribir la SELECT** que permite generar los datos que queremos guardar en la nueva tabla, y **después añadir** delante de la cláusula **FROM** la cláusula **INTO nuevatabla**.

● La sentencia **SELECT INTO** se suele utilizar para crear tablas de **trabajo**, o tablas intermedias, las creamos para una determinada tarea y cuando hemos terminado esa tarea las borramos. También puede ser útil **para sacar datos en una tabla para enviarlos a alguien**.

Por ejemplo: Queremos enviarle a un representante una tabla con todos los datos personales de sus clientes para que les pueda enviar cartas etc...

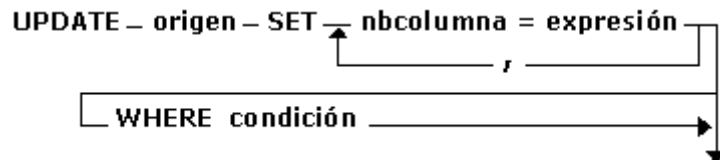
**SELECT numclie AS codigo, nombre, direccion, telefono INTO  
susclientes FROM clientes WHERE repclie = '103';**

Vamos a suponer que hemos añadido a nuestra tabla de *clientes* los campos dirección y teléfono. En el ejemplo anterior la nueva tabla tendrá cuatro columnas llamadas *codigo*, *nombre*, *direccion*, *telefono* y contendrá las filas correspondientes a los clientes del representante 103.

## Modificar el contenido de las filas ( UPDATE )

La sentencia **UPDATE** modifica los valores de una o más columnas en las filas seleccionadas de una o varias tablas.

La sintaxis es la siguiente:



- **Origen** puede ser un **nombre de tabla, un nombre de consulta o una composición de tablas**, también puede incluir la cláusula **IN** si la tabla a modificar se encuentra en una base de datos externa.

- La cláusula **SET** especifica **qué columnas van a modificarse y qué valores asignar a esas columnas**.

- **nbcolumna**, es el **nombre de la columna a la cual queremos asignar un nuevo valor** por lo tanto debe ser una columna de la tabla origen. El SQL estándar exige nombres sin cualificar pero algunas implementaciones (como por ejemplo el SQL de Microsoft Jet que estamos estudiando) sí lo permiten.

- La **expresión** en cada asignación **debe generar un valor del tipo de dato apropiado** para la columna indicada. La expresión **debe ser calculable a partir de los valores de la fila que se está actualizando**. *Expresión no puede ser una subconsulta.*

Ejemplo:

```
UPDATE oficinas INNER JOIN empleados  
ON oficinas.oficina = empleados.oficina  
SET cuota=objetivo*0.01;
```

En este ejemplo queremos actualizar las cuotas de nuestros empleados de tal forma que la cuota de un empleado sea el 1% del objetivo de su oficina. La columna a actualizar es la cuota del empleado y el valor a asignar es el 1% del objetivo de la oficina del empleado, luego la cláusula **SET** será **SET cuota = objetivo\*0.01** o **SET cuota = objetivo/100**. El origen debe contener la cuota del empleado y el objetivo de su oficina, luego el origen será el **INNER JOIN** de empleados con oficinas.

- La cláusula **WHERE** indica **qué filas van a ser modificadas**. Si se omite la cláusula **WHERE** se actualizan todas las filas.

- En la condición del **WHERE** se puede incluir una subconsulta. En SQL standard la tabla que aparece en la **FROM** de la subconsulta no puede ser la misma que la tabla que aparece como origen, pero en el SQL de Microsoft Jet sí se puede.



*Ejemplo:* Queremos poner a cero las ventas de los empleados de la oficina 12

**UPDATE empleados SET ventas = 0 WHERE oficina = 12;**

*Ejemplo:* Queremos poner a cero el límite de crédito de los clientes asignados a empleados de la oficina 12.

**UPDATE clientes SET limitecredito = 0  
WHERE repclie IN (SELECT numemp FROM empleados WHERE oficina = 12);**

● Si para el cálculo de *expresión* se utiliza una columna que también se modifica, el valor que se utiliza es el antes de la modificación, lo mismo para la condición de búsqueda.

● Cuando se ejecuta una sentencia **UPDATE**, primero se genera el origen y se seleccionan las filas según la cláusula **WHERE**. A continuación se coge una fila de la selección y se le aplica la cláusula **SET**, se actualizan todas las columnas incluidas en la cláusula **SET a la vez**, por lo que los nombres de columna pueden especificarse en cualquier orden. Después se coge la siguiente fila de la selección, y se le aplica del mismo modo la cláusula **SET**, así sucesivamente con todas las filas de la selección.

Ejemplo:

**UPDATE oficinas SET ventas=0, objetivo=ventas;**

O bien:

**UPDATE oficinas SET objetivo=ventas, ventas=0;**

Los dos ejemplos anteriores son equivalentes, ya que el valor de *ventas* que se asigna a *objetivo* es el valor antes de la actualización. Se deja como *objetivo* las ventas que ha tenido la oficina hasta el momento, y se pone a cero la columna *ventas*.

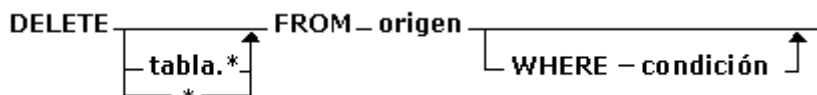
● Si actualizamos una columna definida como clave foránea, esta columna se podrá actualizar o no siguiendo las **reglas de integridad referencial**. El valor que se le asigna debe existir en la tabla de referencia.

● Si actualizamos una columna definida como columna principal de una relación entre dos tablas, esta columna se podrá actualizar o no siguiendo las **reglas de integridad referencial**.

## Borrar filas (DELETE)

La sentencia **DELETE** elimina filas de una tabla.

La sintaxis es la siguiente:



● **Origen** es el nombre de la **tabla de donde vamos a borrar**, podemos indicar un nombre de tabla, incluir la cláusula **IN** si la tabla se encuentra en una base de datos externa, también podemos escribir una composición de tablas.

● La opción **tabla.\*** se utiliza cuando el **origen está basado en varias tablas**, y sirve para indicar en **qué tabla vamos a borrar**.

● La opción **\*** es opcional y es la que se asume **por defecto** y se puede poner únicamente cuando el **origen es una sola tabla**.

● La cláusula **WHERE** sirve para especificar **qué filas queremos borrar**. Se eliminarán de la tabla todas las filas que cumplan la condición. Si **no se indica la cláusula WHERE, se borran TODAS las filas** de la tabla.

● **En la condición de búsqueda** de la sentencia **DELETE**, **se puede utilizar una subconsulta**. En SQL standard la tabla que aparece en la **FROM** de la subconsulta no puede ser la misma que la tabla que aparece en la **FROM** de la **DELETE** pero en el SQL de Microsoft Jet sí se puede hacer.

● Una vez borrados, **los registros no se pueden recuperar**.

● **Si la tabla donde borramos está relacionada con otras tablas** se podrán borrar o no los registros **siguiendo las reglas de integridad referencial** definidas en las relaciones.

Ejemplo:

```
DELETE * FROM pedidos WHERE clie IN (SELECT numclie FROM clientes WHERE nombre = 'Julian López');
```

O bien:

```
DELETE pedidos.* FROM pedidos INNER JOIN clientes ON pedidos.clie = clientes.numclie WHERE nombre = 'Julian López';
```

Las dos sentencias borran los pedidos del cliente *Julian López*. En la segunda estamos obligados a poner *pedidos.\** porque el origen está basado en varias tablas.

**DELETE \* FROM pedidos;** o **DELETE FROM pedidos;** Borra todas las filas de pedidos.

## Resumen del tema

● Si queremos **añadir** en una tabla **una fila con valores conocidos** utilizamos la sentencia **INSERT INTO tabla VALUES (lista de valores)**.

● Si los valores a insertar se encuentran en una o varias tablas utilizamos **INSERT INTO tabla SELECT ...**

● Para **crear una nueva tabla con el resultado de una consulta** con la sentencia **SELECT...INTO tabla FROM...**

● Para **cambiar los datos contenidos en una tabla**, tenemos que actualizar las filas de dicha tabla con la sentencia **UPDATE tabla SET asignación de nuevos valores**.

● Para **eliminar filas de una tabla** se utiliza la sentencia **DELETE FROM tabla**.

● Con la cláusula **WHERE** podemos indicar **a qué filas afecta la actualización o el borrado**.

## Tablas de referencias cruzadas

### Introducción

empleado	mes	vendido
101	1	26478
101	4	150
102	2	3750
102	3	1896
102	6	2130
103	2	2100
103	11	600
106	1	31500
106	12	1458
107	4	652
107	7	2430
107	8	31350
108	1	2925
108	4	1536
108	7	53520
108	8	652
108	10	15000
109	2	5625
109	7	1480
110	1	22500
110	11	632

Cuando queremos representar una **consulta sumaria con dos columnas de agrupación, como una tabla de doble entrada, en la que cada una de las columnas de agrupación es una entrada de la tabla, utilizaremos una** consulta de tabla de referencias cruzadas.

Por ejemplo queremos obtener las ventas mensuales de nuestros empleados. Tenemos que diseñar una consulta sumaria calculando la suma de los importes de los pedidos agrupando por empleado y mes de la venta.

La consulta sería:

```
SELECT rep as empleado, month(fechapedido) as mes, sum(importe) as vendido  
FROM pedidos  
GROUP BY rep, month(fechapedido)
```

El resultado sería la tabla que aparece a la derecha:

La consulta quedaría mucho más elegante y clara presentando los datos en un formato más compacto como el siguiente:

empleado	1	2	3	4	6	7	8	10	11	12
101	26478			150						
102		3750	1896		2130					
103		2100							600	
106	31500									1458
107				652		2430	31350			
108	2925			1536		53520	652	15000		
109		5625				1480				
110	22500								632	

Pues este último resultado se obtiene mediante una "consulta de referencias cruzadas". Observar que una de las columnas de agrupación (rep) sigue definiendo las filas que aparecen (hay una fila por cada empleado), mientras que la otra columna de agrupación (mes) ahora sirve para definir las otras columnas, cada valor de mes define una columna en el resultado, y la celda en la intersección de un valor de rep y un valor de mes es la columna resumen, la que contiene la función de columna (la suma de importe).

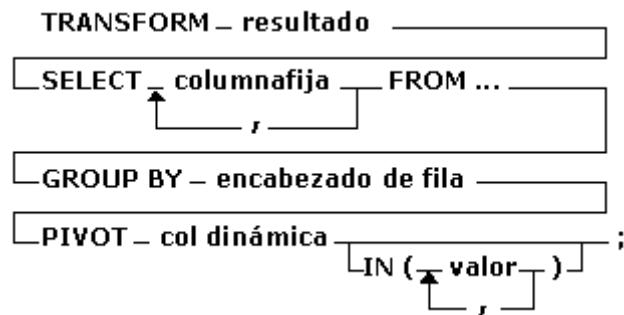
Las consultas de referencias cruzadas se pueden crear utilizando el asistente. Es mucho más cómodo, pero es útil saber cómo hacerlo directamente en SQL por si queremos variar algún dato una vez realizada la consulta con el asistente,

o si queremos definir una consulta de referencias cruzadas que no se puede definir con el asistente.

## La sentencia TRANSFORM

La sentencia **TRANSFORM** es la que se utiliza **para definir una consulta de referencias cruzadas**.

La sintaxis es la siguiente:



● **Resultado** es la **función de columna** que permite obtener el **resultado de las celdas**.

● En la **SELECT** la **columna fija** es la columna que define el **encabezado de filas**, el origen que indicamos en la cláusula **FROM** es la tabla (o tablas) de donde sacamos la información, y en la cláusula **GROUP BY** ponemos la **columna que va a definir las filas del resultado**.

● La **SELECT** puede contener una cláusula **WHERE** para seleccionar la filas que se utilizan para calcular el resultado, puede contener subconsultas pero no la cláusula **HAVING**.

● En la cláusula **PIVOT** indicamos la **columna** cuyos valores van a definir **columnas dinámicas del resultado** a esta columna la llamaremos **pivote**.

● La cláusula **IN** permite definir el **conjunto de valores que queremos que aparezcan como columnas dinámicas**.

● Es conveniente que la columna pivote que sirve de **encabezado de columna** tenga un **número limitado de posibles valores** para que no se generen demasiadas columnas. En nuestro ejemplo es mejor utilizar el mes como encabezado de columna y no de fila ya que posibles empleados hay muchos más y además el mes toma valores que conocemos y podemos utilizar por lo tanto la cláusula **IN** para que aparezcan todos los meses del año.

● En nuestro ejemplo resultado sería **SUM(importe)**, la columna fija es rep con un alias para que salga la palabra empleado en el encabezado, el origen de datos es la tabla pedidos (porque el resultado **SUM(importe)** se obtiene de pedidos), la columna del **GROUP BY** es rep ya que queremos una fila por cada representante, la columna dinámica, la que ponemos en la cláusula **PIVOT** sería **MONTH(fechapedido)**.

La sentencia quedaría de la siguiente forma:

```

TRANSFORM Sum(importe)
SELECT rep as empleado
FROM pedidos
GROUP BY rep
PIVOT month(fechapedido)

```

Lo mejor para montar una consulta de referencias cruzadas en SQL es pensar la sumaria normal y luego distribuir los términos según corresponda.

## Las columnas dinámicas

Como hemos dicho, las columnas dinámicas son las que se generan según los valores almacenados en la columna pivote (la que aparece en la cláusula **PIVOT**), normalmente se genera una columna dinámica por cada valor que se encuentre en la columna pivote del origen de datos.

Cuando los posibles valores que puede tomar la columna pivote son conocidos y queremos definir cuáles queremos que aparezcan, sólo unos cuantos porque no nos interesan algunos, o todos incluso si no generan resultado, en este caso usaremos la cláusula **IN**. En la cláusula **IN** se ponen entre paréntesis todos los posibles valores, o por lo menos los que queremos que aparezcan en el resultado.

Por ejemplo, sólo nos interesan los meses de febrero, mayo y diciembre:

```

TRANSFORM Sum(importe)
SELECT rep as empleado
FROM pedidos
GROUP BY rep
PIVOT month(fechapedido) IN (2,5,12);

```

empleado	2	5	12
101			
102	3750		
103	2100		
106			1458
107			
108			
109	5625		
110			

Si no utilizamos la cláusula **IN**, los meses de mayo y septiembre no aparecen ya que no hay pedidos realizados durante estos meses. Si utilizamos la cláusula **IN** y definimos los doce valores posibles, sí aparecen las columnas correspondientes a estos meses, y observamos que ningún empleado tiene ventas en esos meses.

```

TRANSFORM
Sum(importe)
SELECT rep as
empleado
FROM pedidos
GROUP BY rep
PIVOT
month(fechapedido)
IN

```

empleado	1	2	3	4	5	6	7	8	9	10
101	26478			150						
102		3750	1896			2130				
103		2100								
106	31500									
107				652			2430	31350		
108	2925			1536			53520	652		15000
109		5625					1480			
110	22500									

(1,2,3,4,5,6,7,8,9,10,11,12);

La cláusula **IN** también sirve para cambiar el orden de aparición de las columnas dinámicas, las columnas aparecen en el mismo orden en que aparecen en la cláusula **IN**.

**TRANSFORM**  
**Sum(importe)**  
**SELECT rep as empleado**  
**FROM pedidos**  
**GROUP BY rep**  
**PIVOT**  
**month(fechapedido) IN**  
**(10,11,12,1,2,3,4,5,6,7,**  
**8,9);**

empleado	10	11	12	1	2	3	4	5	6	7	
101				26478			150				
102					3750	1896			2130		
103		600			2100						
106			1458	31500							
107							652			2430	31
108	15000			2925			1536			53520	
109					5625					1480	
110		632		22500							

## Las columnas fijas

Las columnas fijas son las que aparecen delante de las columnas dinámicas, y son fijas porque se genera una sola columna en el resultado por cada columna hayamos indicado en la lista de columnas fijas. Las columnas fijas se indican en la lista de selección de la sentencia **SELECT**, una columna fija que siempre debemos incluir es la que sirve de encabezado de fila, para que podamos saber cada fila a qué valor de encabezado de fila corresponde. Pero además podemos incluir otras columnas, por ejemplo columnas de resumen de cada fila, sin que se tenga en cuenta la agrupación por la columna pivote.

Por ejemplo, queremos saber para cada empleado cuánto ha vendido en total y cuál ha sido el importe mayor vendido en un pedido.

**TRANSFORM**  
**Sum(importe) AS**  
**Suma**  
**SELECT rep AS**  
**empleado,**  
**SUM(importe) AS**  
**[Total**  
**vendido],MAX(imp**  
**orte) AS mayor**  
**FROM pedidos**  
**GROUP BY rep**  
**PIVOT**  
**month(fechapedid**  
**o);**

empleado	Total vendido	mayor	1	2	3	4	6	7	8
101	26628	22500	26478			150			
102	7776	3750		3750	1896		2130		
103	2700	2100		2100					
106	32958	31500	31500						
107	34432	31350				652		2430	31350
108	73633	45000	2925			1536		53520	652
109	7105	5625		5625				1480	
110	23132	22500	22500						

## Resumen del tema

La instrucción **TRANSFORM** se utiliza **para definir una consulta de referencias cruzadas**.

Permite presentar los resultados de una sumaria en una tabla de doble entrada como la que se presenta a continuación:

lista de selección de la SELECT			valores de la columna PIVOT							
empleado	Total vendido	mayor	1	2	3	4	6	7	8	
101	26628	22500	26478			150				
102	7776	3750		3750	1896		2130			
103	2700	2100		2100						
106	32958	31500	31500							
107	34432	31350				652		2430	31350	
108	73633	45000	2925			1536		53520	652	1
109	7105	5625		5625				1480		
110	23132	22500	22500							

valores de la columna GROUP BY

columnas fijas

TRANSFORM resultado

columnas dinámicas



# El DDL, lenguaje de definición de datos

## Introducción

Hasta ahora hemos estudiado las sentencias que forman parte del DML (Data Management Language) lenguaje de manipulación de datos, todas esas sentencias sirven para recuperar, insertar, borrar, modificar los datos almacenados en la base de datos; lo que veremos en este tema son las sentencias que afectan a la estructura de los datos.

El **DDL** (**D**ata **D**efinition **L**anguage) **lenguaje de definición de datos** es la parte del SQL que **más varía de un sistema a otro** ya que esa area tiene que ver con cómo se organizan internamente los datos y eso, cada sistema lo hace de una manera u otra.

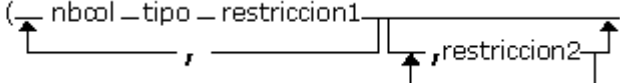
Así como el DML de Microsoft Jet incluye todas las sentencias DML que nos podemos encontrar en otros SQLs (o casi todas), el DDL de Microsoft Jet en cambio contiene menos instrucciones que otros sistemas.

## CREATE TABLE

La sentencia **CREATE TABLE** sirve para **crear la estructura de una tabla** no para rellenarla con datos, nos permite **definir las columnas** que tiene y **ciertas restricciones** que deben cumplir esas columnas.

La **sintaxis** es la siguiente:

```
CREATE TABLE ntabla ( ncol tipo restriccion1 , restriccion2 )
```



ntabla: **nombre** de la **tabla** que estamos definiendo

ncol: **nombre** de la **columna** que estamos definiendo

tipo: **tipo de dato** de la columna, todos los datos almacenados en la columna deberán ser de ese tipo. Para ver qué tipos de datos se pueden emplear haz clic [aquí](#)

Una **restricción** consiste en la definición de una **característica adicional que tiene una columna** o una combinación de columnas, suelen ser características como valores no nulos (campo requerido), definición de índice sin duplicados, definición de clave principal y definición de clave foránea (clave ajena o externa, campo que sirve para relacionar dos tablas entre sí).

restricción1: una **restricción de tipo 1** es una restricción que aparece **dentro de la definición de la columna** después del tipo de dato y **afecta a una columna**, la que se está definiendo.

restricción2: una **restricción de tipo 2** es una restricción que se define **después de definir todas las columnas** de la tabla y **afecta a una columna o a una combinación de columnas**.

Para escribir una sentencia **CREATE TABLE** se empieza por indicar el

**nombre de la tabla** que queremos crear y a continuación **entre paréntesis** indicamos **separadas por comas las definiciones de cada columna** de la tabla, la definición de una columna **consta de** su **nombre, el tipo de dato** que tiene **y** podemos añadir **si queremos una serie de especificaciones** que deberán cumplir los datos almacenados en la columna, **después** de definir cada una de las columnas que compone la tabla **se pueden añadir** una serie de **restricciones**, esas restricciones son las mismas que se pueden indicar para cada columna pero ahora **pueden afectar a más de una columna** por eso tienen una sintaxis ligeramente diferente.

Una **restricción de tipo 1** se utiliza para indicar una característica de la columna que estamos definiendo, tiene la siguiente sintaxis:

Ejemplo:

```
CREATE TABLE tab1 (  
col1 INTEGER CONSTRAINT pk PRIMARY KEY,  
col2 CHAR(25) NOT NULL,  
col3 CHAR(10) CONSTRAINT uni1 UNIQUE,  
col4 INTEGER,  
col5 INT CONSTRAINT fk5 REFERENCES tab2 );
```

Con este ejemplo estamos creando la tabla *tab1* compuesta por: una columna llamada *col1* de tipo entero definida como clave principal, una columna *col2* que puede almacenar hasta 25 caracteres alfanuméricos y no puede contener valores nulos, una columna *col3* de hasta 10 caracteres que no podrá contener valores repetidos, una columna *col4* de tipo entero sin ninguna restricción, y una columna *col5* de tipo entero clave foránea que hace referencia a valores de la clave principal de la tabla *tab2*.