Curso de SQL

Nivel básico

Jorge Navarrete Olmos
jorge@navarreteolmos.com
Febrero de 2004
Valencia

INDICE

1.INTRODUCCION

1.1 CICLO DE DESARROLLO

Cuando abordamos la creación de una base de datos, solemos ponernos a crear tablas sin más, a medida que vamos necesitándolas. Esto puede ser válido para pequeños proyectos y soluciones caseras, pero a la hora de crear una base de datos de considerable envergadura, se hace necesario planificar primero el modelo de datos y de aplicación.

No es mi intención extenderme en este tema ya que no es el objetivo de este curso, pero sí que vamos a ver de forma resumida las fases clásicas del desarrollo de un sistema:

- **Análisis**: Es la fase dónde analizamos los requerimientos de la empresa. Entrevistaremos a usuarios de las distintas áreas de negocio para establecer los requerimientos de datos y reglas de negocio. Después se construyen modelos gráficos según los datos e información recogida.
- **Diseño**: Diseñar la base de datos según el modelo más apto construido en la fase anterior.
- **Construcción y Documentación**: Elaboramos un prototipo del sistema. En esta fase se escriben los scripts para la creación de la base de datos (tablas, vistas, índices, etc...). Además se empieza a crear la documentación tanto para el usuario como para el equipo programador.
- **Transición**: También se le llama fase de pruebas. Se pone el prototipo en funcionamiento en manos de usuarios puntuales, se convierten los datos anteriores si existían y se realizan las modificaciones y correcciones requeridas.
- **Producción**: El sistema completo se pone a disposición de los usuarios. Lo que, entre los programadores, se llama “entrega”. El sistema comienza a operar en producción y debe ser monitorizado, revisado y refinado.

1.2 ALMACENAMIENTO DE DATOS. BASE DE DATOS RELACIONAL

Una organización o empresa tiene varias necesidades de información (empleados, nominas, departamentos, facturas, etc...). A esas piezas de información se les llama datos. Los datos se almacenan en diferentes medios, uno de ellos son las Bases de Datos. Es por esto que podríamos decir que una base de datos es una colección de información organizada.

En cuanto a los principios del modelo relacional, fueron propuestos por el Dr. E. F. Codd en el año 1970 en “Un Modelo de Datos Relacional para Grandes Bancos de Datos Compartidos”. Habían nacido los RDBMS.

Aquí van algunos enlaces de interés respecto al modelo relacional, aunque seguro que encontráis muchas más en www.google.com .

http://www.htmlpoint.com/sql/sql_03.htm
http://www.infor.uva.es/~chernan/Bases/Teoria/TEMA5.pdf
http://www.lcc.uma.es/~enciso/Docencia/unificadas/Enlaces.html

Suponemos que ya se conocen los conceptos de tabla, columna ó campo, y registro. Además vamos a suponer también que disponemos de una base de datos que soporte SQL para realizar los ejemplos. Nosotros vamos a utilizar por comodidad y disponibilidad particular la base de datos Oracle, pero cualquiera es válida. Desde aquí pedir disculpas si se me escapa algún término específico de dicha base de datos.

2. SQL – LENGUAJE DE CONSULTA ESTRUCTURADA

SQL es un lenguaje que puede utilizarse dentro de lenguajes de programación principales para transferir información seleccionada desde una base de datos. SQL también puede controlar el acceso a los recursos de la base de datos.

SQL difiere de muchos lenguajes de programación y de datos en que no es necesario codificar una secuencia de instrucciones para obtener los datos. SQL permite seleccionar, borrar y/o actualizar los datos usando una sola sentencia. También nos proporciona la posibilidad de definir datos. A las instrucciones de selección y/o manipulación de datos se les llama DML (Data Manipulation Language) y a las instrucciones de definición de datos u objetos de la base de datos se las conoce por instrucciones DDL (Data Definition Language).

Nos vamos a centrar en las sentencias DML ya que son las más independientes del sistema de bases de datos con el que estemos trabajando, y al finalizar el curso veremos las sentencias DDL más comunes como puede ser la sentencia CREATE TABLE. De momento, para las prácticas, podeis usar los GUI nativos o de terceros de cada sistema para crear las tablas de muestra.

Vamos ahora a dar una serie de definiciones básicas para el buen entendimiento del tema y que forman parte de casi todos los lenguajes de programación que soportan SQL:

Llamamos **SQL dinámico** a las sentencias que se preparan y procesan en un programa durante la ejecución del mismo. Estas sentencias pueden ser variadas en tiempo de ejecución, haciendo así muy flexible su utilización.

Llamamos **SQL estático** a las sentencias incluidas en un programa y que se preparan antes de la ejecución. Son sentencias “fijas” aunque puedan variarse algunos de sus parámetros.

Una **Consulta** es la instrucción básica de SQL y se define, como ya veremos con la sentencia SELECT.

Una **Subconsulta** es una consulta dentro de una condición de búsqueda que hace referencia a un valor o conjunto de valores que se necesitan para responder a la consulta principal (es una SELECT dentro de otra SELECT). Una subconsulta puede incluir condiciones de búsqueda propias (incluyendo nuevas subconsultas) o ligadas a la consulta principal. En este último caso, al conjunto total lo llamamos *“consulta sincronizada”.*

Los **Objetos SQL** son las Bases de Datos, las Tablas, las Vistas, índices, etc... Dependiendo del lenguaje principal, habrán más o menos objetos SQL. Los objetos SQL se crean mediante instrucciones DDL, que normalmente se almacenan en scripts de texto para poder recrearlos en cualquier momento. Los objetos SQL son mantenidos por el gestor de base de datos correspondiente.

2.1 RECUPERACION DE DATOS

Para la recuperación de datos (filas) de una tabla, se dispone de la sentencia SELECT (que ya hemos mencionado antes) y cuya estructura básica es la siguiente (ya veremos que hay más clausulas):

```
SELECT <campo_1>[,<campo_2>,...<campo_n>] | *
FROM <tabla_1> [alias_tabla_1][,<tabla_2> [alias_tabla_2],...
     <tabla_n> [alias_tabla_n]]
[WHERE <condición_de_selección>]
[GROUP BY <agrupamiento>
[HAVING <selección_de_grupos>]]
[ORDER BY <campos_de_ordenación> [DESC|ASC]] ;
```

Los elementos entre corchetes son opcionales y la | indica una opción u otra, de forma excluyente y el símbolo * indica que queremos recuperar todos los campos de cada fila de la tabla. También hay que destacar que para nombrar un campo se puede hacer por su nombre o por su nombre completo, es decir, con la notación nombre_tabla.nombre_campo, para evitar ambigüedades entre campos del mismo nombre en distintas tablas. Observar que las sentencias SQL terminan siempre con un punto y coma (;).

Veamos las partes de la SELECT por separado:

SELECT <campo_1>[,<campo_2>,...<campo_n>]
Son la relación de campos a recuperar de la tabla. Si queremos recuperar todos los campos, usaremos el símbolo * pero, en este caso, el orden de los campos no será especificado por nosotros. Los campos pueden ser expresiones aritméticas formadas por campos de las tablas, constantes u otras expresiones aritméticas.

FROM <tabla_1> [alias_tabla_1][,<tabla_2> [alias_tabla_2],...
<tabla_n> [alias_tabla_n]]
Indicamos la(s) tabla(s) que intervienen en la consulta. Podemos “renombrar” una tabla mediante un alias para evitar teclear el nombre completo de las mismas.

[WHERE <condición_de_selección>]
La clausula WHERE (donde) indica la condición que han de cumplir los registros ó filas de la tabla que se recuperen. También aparece aquí la unión entre las distintas tablas que puedan intervenir en la consulta, situación que ya veremos más adelante y las subconsultas ya mencionadas.

[GROUP BY <agrupamiento>
Agrupar por. Agrupa los resultados según los campos de <agrupamiento>.

[HAVING <selección_de_grupos>]]
Teniendo... condiciones. Esta clausula va ligada a la de GROUP BY y sirve para aplicar condiciones de búsqueda a cada uno de los grupos obtenidos con GROUP BY.

[ORDER BY <campos_de_ordenación> [DESC|ASC]]
Ordenar por. Ordena las filas ó registros recuperados según los <campos_de_ordenación>. Si se indica más de una columna, los resultados se ordenan primero por el primer campo, después por el segundo, etc... La subclausula DESC indica orden descendente y ASC indica orden ascendente (que es la opción por defecto).

Vamos a ver algunos ejemplos sencillos de consulta. Supongamos que tenemos una tabla llamada EMPLEADOS con la siguiente estructura:

NOMBRE DEL CAMPO	TIPO DE DATO	COMENTARIO
codigo	NUMBER(5)	Código del empleado. Clave primaria.
nombre	VARCHAR2(50)	Nombre y apellidos del empleado.
dni	VARCHAR2(10)	DNI/NIF del empleado.
departamento	NUMBER(3)	Código de departamento al que pertenece
salario	NUMBER(3)	Salario del empleado.
fecha_alta	DATE	Fecha de contratación del empleado.
jefe	NUMBER(5)	Código del jefe del empleado.

Queremos recuperar el nombre y el dni de todos los empleados. Para ello necesitamos usar la tabla EMPLEADOS. La consulta podría ser:

```
SELECT nombre, dni FROM EMPLEADOS ;
SELECT EMPLEADOS.nombre, EMPLEADOS.dni FROM EMPLEADOS ;
SELECT e.nombre, e.dni FROM EMPLEADOS e ;
```

En esta última hemos utilizado el alias “e” para nombrar a la tabla EMPLEADOS, con lo que se simplifica bastante la nomenclatura nombre_tabla.nombre_campo. Si quisiéramos ordenarla por orden alfabético sería (en el caso más simple):

```
SELECT nombre, dni FROM EMPLEADOS ORDER BY nombre ;
```

También podemos especificar el orden por el número de campo, es decir: ORDER BY 1 ú ORDER BY 2.

Pongamos por caso que ahora queremos recuperar los nombres y dni's de los empleados del departamento de ventas (el que tiene código 1). La sentencia sería como la siguiente:

```
SELECT nombre, dni FROM EMPLEADOS
WHERE departamento=1
ORDER BY nombre ;
```

Ahora veamos algún ejemplo con la clausula GROUP BY. Ahora queremos saber cuantos empleados tenemos en el citado departamento de ventas, así que tendremos que contarlos ¿no?. Para ello usaremos la función COUNT() de SQL (las funciones más usadas las veremos en siguientes capítulos) que baste decir, de momento, que sirve para contar registros.

```
SELECT COUNT(*) FROM EMPLEADOS
WHERE departamento=1 ;
```

Vale, pero ahora queremos que nos muestre el departamento y cuantos empleados tiene cada uno de ellos. Para ello vamos a agrupar por.... departamento, efectivamente:

```
SELECT departamento, COUNT(*) FROM EMPLEADOS
GROUP BY departamento ;
```

Y para acabar con estos ejemplos básicos de recuperación de datos, usaremos la clausula HAVING. Ahora lo que vamos a hacer es recuperar, como antes, los departamentos y su número de empleados pero solo los que tienen más de 3 empleados.

```
SELECT departamento, COUNT(*) FROM EMPLEADOS
GROUP BY departamento
HAVING COUNT(*)>3 ;
```

Más adelante, cuando hablemos de las funciones de grupo como puede ser COUNT(), SUM() y otras más, veremos que al ponerlas en una sentencia SELECT, estaremos obligados a agrupar por el resto de campos que no sean funciones de grupo. Pero eso ya será más adelante.

Para finalizar, deciros que las clausulas que hemos visto no son todas las que son. Tampoco hemos visto las subconsultas, ya que merecen un capítulo aparte. Seguiremos viendo la sentencia SELECT en los siguientes capítulos a medida que vayamos viendo conceptos. De momento hemos abierto boca y ya podemos practicar.

APENDICE A. OPERADORES DE COMPARACION Y LOGICOS

Estos operadores los utilizaremos en varias clausulas de las sentencias SQL, sobre todo en la clausula WHERE.

OPERADOR	SIMBOLO	EJEMPLO DE USO
IGUAL	=	departamento=1
DISTINTO	<> !=	departamento <> 1
MENOR	<	departamento < 4
MAYOR	>	departamento > 4
MENOR IGUAL	<=	departamento <= 4
MAYOR IGUAL	>=	departamento >= 4
O	OR	departamento=1 OR departamento=2
Y	AND	nombre='JOSE MARTINEZ' AND departamento=2
ENTRE (RANGO)	BETWEEN	departamento BETWEEN 1 AND 5
EN	IN	departamento IN (1,2,5,10)
COMO	LIKE	nombre LIKE 'JOSE%'
NO	NOT	nombre NOT LIKE 'JOSE%'
ES NULO	IS NULL	departamento IS NULL

NOTA: El símbolo % se usa como comodín. Es decir, en el ejemplo del operador LIKE hemos puesto:

```
nombre LIKE 'JOSE%'
```

Esto quiere decir que los nombres empiecen por JOSE y el resto de la cadena que haya en nombre nos da igual. Efectivamente, como ya estas pensando, podríamos poner el comodín delante también:

```
nombre LIKE '%MARTINEZ%'
```

Así seleccionamos todos los registros cuyo campo *nombre* contenga la cadena 'MARTINEZ', es decir, todos los 'MARTINEZ' de la tabla.

El símbolo _ denota un solo carácter y su utilización es análoga a la del símbolo %.

Si se desea una coincidencia exacta para los comodines, se usa la opción ESCAPE. Dicha opción especifica cual es el carácter de ESCAPE. Por ejemplo, para visualizar los nombres de empleados cuyo nombre contiene “N_A” usaríamos la siguiente sentencia SQL:

```
SELECT nombre FROM EMPLEADOS
WHERE nombre LIKE '%A\_B%' ESCAPE '\' ;
```

PRECEDENCIA DE LOS OPERADORES

- Todos los operadores de comparación
- NOT
- AND
- OR

APENDICE B. OPERADORES ARITMETICOS Y SU PRECEDENCIA

Los operadores aritméticos son usados para crear expresiones aritméticas. Éstas pueden estar formadas por nombres de campos, valores constantes y operadores aritméticos.

OPERADOR	DESCRIPCION
+	Suma
-	Resta
*	Multiplicación
/	División

Se pueden usar los operadores aritméticos en cualquier sentencia SQL excepto en la clausula FROM.

```
SELECT nombre, salario, salario * 2.06
FROM EMPLEADOS ;
```

Respecto a la precedencia de los operadores, si una expresión aritmética contiene más de un operador, la multiplicación y la división se evalúan primero. Si los operadores dentro de una expresión tienen la misma prioridad se evalúan de izquierda a derecha. Podemos utilizar paréntesis para forzar el orden de evaluación de los operadores aritméticos y para clarificar las sentencias.

Resumiendo:

1. La multiplicación y la división tienen igual prioridad pero mayor que la suma y la resta.
2. La suma y la resta tienen la misma prioridad.
3. Usaremos paréntesis para obligar a que la expresión encerrada en ellos se evalúe primero.

DEFINICION DEL VALOR NULL (Valores nulos)

Si un registro carece de valor para un campo en particular se dice que es un valor NULL, o que contiene un NULL.

NULL es un valor inaccesible, sin valor, desconocido o inaplicable. Un valor nulo no representa ni un cero ni un espacio en blanco. Las columnas o campos de cualquier tipo de datos pueden contener valores nulos excepto si se ha especificado como NOT NULL al crear la tabla o si es una clave primaria.

Hay que tener especial atención con los valores NULL a la hora de evaluar expresiones aritméticas, ya que si en una expresión, cualquier valor es nulo, el resultado también lo será. Por ejemplo, una división por cero nos dará un error, pero una división por NULL nos dará un nulo. Para controlar la evaluación de los valores NULL, disponemos de la función NVL que nos devolverá un valor por defecto en caso de que su argumento sea un valor nulo.

3.- FUNCIONES DE SQL

Ya hemos nombrado alguna que otra función SQL por lo que vamos a explicar lo que son, para qué se usan y cuales son las más importantes. Las funciones son una característica muy importante en cualquier lenguaje y en SQL no podía ser de otro modo. Las usaremos para:

- Realizar cálculos sobre datos
- Modificar items de datos particulares
- Manipular la salida de grupos de filas
- Modificar formato de los datos para su presentación
- Convertir los tipos de datos de las columnas.

Como todas las funciones, aceptan argumentos y devuelven un valor ó valores.

Vamos a dividir el capítulo en dos:

- Funciones a nivel de fila
- Funciones a nivel de grupos de filas, o funciones de grupo.

3.1- FUNCIONES A NIVEL DE FILA

Estas funciones operan solo sobre filas y devuelven un único resultado por cada una de ellas. Hay una gran variedad de ellas, pero veremos las de carácter, número, fecha y conversión.

Las características más relevantes de las funciones de fila son:

- Devuelven un resultado para cada fila recuperada
- Pueden devolver un dato de diferente tipo que el referenciado
- Se pueden usar en las clausulas SELECT, WHERE y ORDER BY
- Se pueden anidar

FUNCIONES DE CARACTERES

Estas funciones aceptan caracteres como datos de entrada y pueden devolver caracteres o números. Podríamos subdividirlas en dos: **funciones de conversión** (LOWER, UPPER, INITCAP) y de **manipulación de caracteres** (CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD).

Vamos a verlas de una en una con un pequeño ejemplo que deje claro su funcionamiento:

Funciones de conversión:

LOWER: Convierte en minúsculas la cadena de caracteres.
UPPER: Convierte en mayúsculas la cadena de caracteres.
INITCAP: Convierte la primera letra de cada palabra a mayúsculas.

Función	Resultado
LOWER('Curso de SQL')	curso de sql
UPPER('Curso de SQL')	CURSO DE SQL
INICAP('Curso de SQL')	Curso De Sql

Funciones de manipulación de caracteres:

CONCAT: Concatena valores. Sólo dos parámetros. Para concatenar más cadenas, usaremos el operador de concatenación (|| en el caso de Oracle)
SUBSTR: Extrae una cadena de una longitud y desde una posición.
LENGTH: Devuelve la longitud de una cadena en formato numerico.
LPAD: Justifica a la derecha con un carácter determinado.
RPAD: Justifica a la izquierda con un carácter determinado.

Función	Resultado
CONCAT('Hola',' Mundo')	Hola Mundo
SUBSTR('Curso de SQL',7,2)	de
LENGTH('Hola')	4
LPAD(5000,10,'*')	******5000
RPAD(5000,10,'*')	5000******

FUNCIONES NUMERICAS

Las funciones numéricas aceptan números como datos de entrada y devuelven valores numéricos. Las más usuales son ROUND, TRUNC y MOD.

ROUND: Redondea un valor al decimal especificado.
TRUNC: Trunca un valor en el decimal especificado.
MOD: Devuelve el resto de la división.

Función	Resultado
ROUND(166.386,2)	166.39
TRUNC(166.386,2)	166.38
MOD(25,4)	1

NOTA: Si el segundo argumento de MOD es cero, el resultado es el primer argumento y NO cero.

FUNCIONES DE FECHA

Para empezar, decir que podemos operar aritméticamente con las fechas.

- Sumar o resta un número a ó de una fecha, nos da otra fecha.
- Restar dos fechas nos devuelve los días entre esas fechas.
- Sumar horas a una fecha dividiendo la cantidad de horas por 24.

las siguientes funciones de fecha son usadas en Oracle, pero en cualquier otra base de datos deben aparecer sus análogas o las mismas.

MONTHS_BETWEEN: Número de meses entre dos fechas. Puede ser negativo si la primera fecha es menor que la segunda.
ADD_MONTHS: Agregar meses a una fecha. El número de meses a agregar puede ser negativo.
NEXT_DAY: Próximo día de la fecha especificada.
LAST_DAY: Último día del mes de la fecha.
ROUND: Redondea una fecha según el formato especificado.
TRUNC: Trunca una fecha según el formato especificado.
TO_CHAR: Convierte a caracteres una fecha según el formato.
TO_DATE: Convierte a fecha una cadena de caracteres válida.

Función	Resultado
MONTHS_BETWEEN('01/10/2003','01/12/2003')	2
ADD_MONTHS('11/01/1994',6)	11/07/1994
NEXT_DAY('01/02/2004','FRIDAY')	06/02/2004
LAST_DAY('02/02/2004')	29/02/2004
ROUND('25/07/2003','MONTH')	01/08/2003
ROUND('25/07/2003','YEAR')	01/01/2004
TRUNC('25/07/2003','MONTH')	01/07/2003
TRUNC('25/07/2003','YEAR')	01/01/2003
TO_CHAR(fecha_alta,'DD/MM/RR')	'01/01/04'
TO_CHAR(fecha_alta,'DD/MM/RRRR HH:MI')	'01/01/2004 10:05'
TO_DATE('01/01/04','DD/MM/RRRR')	01/01/2004

Ya hemos visto algún ejemplo de la función TO_CHAR usándola con fechas, ahora volvemos un poco atrás y la usaremos con números.

TO_CHAR(numero,'formato')

El formato que se le pasa como argumento aparte del número a convertir a carácter, determinará el resultado final de la conversión. Usaremos estos formatos combinados para mostrar el resultado:

Elemento	Descripción	ejemplo	Resultado
9	Posición numérica. El nº de 9's determinará el ancho de visualización	999999	1234
0	Muestra ceros a la izquierda o detrás del punto decimal.	00099.90	01234.00
.	Punto decimal en la posición	99999.90	1234.00
,	Coma en la posición especificada	999,999	1,234

Y terminamos con la función TO_NUMBER(char) que es la contraria a TO_CHAR, pero más simple, ya que la cadena de caracteres debe ser válida si no queremos obtener un error.

Función	Resultado
TO_NUMBER('09876')	9876
TO_NUMBER('98.76)	98.76
TO_NUMBER('1,345.98')	ERROR (por la ,)

FUNCION NVL

Esta función, como ya hemos comentado antes, previene los resultados erráticos de las expresiones aritméticas que incluyen NULL en alguno de sus elementos. ¿Cómo? Pues simplemente establece un valor para el caso de que el elemento sea NULL. Así de sencillo. Es decir que si en una expresión ponemos NVL(departamento,0) y “departamento” es nulo, nos devolverá un cero. Quizá este ejemplo no sea el más acertado, pero pensemos ahora en un % de comisión de un vendedor que evaluado con las ventas pueda darnos valores nulos.

La cuestión se agrava si quisiéramos sumar las comisiones de nuestros vendedores multiplicando primero las ventas individuales por su porcentaje de comisión. Los valores nulos sería arrastrados hasta darnos un total nulo, totalmente falso.

El valor por defecto debe coincidir con el tipo de dato que estamos tratando.

Antes de concluir esta primera parte del capítulo de funciones, hay que resaltar que tanto Oracle como otras bases de datos, así como los sistemas operativos, llevan una configuración regional que afecta a los formatos numéricos sobre todo, en cuanto al punto y la coma. Por ello, en Oracle se usa el símbolo D para sustituir al punto decimal en los formatos que hemos visto y el símbolo G para sustituir a la coma de millares. De esta forma se evita que las instrucciones puedan fallar en distintos sistemas.

3.2- FUNCIONES A NIVEL DE GRUPOS DE FILAS

Las funciones de grupo son funciones que operan sobre conjuntos de registros para dar un resultado a nivel de grupo. Dichos grupos pueden estar constituidos por la tabla entera o por partes de la misma. Estas funciones aparecen en las clausulas SELECT y HAVING. Estas son las más comunes:

AVG([DISTINCT|ALL] n): Valor promedio de n, ignorando los valores nulos.
COUNT({*|[DISTINCT|ALL] expr}): Cantidad de filas cuando "expr" da un valor no nulo. Para contar todas las filas (incluyendo duplicadas y valores nulos) usaremos "*".
MAX([DISTINCT|ALL] expr): Valor máximo de "expr".
MIN([DISTINCT|ALL] expr): Valor mínimo de "expr".
STDDEV([DISTINCT|ALL]x): Desviación estándar de x, ignorando los nulos.
SUM([DISTINCT|ALL]n): Suma los valores de n ignorando los nulos.
VARIANCE([DISTINCT|ALL]x): Varianza de n ignorando los valores nulos.

- DISTINCT hace que la función considere sólo los valores no duplicados, mientras que ALL incluye todos (es el valor por defecto).
- Los tipos de datos de los argumentos de "expr" pueden ser CHAR, VARCHAR2, NUMBER o DATE.
- Todas las funciones de grupo excepto COUNT(*) ignoran los valores nulos. Para evitar esto podemos usar la función NVL para sustituir el valor NULL por otro valor.
- AVG y SUM se usan para datos numéricos.
- MIN y MAX se usan para cualquier tipo de dato.

Como lo mejor para entenderlas es usarlas, veamos unos cuantos ejemplos:

Obtener la media, salario más alto, más bajo y la suma de salarios para el departamento de ventas (sin tener en cuenta los nulos):

```
SELECT AVG(salario), MAX(salario), MIN(salario), SUM(salario)
FROM EMPLEADOS WHERE departamento=1 ;
```

Obtener la media, salario más alto, más bajo y la suma de salarios para el departamento de ventas teniendo en cuenta todos los valores:

```
SELECT AVG(NVL(salario,0)), MAX(NVL(salario,0)),
       MIN(NVL(salario,0)), SUM(NVL(salario,0))
FROM EMPLEADOS WHERE departamento=1 ;
```

Obtener la fecha de contratación más antigua y la más moderna:

```
SELECT MAX(fecha_alta), MIN(fecha_alta) FROM EMPLEADOS ;
```

Obtener el primer nombre de empleado alfabéticamente y el último:

```
SELECT MAX(nombre), MIN(nombre) FROM EMPLEADOS ;
```

Obtener el número de empleados del departamento de ventas:

```
SELECT COUNT(*) FROM EMPLEADOS WHERE departamento=1;
```

Obtener el numero de empleados que tienen asignado jefe:

```
SELECT COUNT(jefe) FROM EMPLEADOS ;
```

Obtener el número de departamentos distintos que hay en EMPLEADOS.

```
SELECT COUNT(DISTINCT(departamento)) FROM EMPLEADOS;
```

Por último, comentar que las funciones de grupo se pueden anidar como las demás funciones:

```
SELECT MAX(AVG(salario)) FROM EMPLEADOS GROUP BY departamento ;
```

Aquí calculamos la máxima de las medias de los salarios por departamento.

4 RECUPERACION DE DATOS. GRUPOS DE DATOS.

Después de ver las funciones de grupo, ya podemos profundizar más en las clausulas GROUP BY y HAVING de la sentencia SELECT. A veces necesitamos dividir la información de una tabla en grupos más pequeños a la hora de la recuperación de datos. Esto lo haremos con GROUP BY.

El siguiente ejemplo muestra el número de departamento y el salario medio para cada uno de ellos.

```
SELECT departamento, AVG(salario) FROM EMPLEADOS
GROUP BY departamento ;
```

Estas son una serie de reglas para el uso de GROUP BY y HAVING.

- Si se incluye una función de grupo en la sentencia SELECT, no se puede seleccionar resultados individuales a menos que la columna aparezca en la clausula GROUP BY.
- Con la clausula WHERE se pueden excluir filas antes de la división en grupos.
- No se puede usar un “alias” de campo ó columna en la clausula GROUP BY.
- Por defecto las filas se ordenan en forma ascendente de acuerdo a la lista GROUP BY. Esto se puede modificar con ORDER BY.
- La(s) columna(s) referenciada(s) por GROUP BY no es necesario seleccionarla(s) en la lista de campos a recuperar, aunque sin ella(s) los datos recuperados pueden no ser relevantes.

  ```
  SELECT MAX(salario) FROM EMPLEADOS GROUP BY departamento ;
  ```
- Se puede usar la función de grupo en la clausula ORDER BY.

  ```
  SELECT AVG(salario) FROM EMPLEADOS GROUP BY AVG(salario) ;
  ```
- Podemos agrupar por más de una columna, creando grupos dentro de grupos

  ```
  SELECT departamento, cargo, SUM(salario)
  FROM EMPLEADOS
  GROUP BY departamento, cargo ;
  ```

  (Aquí hemos supuesto que disponemos de un campo ó columna llamada cargo que informa del puesto que ocupa cada empleado). La función SUM está siendo aplicada al salario para todos los “cargos” dentro de cada numero de departamento.

Para finalizar, veremos como excluir resultados de un grupo y como anidar funciones de grupo.

4.1 EXCLUSION DE RESULTADOS DE UN GRUPO. USO DE HAVING

De la misma forma que usamos WHERE para restringir los registros que seleccionamos, usaremos HAVING para restringir grupos de registros. Es decir: los grupos que se correspondan con la clausula HAVING serán los mostrados.

```
SELECT departamento, max(salario) FROM DEPARTAMENTOS
GROUP BY departamento HAVING MAX(salario)>3000 ;
```

Podemos utilizar la clausula GROUP BY sin utilizar una función de grupo en la SELECT:

```
SELECT departamento FROM EMPLEADOS
GROUP BY DEPARTAMENTO
HAVING MAX(salario) > 3000 ;
```

Por supuesto podemos ordenar los resultados:

```
SELECT cargo, SUM(salario) FROM EMPLEADOS
WHERE cargo NOT LIKE 'VENDEDOR%'
GROUP BY cargos
HAVING SUM(salario) > 5000
ORDER BY SUM(salario) ;
```

En este último ejemplo vemos como la clausula WHERE restringe los registros antes de que se calculen los grupos y se excluyan resultados mediante HAVING.

La clausula HAVING puede ponerse antes de la clausula GROUP BY pero no se recomienda por lógica. Los grupos se forman, y las funciones de grupo se calculan antes de que la clausula HAVING sea aplicada a los grupos.

5. RECUPERACION DE DATOS. RELACIONANDO TABLAS. JOINS.

En los ejemplos anteriores hemos visto un campo en la tabla EMPLEADOS llamado “departamento”. Además vemos que es un código numérico que debe hacer referencia a otra tabla que contenga los departamentos de la empresa. Es decir, que tenemos una tabla DEPARTAMENTOS con la siguiente estructura:

NOMBRE DEL CAMPO	TIPO DE DATO	COMENTARIO
codigo	NUMBER(5)	Código del departamento. Clave primaria
nombre	VARCHAR2(30)	Nombre del departamento

Al campo EMPLEADOS.departamento se le llama “clave ajena” (Foreign key) donde el campo DEPARTAMENTOS.codigo es la clave primaria. Recordemos que una clave primaria siempre es única, pero una clave única no tiene por qué ser la clave primaria de una tabla. También decir que una clave ajena tiene que coincidir con un valor de una clave primaria ya existente o con un valor de una clave única ya que en otro caso deberá ser NULL. Por último, una clave ajena puede hacer referencia a una clave primaria o una clave única en la propia tabla. Por ejemplo, tenemos el caso del campo “jefe” en la tabla EMPLEADOS, que es una clave ajena que referencia al campo “codigo” de empleado de la propia tabla EMPLEADOS. Es decir, se almacena el código del empleado que es el jefe del empleado actual.

Debido a que los datos sobre diferentes entidades se almacenan en tablas diferentes, podríamos necesitar combinar dos o más tablas para responder a una solicitud de recuperación de datos. Por ejemplo, queremos saber el nombre de cada departamento donde trabajan los empleados. Está claro que necesitamos información de la tabla EMPLEADOS que contiene los datos de los empleados y de la tabla DEPARTAMENTOS que contiene el nombre de los mismos. Para realizar la consulta debemos "relacionar" las dos tablas mediante la clave ajena "departamento" de EMPLEADOS y la clave primaria "codigo" de DEPARTAMENTOS.

Primero veremos lo que es un "join". Cuando necesitamos datos de más de una tabla de la base de datos, se utiliza una condición join. Las filas de una tabla pueden unirse a las de otra según los valores comunes existentes en las mismas, osea, los valores de clave ajena y de clave primaria. La condición de join la especificamos en la clausula WHERE:

```
SELECT e.nombre, e.dni, d.nombre
FROM EMPLEADOS e, DEPARTAMENTOS d
WHERE e.departamento = d.codigo
```

Bastante claro, ¿no?. Es evidente que cualquier departamento grabado en EMPLEADOS deberá estar en DEPARTAMENTOS, pero podría darse el caso de que un empleado tuviera en el campo "departamento" un nulo (NULL). Es decir, que el trabajador no estuviera asignado a ningún departamento, por ejemplo, el Director General. En este caso, nos daremos cuenta que con la SELECT anterior no aparecen estos registros ya que no hay coincidencia en la clausula WHERE.

Imaginemos ahora los registro de una tabla de clientes que no tienen asignado todavía un comercial fijo que les atienda. Es más, no es obligatorio que lo tenga. ¿Cómo podríamos obtener en una consulta los clientes y los nombres de los comerciales que los atienden si los que no tienen comercial no nos aparecen?. ¿Qué se hace para remediar estas situaciones?. La respuesta son las "outer join". Entonces.... ¿tenemos varios tipos de joins?. Sí.

Equijoin: Para determinar el nombre del departamento de un empleado, se compara el valor de la columna "departamento" de EMPLEADOS con los valores de "codigo" de la tabla DEPARTAMENTOS. La relación entre ambas es un equijoin, es decir los valores deben ser iguales. Los equijoins también son llamados "inner joins" (esto es terminología para que os suene). El ejemplo anterior es un equijoin.

Non-equijoins: Son las joins en las que la condición no es de igualdad. Para comprender esto vamos a suponer otra nueva tabla llamada NIVEL_SALARIAL con esta estructura:

NOMBRE DEL CAMPO	**TIPO DE DATO**	**COMENTARIO**
nivel	NUMBER(5)	Nivel salarial. Clave primaria
salario_min	NUMBER(15,2)	Salario mínimo del nivel actual.
salario_max	NUMBER(15,2)	Salario máximo del nivel actual.

¿Como haríamos par obtener el nivel salarial de cada empleado?

```
SELECT e.nombre, e.salario, n.nivel FROM EMPLEADOS e, NIVEL_SALARIAL n
WHERE e.salario BETWEEN n.salario_min AND n.salario_max ;
```

outer joins: Se usan para devolver registros sin coincidencias directas. Si una fila no satisface la condición de join, no aparecerá en los resultados de la consulta. Por ejemplo, si no hay ningún empleado que trabaje en un departamento, éste no aparecerá:

```
SELECT e.nombre, e.departamento, d.nombre
FROM EMPLEADOS e, DEPARTAMENTOS d
WHERE e.departamento = d.codigo ;
```

Para obtener las filas que no aparecen usaremos el operador del outer join, que es el "(+)"

```
SELECT e.nombre, e.departamento, d.nombre
FROM EMPLEADOS e, DEPARTAMENTOS d
WHERE e.departamento(+) = d.codigo ;
```

El operador se coloca en el lado del join que es "deficiente" en información. El operador outer join puede aparecer solo en un lado de la expresión. Este recupera las filas de una tabla que no tiene correspondencia directa en la otra tabla. Una condición que incluye una outer join no puede utilizar el operador IN o unirse a otra condición por el operador OR, pero sí podemos usar el operado AND.

Self joins: Es cuando necesitamos combinar una tabla consigo misma. Por ejemplo, para encontrar el nombre del jefe de cada empleado necesitamos combinar la tabla EMPLEADOS consigo misma ya que un jefe también es un empleado y, como tal, se encuentra tambien en la tabla EMPLEADOS. En estos casos miramos dos veces la misma tabla, una para recuperar al empleado y otra para recuperar el nombre del jefe.

```
SELECT trab.nombre EMPLEADO, jefes.nombre JEFE
FROM EMPLEADOS trab, EMPLEADOS jefes
WHERE trab.jefe = jefes.codigo ;
```

Como se ve, hemos referenciado la misma tabla EMPLEADOS dos veces con diferentes alias de tabla, simulando así tener dos tablas y hemos hecho el join con los campos jefe y codigo.

Ahora bien, ¿qué pasa si omitimos una condición de join o la condición es inválida?. La respuesta se llama "Producto Cartesiano" y su nombre refleja exactamente lo que ocurre: Se combinan todas las tablas de la primera tabla con todas las filas de la segunda.

El producto cartesiano genera una gran cantidad de filas y este resultado es raramente útil. Por ello debemos asegurarnos de incluir siempre una condición de join válida en la clausula WHERE.

Si quereis un ejemplo de producto cartesiano, simplemente quitar la clausula WHERE del ejemplo anterior o de cualquiera de los ejemplos de joins anteriores.

Bien, pues hemos visto cómo obtener datos de varias tablas relacionándolas mediante las claves primarias, claves ajenas y/o con joins. En caso de querer relacionar más de dos tablas, el procedimiento es el mismo añadiendo más tablas en la clausula FROM y las condiciones de unión correspondientes en la clausula WHERE.

Por ejemplo, supongamos que disponemos de otra tabla llamada SALARIOS con la siguiente estructura:

NOMBRE DEL CAMPO	TIPO DE DATO	COMENTARIO
empleado	NUMBER(5)	Código de empleado. Clave primaria
fecha_salario	DATE	Fecha de fijación del salario.
salario	NUMBER(15,2)	Salario

En este caso la primary key serían los campos empleado+fecha_salario ya que un empleado puede tener varios registros de salario a distintas fechas (histórico de salarios por empleado). Es evidente que el campo empleado es a su vez una clave ajena del campo codigo de la tabla EMPLEADOS.

Ahora queremos obtener una lista de empleados, con el nombre del departamento donde trabajan (si no tienen departamento asignado, deben aparecer también), el nombre de su jefe y el sueldo del empleado hasta el 31/12/2003 (suponemos que un empleado siempre tiene sueldo, jejeje).

```
SELECT e.nombre EMPLEADO, d.nombre DEPART, j.nombre JEFE, s.salario SUELDO
FROM EMPLEADOS e, EMPLEADOS j, DEPARTAMENTOS d, SALARIOS s
WHERE e.departamento = d.codigo(+)
  AND e.jefe = j.codigo
  AND e.codigo = s.empleado
  AND s.fecha_salario =
           (SELECT MAX(fecha_salario) FROM SALARIOS sal
            WHERE sal.empleado = s.empleado
              AND TRUNC(fecha_salario)<=TO_DATE('31/12/03','DD/MM/RRRR'));
```

Sí ya sé que en esta SELECT aparece algo nuevo. Se llama subconsulta y además es de un tipo especial denominada “sincronizada” ya que en ella aparece un campo de la SELECT exterior que provoca que se sincronicen las dos. El tema de Subconsultas lo vamos a ver en el capítulo siguiente.

De momento quedaros con que hemos usado varias tablas con diferentes joins, hemos usado outer join, self join y non-equijoins. También hemos usado funciones de grupo y de conversión de datos.

6.- SUBCONSULTAS

Las subconsultas se utilizan cuando necesitamos más de una consulta para resolver un problema de recuperación de datos. Cierto es que podríamos hacer un programa que usara primero una consulta y después otra que utilizara los resultados de la primera, pero aquí tratamos de resolver los retos usando una consulta, por muy compleja que sea, para demostrar la potencia del lenguaje SQL. La programación utilizando SQL es otro cantar y merece un curso entero.

Siguiendo los ejemplos del curso, supongamos que queremos saber los empleados que ganan más que otro empleado llamado “Jorge” y cuyo código es el uno. Para resolver esto, necesitamos dos consultas: una para saber lo que gana “Jorge” y otra para obtener los empleados que ganen más que lo que gana “Jorge”. Vamos a resolverlo escribiendo una consulta dentro de otra, es decir: usando una subconsulta.

Para averiguar el salario de “Jorge” haríamos:

```
SELECT salario FROM EMPLEADOS
WHERE codigo = 1 ;
```

Pues para obtener los empleados que ganan más que “Jorge” hacemos:

```
SELECT nombre, salario FROM EMPLEADOS
WHERE salario > ( SELECT  salario FROM EMPLEADOS
                  WHERE codigo = 1 ) ;
```

La consulta más interna o subconsulta devuelve el salario de “Jorge” que será utilizado por la consulta principal (la externa) para obtener el resultado deseado.

Usar una subconsulta es equivalente a realizar dos consultas secuencialmente y utilizar el resultado de la primera como valor para la segunda consulta (la principal).

En definitiva: una subconsulta es una sentencia SELECT que está incluida en una clausula de otra sentencia SELECT (realmente de otra sentencia SQL como ya veremos cuando estudiemos las sentencias INSERT, UPDATE y DELETE). Se puede poner la subconsulta en algunas de las clausulas de un comando SQL:

- Clausula WHERE
- Clausula HAVING
- Clausula FROM de una sentencia SELECT o DELETE

Esta sería una pequeña guía para escribir subconsultas:

- Para las subconsultas se utilizan los operadores (en el ejemplo anterior el “>”) de comparación que se dividen en dos clases: Operadores de fila simple (>,=,>=,<,<>,<=) y operadores de filas múltiples (IN, NOT IN, ANY, ALL).
- Encerrar siempre una subconsulta entre paréntesis.
- Colocar la subconsulta después del operador de comparación.
- No agregar una clausula ORDER BY a una subconsulta. Se puede tener solamente una clausula ORDER BY y, específicamente, debe ser la última clausula en la sentecia SELECT principal.

6.1.- TIPOS DE SUBCONSULTAS

Existen varios tipos de subconsultas dependiendo del resultado de las mismas:

- Subconsulta mono-registro
- Subconsulta multi-registro
- Subconsulta multi_columna: Devuelve más de una columna de la sentencia SELECT anidada.

6.1.1.- SUBCONSULTAS MONO-REGISTRO (single-row)

Devuelve un solo registro de la sentencia SELECT anidada. Es el caso que vimos en el ejemplo del capítulo 5. Se utilizan los operadores de comparación de fila simple (>,=,>=,<,<>,<=).

Ejemplo: Obtener los empleados, alfabéticamente, que están en el mismo departamento que “Jorge”.

```
SELECT nombre, departamento FROM EMPLEADOS
WHERE departamento = ( SELECT  departamento FROM EMPLEADOS
                         WHERE codigo = 1 )
ORDER BY nombre ;
```

Ejemplo 2: Lo mismo de antes pero también queremos el nombre del departamento y el salario de cada empleado.

```
SELECT e.nombre, e.salario, e.departamento, d.nombre
FROM EMPLEADOS e, DEPARTAMENTOS d
WHERE e.departamento = d.codigo
  AND e.departamento = ( SELECT  departamento FROM EMPLEADOS
                           WHERE codigo = 1 )
ORDER BY e.nombre ;
```

Fijaros que en el ORDER BY pongo la notación completa, ya que en la SELECT hay dos campos “nombre”, y si pusiera “ORDER BY nombre” obtendríamos un error por ambigüedad ya que no se sabe por cual de los dos queremos ordenar. También lo podía haber resuelto poniendo “ORDER BY 1”.

Ejemplo 3: Lo mismo que el ejemplo 2 pero además solo queremos los empleados que ganen más que “Jorge”, obteniendo así sus compañeros de departamento que ganan más que él.

```
SELECT e.nombre, e.salario, e.departamento, d.nombre
FROM EMPLEADOS e, DEPARTAMENTOS d
WHERE e.departamento = d.codigo
  AND e.departamento = ( SELECT  departamento FROM EMPLEADOS
                           WHERE codigo = 1 )
  AND e.salario > ( SELECT  salario FROM EMPLEADOS
                      WHERE codigo = 1 )
ORDER BY e.nombre ;
```

Como podemos ver, podemos poner varias subconsultas según las necesidades del problema a resolver, y en este caso, las dos subconsultas devuelven un solo registro.

En el ejemplo del capítulo 5, vimos que se pueden usar funciones de

grupo en una subconsulta. De este modo devolvemos un único registro.

```
SELECT nombre, salario FROM EMPLEADOS
WHERE salario = ( SELECT MIN(salario) FROM EMPLEADOS ) ;
```

Con esta consulta, obtenemos los empleados que cobran el mínimo sueldo de la empresa.

Casi para terminar vamos a ver el uso de HAVING en subconsultas. Ya hemos comentado que una subconsulta puede ponerse en la clausula HAVING, así que lo mejor será un ejemplo:

```
SELECT departamento, MIN(salario) FROM EMPLEADOS
GROUP BY departamento
HAVING MIN(salario) > ( SELECT MIN(salario) FROM EMPLEADOS
                        WHERE departamento=(SELECT departamento
                                            FROM EMPLEADOS
                                            WHERE codigo=1) ) ;
```

Esta SELECT visualizaría todos los departamentos que tienen un salario mínimo mayor que el salario mínimo de los empleados del departamento de “Jorge”. Rizando el rizo hemos anidado una subconsulta dentro de otra subconsulta, además de usar la clausula HAVING.

Y por último voy a especificar dos errores típicos cuando usamos subconsultas mono-registro o “single-row”:

- Que una subconsulta devuelva más de un registro cuando se espera uno solo. Esto provoca un error ya que estaremos utilizando operadores de fila única. (En oracle obtendríamos un bonito ORA-01427: single-row subquery returns more than one row)
- Que una subconsulta no devuelva ningún registro. Debemos asegurarnos que la subconsulta devolverá un registro para que obtengamos el resultado deseado. En estos casos, la consulta principal tampoco devolverá ningún registro pero no dará ningún error.

6.1.2.- SUBCONSULTAS MULTI-REGISTRO (multiple-row)

Devuelve más de un registro de la sentencia SELECT anidada. Se utilizan los operadores de comparación de fila múltiple (IN, NOT IN, ANY, ALL). Estos operadores esperan uno o más registros.

Operadores multi-registro:

Operador	Significado
IN	Igual a los valores de cierta lista.
ANY	Compara los valores con cada valor devuelto por la subconsulta
ALL	Compara los valores con cada uno de los valores devueltos por la subconsulta.

Ejemplo (IN): Obtener los empleados que ganan un salario igual a cualquiera de los salarios mínimos por departamento.

```
SELECT nombre, salario, departamento FROM EMPLEADOS
WHERE salario IN ( SELECT MIN(salario) FROM EMPLEADOS
                   GROUP BY departamento) ;
```

La consulta interna nos devolverá varios registros, uno por departamento existente en la tabla EMPLEADOS (recordemos que no tienen porque estar todos en la tabla EMPLEADOS, pueden haber departamentos "vacios"). Si la subconsulta devolviera 4 valores: 1000, 1500, 1000 y 2000, la consulta principal se evaluaría exactamente así:

```
SELECT nombre, salario, departamento FROM EMPLEADOS
WHERE salario IN (1000,1500,1000,2000) ;
```

El operador ANY (y su sinónimo SOME) compara un valor con cada valor devuelto por la subconsulta. Por ejemplo, queremos visualizar los empleados cuyo salario es menor que el de cualquiera de los empleados del departamento 1:

```
SELECT codigo, nombre, salario FROM EMPLEADOS
WHERE salario < ANY ( SELECT salario FROM EMPLEADOS
                      WHERE departamento = 1 ) ;
```

Notas:

- < ANY significa "menor que cualquiera".
- > ANY significa "más que cualquiera".
- = ANY significa lo mismo que IN.

El operador ALL compara un valor con cada valor devuelto por una subconsulta. En el ejemplo anterior, trasladado a ALL sería:

```
SELECT codigo, nombre, salario FROM EMPLEADOS
WHERE salario < ALL ( SELECT salario FROM EMPLEADOS
                      WHERE departamento = 1 ) ;
```

y significa "obtener los empleados cuyo salario sea menor que el de todos los empleados del departamento 1. Queda clara la diferencia con ANY, ¿verdad?.

El operador NOT puede ser utilizado tanto con IN como con ANY y ALL.

6.1.3.- SUBCONSULTAS MULTI-COLUMNA

Hasta ahora en todas las subconsultas que hemos escrito utilizábamos una sola columna (campo) de la tabla para compararla en la clausula WHERE o en la clausula HAVING de la SELECT. Ahora construiremos subconsultas con más de una columna para ser comparadas mediante operadores lógicos. Esto nos permitirá transformar diferentes condiciones WHERE en una única clausula WHERE.

```
SELECT nombre, departamento, salario FROM EMPLEADOS
WHERE (departamento, salario) IN ( SELECT departamento, salario
                                   FROM EMPLEADOS
                                   WHERE jefe = 2 ) ;
```

En este caso estamos recuperando cualquier empleado cuyo departamento y salario coincidan (los dos) con el departamento y salario de cualquier empleado cuyo jefe sea el que tiene código de empleado 2.

Las comparaciones en una subconsulta multi-columna pueden ser comparaciones "par-wise" o "non-parwise". En el caso anterior estabamos ante una comparación "par-wise", es decir, cada registro debe coincidir

tanto en el departamento como en el salario con los de los empleados de cuyo jefe era el 2.

Si se quiere una comparación “non-parwise” (también llamado producto cruzado ó cross product) tenemos que usar una clausula WHERE con varias condiciones.

No entraremos en detalle respecto a estos casos ya que suponen un conocimiento más avanzado, pero baste con decir que no siempre devuelven los mismos resultados aunque a primera vista lo parezca.

6.1.4.- VALORES NULOS EN UNA SUBCONSULTA

```
SELECT e.nombre FROM EMPLEADOS e
WHERE e.codigo NOT IN ( SELECT j.jefe FROM EMPLEADOS j ) ;
```

Esta consulta intenta recuperar todos los empleados que no tienen ningún subordinado. Sin embargo no nos devolverá ningún registro ya que uno de los valores devueltos por la subconsulta es un NULL por lo que la consulta no devuelve registros. La razón es que todas las comparaciones contra un NULL devuelven NULL, por ello siempre que los valores nulos sean un resultado probable de una subconsulta, no hay que usar el operador NOT IN (que equivale a <>ALL). Utilizar el operador IN no es problema ya que equivale a =ANY y la siguiente consulta para obtener los empleados que tienen subordinados si que funcionará correctamente:

```
SELECT e.nombre FROM EMPLEADOS e
WHERE e.codigo IN ( SELECT j.jefe FROM EMPLEADOS j ) ;
```

6.1.5.- USO DE UNA SUBCONSULTA EN LA CLAUSULA FROM

Podemos usar una subconsulta como parte de la clausula FROM de una SELECT como si fuera una tabla o vista más. Las normas son las mismas que las ya explicadas, solo cambia la posición en la SELECT principal. Lo que si es “casi” imprescindible es nombrarla con un alias para poder nombrarla como “tabla”.

```
SELECT e.nombre, e.departamento, b.media
FROM EMPLEADOS, ( SELECT departamento, avg(salario) median
                    FROM EMPLEADOS
                    GROUP BY departamento ) b
WHERE e.departamento = b.departamento
  AND e.salario > b.media ;
```

Con esta SELECT estamos obteniendo los empleados cuyo salario sea mayor que la media de su departamento. Hemos hecho un join y una subconsulta con grupos.

6.1.6.- SUBCONSULTAS SINCRONIZADAS

Una subconsulta puede estructurarse para que se ejecute repetidamente, una vez por cada una de las filas consideradas para selección por la consulta principal. A esto se le llama “subconsula sincronizada”.

Se utiliza cuando una subconsulta anidada devuelve un resultado o grupo de resultados diferentes para cada fila candidata considerada para selección por una consulta externa.

- La subconsulta es dirigida por la consulta externa.
- Una subconsulta sincronizada se contruye al utilizar en la claúsula WHERE de la subconsulta, referencias a la tabla declarada en la consulta externa.
- El valor actual de la columna relevante en la SELECT externa es emparejado entonces con el valor de la columna en la subconsulta para cada una de las ejecuciones de la misma.
- Las columnas de la SELECT interna y externa pueden ser de la misma tabla o de diferentes tablas.
- Se ejecuta la subconsulta por cada una de las filas de la tabla referenciada en la SELECT externa.

Como ejemplo, volvamos a ver la SELECT que vimos en el capítulo 5 (joins), al final.

```
SELECT e.nombre EMPLEADO, d.nombre DEPART, j.nombre JEFE, s.salario SUELDO
FROM EMPLEADOS e, EMPLEADOS j, DEPARTAMENTOS d, SALARIOS s
WHERE e.departamento = d.codigo(+)
  AND e.jefe = j.codigo
  AND e.codigo = s.empleado
  AND s.fecha_salario =
          (SELECT MAX(fecha_salario) FROM SALARIOS sal
           WHERE sal.empleado = s.empleado
             AND TRUNC(fecha_salario)<=TO_DATE('31/12/03','DD/MM/RRRR'));
```

Vemos que en la subconsulta, concretamente en la condición de join, hacemos referencia a una columna de la consulta principal (en rojo), que a su vez se corresponde con el código de empleado (e.codigo).

6.1.7.- EXISTENCIA DE SUBCONSULTAS

En el apartado 6.1.4 vimos el problema ocasionado por la no devolucion de resultados en las subconsultas. Bueno pues esto puede ser aprovechado mediante la clausula EXIST en combinación con NOT.

La obtención o no de filas en una subconsulta puede utilizarse para seleccionar filas de la consulta externa o principal.

Si la subconsulta encuentra una o más filas, EXIST será cierto, y si no se devuelven registros, EXIST será falso.

Ejemplo: Obtener los hospitales que tienen sala de recuperación.

```
SELECT nombre FROM hospital h
WHERE EXIST ( SELECT nombre FROM salas s
                WHERE h.codigo = s.cod_hospital
                  AND s.nombre = 'RECUPERACION' ) ;
```

En este caso hemos supuesto dos tablas: “hospital” y “salas”, relacionadas por el código de hospital, como se puede ver en el join de la subconsulta (sincronizada, para más señas)

Con esto finalizamos el estudio básico de las subconsultas.

7.- MANIPULACION DE DATOS

El lenguaje de manipulación de datos (DML) es parte esencial del SQL. Si queremos actualizar, insertar o eliminar datos de la base de datos, tenemos que ejecutar una sentencia DML. Un conjunto de sentencias DML que aún no se han hecho permanentes se denomina transacción, de las cuales hablaremos al final.

Las sentencias DML son tres: INSERT, UPDATE y DELETE. Como es de esperar, INSERT añade un nuevo registro a una tabla (realmente puede insertar un grupo de registros de golpe), UPDATE modifica registros existentes y DELETE elimina registros ya existentes de una tabla.

Antes de comenzar con ellas, es conveniente que esquematicemos nuestra base de datos de ejemplo, con sus relaciones y "constraints" (claves primarias (PK), claves ajenas (FK), columnas NOT NULL,...) ya que vamos a hablar de "violaciones" a esas constraints, y de paso recordamos la estructura de las tablas.

EMPLEADOS

codigo	NUMBER(5)	NOT NULL, PK
nombre	VARCHAR2(50)	
dni	VARCHAR2(10)	
departamento	NUMBER(3)	FK (DEPARTAMENTOS.CODIGO)
salario	NUMBER(3)	NOT NULL
fecha_alta	DATE	NOT NULL
jefe	NUMBER(5)	FK (EMPLEADOS.CODIGO)

DEPARTAMENTOS

codigo	NUMBER(5)	NOT NULL PK
nombre	VARCHAR2(30)	

SALARIOS

empleado	NUMBER(5)	NOT NULL, PK, FK (EMPLEADOS.CODIGO)
fecha_salario	DATE	NOT NULL
salario	NUMBER(15,2)	NOT NULL

NIVEL_SALARIAL

nivel	NUMBER(5)	NOT NULL PK
salario_min	NUMBER(15,2)	NOT NULL
salario_max	NUMBER(15,2)	NOT NULL

7.1- SENTENCIA INSERT

La sintaxis básica de la sentencia INSERT es:

```
INSERT INTO tabla [(columna1 [,columna2 ...])]
      VALUES (valor1 [,valor2 ...]) ;
```

donde:

- tabla: es el nombre de la tabla donde se va a insertar el registro.
- columna*n*: es el nombre de la(s) columna(s) a cumplimentar.
- valor*n*: es el valor correspondiente a esa columna (campo).

Ejemplo:

```
INSERT INTO DEPARTAMENTOS (codigo, nombre)
      VALUES (7,'ADMINISTRACION') ;
```

Si insertamos una fila con todos los valores de los campos informados, podríamnos no poner la lista de campos, pero entonces la lista de valores a insertar debe estar obligatoriamente en el orden que por defecto tienen las columnas o campos en la tabla. ¿Como averiguamos este orden? Con el comando DESCRIBE de SQL que nos da la estructura de una tabla:

```
SQL> DESCRIBE EMPLEADOS

Nombre                  Null?           Tipo
codigo                  NOT NULL        NUMBER(5)
nombre                                  VARCHAR2(50)
```

Es decir, que podíamos haber escrito la sentencia INSERT anterior así:

```
INSERT INTO DEPARTAMENTOS VALUES (7,'ADMINISTRACION') ;
```

Como vemos en la salida del comando DESCRIBE, hay una columna que NO puede ser NULL, la cual es la PRIMARY KEY de la tabla (Ya dijimos al comienzo del curso que una primary key no puede ser nula) y otra columna que sí puede ser nula. Vamos, que podemos crear un departamento sin “nombre”. Esto nos lleva a pensar como haríamos para insertar un registro con esa columan a NULL. Pues bien, hay dos formas. La primera consiste en, simplemente, no especificar esa columna en la sentencia INSERT, y la segunda en especificar el valor NULL como valor para dicha columna:

```
INSERT INTO DEPARTAMENTOS (codigo) VALUES (7) ;

INSERT INTO DEPARTAMENTOS (codigo, nombre)
      VALUES (7,NULL) ;
```

Es muy importante asegurarnos que una columna admite valores NULL a la hora de ejecutar un INSERT ya que cualquier columna no listada específicamente en la sentencia obtendrá un valor NULL en el nuevo registro, y si no se permite, lo que obtendremos será un bonito mensaje de error de la base de datos.

Otro punto a tener en cuenta, y más evidente, es que no podemos insertar filas duplicadas en su(s) columna(s) de la primary key. Es decir, no podemos, en el caso de este ejemplo, insertar dos departamentos con el mismo “codigo” ya que también obtendríamos un error de la base de datos, en este caso, un error de integridad de “clave primaria violada”.

Para insertar valores de fecha debemos utilizar la función de conversión TO_DATE() vista en capítulos anteriores, ya que al escribir la sentencia INSERT debemos convertir el valor introducido a tipo fecha.

```
INSERT INTO SALARIOS (empleado, fecha_salario, salario)
      VALUES(2, TO_DATE('14/02/2004','DD/MM/RRRR'), 1550.35) ;
```

Para insertar valores especiales en registros de la base de datos, podemos usar funciones. Lo primero que se me viene a la cabeza es querer insertar la fecha actual en un campo de una tabla. Para ello, en Oracle disponemos de la función SYSDATE (otras BD tendrán su equivalente).

```
INSERT INTO SALARIOS (empleado, fecha_salario, salario)
      VALUES(3, SYSDATE, 1200.75) ;
```

Ya vimos en los formatos de fecha, que este tipo de dato tiene componente horario (HH:MI:SS). Si no se especifica en el formato de inserción, por defecto se toma la medianoche (00:00:00).

Otra cuestión importante respecto a la integridad referencial es que, en estos dos últimos ejemplos, estamos suponiendo que YA EXISTEN unos empleados en la tabla EMPLEADOS con “codigo” 2 y 3 respectivamente, ya que en caso contrario también obtendríamos un error de integridad, en este caso de “Clave ajena violada. No se encuentra registro padre.”

7.1.1- INSERTAR MEDIANTE SUBCONSULTA

En este caso se trata de introducir un conjunto de registros con valores extraidos de otra(s) tabla(s). En lugar de la clausula VALUES se utiliza una subconsulta.

```
INSERT INTO tabla [(columna1, [columna2, ...])] subconsulta ;
```

Lo importante es que el número de columnas de la lista del INSERT debe coincidir en número y tipo con los valores devueltos por *subconsulta*.

Para el siguiente ejemplo, supondremos creada una tabla llamada CONTABLES con la siguiente estructura:

EMPLEADOS

codigo	NUMBER(5)	NOT NULL, PK
nombre	VARCHAR2(50)	
dni	VARCHAR2(10)	
salario	NUMBER(3)	NOT NULL
fecha_alta	DATE	NOT NULL

Como podeis podría tratarse de una tabla auxiliar que nos ha pedido nuestro jefe con todos los empleados del departamento contable (codigo de departamento 5). Debemos dársela con los datos correspondientes.

```
INSERT INTO CONTABLES SELECT codigo, nombre, dni, salario,
                             fecha_alta
                      FROM EMPLEADOS
                      WHERE departamento = 5 ;
```

Hay que decir que la subconsulta puede devolver campos de distintas tablas y, de este modo, rellenar una tabla resumen para que nuestro jefe disponga de los datos que necesite en una sola tabla. Por ejemplo, ahora nuestro bien amado y nunca bien ponderado jefe quiere una tabla con todos los datos referentes a los empleados que ganen más de 1500 euros de salario. Claro está, no le sirve el código de departamento, quiere el nombre y tampoco le vale el código del jefe de cada empleado sino su nombre. Para ello crearíamos una tabla RESUMEN_EMPLEADOS con las siguientes columnas:

RESUMEN_EMPLEADOS

codigo	NUMBER(5)	NOT NULL, PK
nombre	VARCHAR2(50)	
dni	VARCHAR2(10)	
nombre_dep	VARCHAR2(30)	
salario	NUMBER(3)	NOT NULL
fecha_alta	DATE	NOT NULL
nom_jefe	VARCHAR2(50)	

Y la rellenaríamos con la siguiente sentencia INSERT:

```
INSERT INTO RESUMEN_EMPLEADOS SELECT e.codigo, e.nombre, e.dni,
                                     d.nombre,  e.salario,  e.fecha_alta,
                                     jefes.nombre
                              FROM EMPLEADOS e, DEPARTAMENTOS d,
                                   EMPLEADOS jefes
                              WHERE e.departamento = d.codigo
                                AND e.jefe = jefes.codigo
                                AND e.salario > 1500 ;
```

(No vamos a ver la sentencia CREATE TABLE, pero solo hacer mención que podemos crear la tabla directamente de este modo solo con sustituir INSERT INTO RESUMEN_EMPLEADOS por CREATE TABLE RESUMEN_EMPLEADOS AS)

7.2- SENTENCIA UPDATE

La sentencia UPDATE nos servirá para modificar datos de columnas de las tablas en registros ya existentes, por supuesto. Esta sentencia puede afectar a más de un registro al mismo tiempo, y esto que la hace muy potente, la convierte también en peligrosa si somos, como yo digo, de "gatillo rápido" (Vamos, que se pulsa en Intro rápidamente sin comprobar las sentencias). De todas formas tranquilos: siempre nos quedará el ROLLBACK si es que aún no hemos efectuado el COMMIT (veremos los dos comandos en el capítulo de Transacciones).

La sintáxis básica de UPDATE es:

UPDATE *tabla* SET *columna*=*valor* [,*columna*=*valor*,...]
[WHERE *condicion*] ;

- *tabla* = Tabla a actualizar
- *columna* = columna (campo) de la tabla a actualizar
- *valor* = valor o subconsulta para la columna
- *condicion* = Condición que identifica las filas que han de ser actualizadas. Si se omite, se actualizarán TODAS las filas de la tabla. Puede estar basada en una subconsulta.

Si sólo queremos actualizar sobre un registro, normalmente usaremos la primary key de la tabla para identificar ese registro. El uso de otras columnas puede causar que inesperadamente se actualicen varias filas. Por ejemplo, identificar en la tabla EMPLEADOS un registro por su nombre puede ser peligroso ya que pueden haber varios empleados con el mismo nombre.

Ejemplo: Pasar al empleado 1 al departamento 5.

```
UPDATE EMPLEADOS SET departamento=5
WHERE codigo = 1 ;
```

Ejemplo: Subir un 5% el salario de todos los empleados del departamento 5.

```
UPDATE EMPLEADOS SET salario=salario*1.05
WHERE departamento = 5 ;
```

Ejemplo: Subir el IPC (2%) a todos los empleados.

```
UPDATE EMPLEADOS SET salario=salario*1.02 ;
```

7.2.1- UPDATE CON SUBCONSULTAS

Simplemente se trata de implementar una subconsulta en la clausula SET de la sentencia UPDATE ó en la clausula WHERE.

Ejemplo: Modificar el departamento y el salario del empleado 25 con los valores actuales del empleado 10.

```
UPDATE EMPLEADOS
      SET (departamento, salario) = (SELECT departamento, salario
                                     FROM EMPLEADOS
                                     WHERE codigo=10)
      WHERE codigo = 25 ;
```

También podemos usar las subconsultas en la clausula WHERE tal y como vemos en el siguiente ejemplo, que cambia el salario a todos los empleados del departamento del empleado 10, por el salario del mismo empleado 10.

```
UPDATE EMPLEADOS SET salario = ( SELECT salario FROM EMPLEADOS
                                 WHERE codigo=10 )
      WHERE departamento = ( SELECT departamento FROM EMPLEADOS
                             WHERE codigo=10) ;
```

Hacer notar que las subconsultas pueden ser construidas utilizando otras tablas, siempre y cuando el tipo de datos a actualizar ó el especificado en la clausula WHERE coincida.

Por supuesto, la integridad referencial de la base de datos es prioritaria, y no podríamos, por ejemplo, actualizar el campo departamento de la tabla EMPLEADOS con un valor inexistente en la tabla DEPARTAMENTOS, ya que ese campo es una clave ajena (FK) del campo de la tabla DEPARTAMENTOS. Si lo intentamos en Oracle obtendremos un error ORA-02291, “Error de integridad. Clave padre no encontrada”.

7.3- SENTENCIA DELETE

Como ya habréis deducido, la sentencia DELETE sirve para eliminar registros de una tabla. Al igual que UPDATE, la sentencia DELETE es “peligrosa” ya que podemos borrar registros no deseados si nos equivocamos al escribirla. Igualmente, es aconsejable comprobar lo realizado antes de efectuar el COMMIT, y en caso de error, hacer un ROLLBACK (insisto que estos comandos los veremos en el capítulo siguiente de Transacciones).

La sintaxis de la sentencia DELETE es:

```
DELETE [FROM] tabla [WHERE condicion] ;
```

- *table*: Nombre de la tabla de la cual queremos borrar registros
- *condicion*: Condición que identifica las filas que han de ser borradas. Si se omite, se actualizarán TODAS las filas de la tabla. Puede estar basada en una subconsulta.

Como ya hemos dicho, los registros a borrar se especifican en la clausula WHERE:

```
DELETE FROM DEPARTAMENTOS WHERE nombre IS NULL ;
```

Esto borraría los departamentos cuyo nombre esté a nulo.

Si no ponemos clausula where, borraremos TODOS los registros:

```
DELETE FROM EMPLEADOS;
```

Y un ejemplo con subconsulta accediend a otra tabla:

```
DELETE FROM EMPLEADOS WHERE departamento = (SELECT departamento
                                            FROM DEPARTAMENTOS
                                            WHERE nombre='VENTAS');
```

Mucho cuidado con las restricciones de integridad. En el primer caso, si alguno de los departamentos con nombre nulo está referenciado en la tabla EMPLEADOS, obtendríamos en Oracle un ORA-02292, violación de integridad. Registro hijo encontrado.

Este último error de integridad puede ser “evitado”. Lo digo entre comillas porque el método es útil en muchas ocasiones pero peligroso en otras. Podríamos crear la restricción de integridad con la “coletilla” DELETE CASCADE. Esto significa que cuando borramos un registro que tiene referencias en registros de otras tablas, estos tambien se borrarán sin preguntar. Vamos, que si borramos un “padre”, borramos de tirón a todos sus “hijos”. Imaginaros el descalabro que se puede montar en caso de equivocación. Sin embargo la condición de DELETE CASCADE es útil en otras situaciones.

Moraleja: La potencia proporciona poder y hay que saber utilizarlo.

8.- TRANSACCIONES EN LA BASE DE DATOS

En este capítulo si que me voy a ceñir más a Oracle, ya que es la que he manejado mucho y la que conozco bastante bien. De todas formas, los conceptos son los mismos y puede variar el método de hacer las cosas, así que he preferido incluir este capítulo aunque sea un poco "dedicado".

El servidor de base de datos asegura la consistencia de los datos utilizando transacciones. Las transacciones permiten tener el control cuando los datos cambian y aseguran la consistencia de los mismos ante un fallo del proceso que esté ejecutando el usuario ó un fallo del sistema.

Las transacciones son un conjunto de instrucciones DML (data manipulation language) que efectúan cambios en los datos, ya sean actualizaciones, inserciones y/o borrados (también hay transacciones con DDL, aunque nos centraremos en las de DML).

Para entenderlo mejor, pensemos en una transferencia bancaria entre dos cuentas: Primero sacamos el dinero de una cuenta y luego lo introducimos en otra. Las dos acciones se deben realizar. Si una falla, la otra se debe deshacer o no efectuarse. Eso es lo que pretende conseguir una transacción: consistencia.

Una transacción comienza cuando se realiza el primer comando SQL y termina cuando ocurre uno de los siguientes eventos:

- COMMIT ó ROLLBACK
- Un comando DDL como CREATE (se realiza un COMMIT automático)
- Detección de algunos errores muy específicos (deadlocks: bloqueos de la muerte ó abrazo mortal)
- Fin de la sesión del usuario
- Fallo o caída del sistema

Después de que finalice una transacción, se inicia la siguiente con la próxima sentencia SQL si la hubiera.

Un COMMIT hace permanentes los cambios efectuados en la transacción actual y la finaliza. Hasta que se efectúa el COMMIT, el usuario que lanzó la transacción ve los datos modificados en su sesión. Los demás usuarios no. Es decir: si yo inserto tres registros y actualizo otro, veré los cambios con la SELECT correspondiente, pero esos cambios aún no se han grabado en la base de datos. Para los demás usuarios no ha ocurrido nada todavía. Si uno de esos otros usuarios intenta modificar un registro alterado por mí antes de que haya hecho el COMMIT, tendrá el registro bloqueado y esperará hasta que mi transacción termine.

Un ROLLBACK [TO SAVEPOINT *nombre_savepoint*] finaliza la transacción en curso descartando todos los cambios pendientes. Es decir: una vuelta atrás, como su nombre indica. Esta vuelta atrás puede ser hasta un punto determinado por la clausula [TO SAVEPOINT nombre_savepoint] que se especifica en un programa. Esta clausula NO es un estándard de SQL ANSI.

En caso de caída del sistema se realiza, de forma automática, un ROLLBACK. Esto hace que el fallo no provoque cambios no deseados ó parciales y deja las tablas en su estado inicial desde el último COMMIT, protegiendo así la integridad de las tablas.

8.1.- ESTADO DE LOS DATOS ANTES DE COMMIT O ROLLBACK

Cada cambio realizado en los datos durante una transacción es temporal hasta que se efectúa un COMMIT. Estas operaciones que vamos efectuando afectan al búffer de la base de datos, por lo que el estado previo de los datos puede ser recuperado.

El usuario que efectúa la transacción puede visualizar los cambios mediante select's ya que para él son totalmente visibles. Los demás usuarios no pueden ver los cambios que se están realizando. Esto se denomina consistencia en lectura, ya que los demás usuarios ven los datos como están desde el último COMMIT (hay que tener en cuenta que el usuario que realiza la transacción podría arrepentirse y realizar un ROLLBACK).

Por último, los registros afectados por la transacción son bloqueados para que otros usuarios no puedan realizar cambios sobre ellos.

Como ejemplo haremos que el usuario 1 modifique el nombre del empleado cuyo código es el 35 y que borre un registro determinado. Observaremos como el usuario 2 no “ve” los cambios ya que no se ha efectuado el COMMIT.

Usuario 1	Usuario 2
SELECT nombre FROM EMPLEADOS WHERE codigo=35;	SELECT nombre FROM EMPLEADOS WHERE codigo=35;
Resultado: 'Fulanito'	Resultado: 'Fulanito'
UPDATE EMPLEADOS SET nombre='Pepe' WHERE codigo=35; Resultado: 1 registro actualizado.	
SELECT nombre FROM EMPLEADOS WHERE codigo=35;	SELECT nombre FROM EMPLEADOS WHERE codigo=35;
Resultado: 'Pepe'	Resultado: 'Fulanito'
SELECT COUNT(*) FROM EMPLEADOS;	SELECT COUNT(*) FROM EMPLEADOS;
Resultado: 40 registros.	Resultado: 40 registros.
DELETE FROM EMPLEADOS WHERE codigo=25;	
SELECT COUNT(*) FROM EMPLEADOS;	SELECT COUNT(*) FROM EMPLEADOS;
Resultado: 39 registros.	Resultado: 40 registros.

8.2.- ESTADO DE LOS DATOS DESPUES DE COMMIT

Ya hemos dicho que COMMIT hace permanentes los cambios pendientes. Estos cambios se escriben en la base de datos, perdiéndose así definitivamente el estado anterior de los datos. Ahora todos los usuarios pueden ver los datos resultantes tras la transacción ya que se han escrito realmente en la base de datos. Por supuesto se eliminan los bloqueos de los registros afectados por lo que los demás usuarios ya pueden modificarlos. Por último se borran todos los SAVEPOINT's si los había.

Usuario 1	Usuario 2
COMMIT Commit completado	
SELECT nombre FROM EMPLEADOS WHERE codigo=35;	SELECT nombre FROM EMPLEADOS WHERE codigo=35;
Resultado: 'Pepe'	Resultado: 'Pepe'
SELECT COUNT(*) FROM EMPLEADOS;	SELECT COUNT(*) FROM EMPLEADOS;
Resultado: 39 registros.	Resultado: 39 registros.

El usuario 2 sólo “ve” los cambios cuando el usuario 1 realiza el COMMIT. Antes de esto, para él, no habrá cambios.

8.3.- ESTADO DE LOS DATOS DESPUES DE ROLLBACK

También hemos comentado que ROLLBACK deshace los cambios pendientes. Se restaura los valores anteriores de los datos y se eliminan los bloqueos.

Si suponemos realizada la transacción del ejemplo anterior (punto 8.1) y el usuario 1 ahora hace un ROLLBACK tendríamos:

Usuario 1	Usuario 2
ROLLBACK	
Resultado: Rollback completado.	
SELECT nombre FROM EMPLEADOS WHERE codigo=35;	SELECT nombre FROM EMPLEADOS WHERE codigo=35;
Resultado: 'Fulanito'	Resultado: 'Fulanito'
SELECT COUNT(*) FROM EMPLEADOS;	SELECT COUNT(*) FROM EMPLEADOS;
Resultado: 40 registros.	Resultado: 40 registros.

Es decir: No ha cambiado nada desde la situación inicial y para el usuario 2 NUNCA ha ocurrido nada, excepto que si intentó modificar el mismo registro que cambió el usuario 1 (o el que borró) se hubiera quedado esperando a que se efectuara el COMMIT o el ROLLBACK de la transacción, ya que esos registros se encontraban bloqueados.

Existe también el llamado ROLLBACK a nivel de sentencia. Este se produce cuando la propia sentencia DML falla mientras se ejecuta. En este

caso se efectúa un ROLLBACK sobre los datos afectados por esa sentencia. Esto es útil en actualizaciones y/o borrados masivos de registros, ya que si la sentencia falla por ejemplo por una restricción de identidad, se deshace el efecto total de la sentencia. Si antes de esta sentencia habían cambios pendientes provocados por otra(s) sentencia(s), éstos SE MANTIENEN. Sólo se deshacen los cambios de la sentencia que ha fallado.

Los citados cambios pendientes que se mantienen pueden ser confirmados con COMMIT o deshechos con ROLLBACK por el usuario, según desee.

En el caso de Oracle, se realiza un COMMIT implícito antes y después de una sentencia DDL (como puede ser CREATE TABLE) por lo que, aunque esta sentencia falle, los cambios pendientes anteriores no podrán ser deshechos.

8.4.- CONSISTENCIA EN LECTURA

Los usuarios de una base de datos realizan dos tipos de acceso a la base de datos:

- Operaciones de lectura (SELECT)
- Operaciones de escritura (INSERT, UPDATE, DELETE)

Por lo tanto es necesario que:

- Las lecturas y escrituras a la base de datos aseguren una vista consistente de los datos en cualquier circunstancia.
- Las lecturas no vean datos que están en proceso de cambio.
- Las escrituras aseguren que los cambios a la base de datos se hacen de forma consistente.
- Los cambios realizados por una escritura no crean conflictos con los cambios que otra escritura esté realizando.
- Se asegure que, sobre los mismos datos, las lecturas no esperan a las escrituras y viceversa.

El propósito de la consistencia en lectura es asegurar que cada usuario ve los datos tal y como están desde el último COMMIT, antes de que comience una operación DML. Y ¿como se implementa esto en Oracle?. Pues con los “segmentos de rollback” o “segmentos de deshacer”.

La consistencia en lectura es automática. Se mantiene una copia parcial de la base de datos en segmentos de rollback.

Me explico:

Cuando una operación de INSERT, UPDATE ó DELETE se realiza contra la base de datos, el servidor hace una copia de los datos afectados antes de su cambio en un segmento de rollback.

Todas las lecturas, excepto las de la sesión que realiza el cambio, ven la base de datos como estaba ya que leen del segmento de rollback si los datos coinciden. Antes de hacer el COMMIT de los cambios, sólo el usuario (sesión) que los realiza ve los cambios, el resto lee del segmento de rollback. Así se garantiza la consistencia en lectura.

Al hacer el COMMIT, los cambios se hacen visibles a cualquiera que ejecute una SELECT, ya que se vacía el segmento de rollback (el espacio ocupado por los datos cambiados) para poder ser reutilizado.

Si se efectua un ROLLBACK los cambios se deshacen, es decir, se escriben los datos que están en el segmento de rollback (que son los valores anteriores a los cambios) en la base de datos, y todos los usuarios ven los datos como estaban antes de comenzar la transacción.

No hemos hablado de bloqueos casi, ni es mi propósito, pero solo para los inquietos decir que:

1. Previenen conflictos entre transacciones que acceden a los mismos recursos (como pueden ser tablas o registros).
2. Los bloqueos en Oracle son automáticos. No requieren acciones por parte del usuario y (según Oracle) ofrecen máxima concurrencia y máxima integridad.
3. Existen también bloqueos manuales por parte del usuario. (Una SELECT con la clausula FOR UPDATE lo logra)
4. Se mantienen mientras dura una transacción.
5. Existen bloqueos exclusivos (escritura) y compartidos (de lectura)

Respecto al punto 3, sí voy a explicar la SELECT FOR UPDATE ya que es algo que se suele usar.

Imaginemos una tabla con un contador que nos dará el próximo número de factura. Independientemente de que esto sea una práctica aconsejable o no, cuando queramos crear un nueva factura, deberemos acudir a esta tabla y obtener el número, crear la factura con ese número, incrementarlo en una unidad y volverlo a grabar en la tabla. Mientras hacemos esto debemos asegurarnos que NADIE puede modificar ni obtener ese número del contador, ya que podría interferir con nuestra transacción y darse el caso de intentar insertar dos facturas con el mismo número. ¿Cómo hacemos para solucionar esto?. Con una SELECT y la clausula FOR UPDATE.

Ejemplo:

Tenemos la tabla CONFIGURACION con la siguiente estructura:

CONFIGURACION

empresa	NUMBER(5)	NOT NULL, PK
cont_facturas	NUMBER(15)	NOT NULL
cont_albaranes	NUMBER(15)	NOT NULL
cont_pedidos	NUMBER(15)	NOT NULL
cont_presupues	NUMBER(15)	NOT NULL
cont_clientes	NUMBER(15)	NOT NULL
cont_proveedores	NUMBER(15)	NOT NULL
cont_vendedores	NUMBER(5)	NOT NULL
cont_empleados	NUMBER(5)	NOT NULL

Al hacer:

```
SELECT cont_facturas FROM CONFIGURACION
  WHERE empresa = 1 FOR UPDATE;
```

Estamos bloqueando el registro **entero** de la empresa 1. De esta forma nadie va a molestarnos mientras hacemos el resto de la transacción.

Pero ¿qué pasa con el resto de campos?. ¿Por qué bloquearlos si no van a influir en nuestro proceso?. Pues es verdad ¿no?. Bien, para eso podemos añadir "algo más" a la clausula FOR UPDATE.

```
SELECT cont_facturas FROM CONFIGURACION
  WHERE empresa = 1 FOR UPDATE OF cont_facturas;
```

Así sólo bloqueamos el campo *cont_facturas* del registro de la empresa 1, dejando la posibilidad de que otros usuarios accedan y/o actualicen los contadores restantes si los necesitan. Sencillo eh?.

9.- ANEXOS FINALES.

Se que me he dejado cosas en el tintero, muchas de ellas a propósito ya que requieren algún conocimiento adicional. El objetivo de este curso era iniciaros al SQL y nunca ha pretendido ser un curso completo. En este sentido creo que ha sido un resúmen útil y que se puede ampliar profundizando mucho más en cuanto a las sentencias SELECT se refiere, además de ver algún caso de UPDATE complicadillo, pero comoya he dicho no era el objetivo de este curso.

Aún así voy a dejar dos apuntes más, muy por encima también. Unos se utilizan bastante (las uniones) y otros no, pero son bastante curiosos (recuperación jerárquica).

9.1- EXPRESIONES CON SELECTS

Es exactamente lo que parece: podemos formar expresiones con sentencias SELECT mediante operadores y conjuntos (filas resultantes de cualquier sentencia SELECT válida, incluyendo el conjunto vacío), y los operadores en este caso, son:

UNION: Combina todas las filas del primer conjunto con todas las filas del segundo conjunto. Se eliminan los resultados duplicados.

```
SELECT nombre FROM EMPLEADOS
  WHERE jefe = 1
UNION
SELECT nombre FROM EMPLEADOS
  WHERE departamento = 5 ;
```

Con este ejemplo, obtendríamos los empleados cuyo jefe es el 1 más los empleados del departamento 5.

IMPORTANTE: las columnas deben ser compatibles en los dos conjuntos de datos. Es decir: del mismo tipo. Si utilizamos "alias" de campo, facilitaremos más el uso de UNION, ya que podemos obtener y unir campos de distinto nombre y diferentes tablas, pero del mismo tipo y mismo "alias" para poder ser usado (el alias) luego como variable.

```
SELECT nombre DENOMINACION FROM empleados WHERE jefe = 1
UNION
SELECT nom_cli DENOMINACION FROM clientes
   WHERE cod_vendedor = 5
ORDER BY 1 ;
```

Los conjuntos NO pueden llevar ORDER BY, pero sí la UNION total.

INTERSECT: Nos devolverá las filas comunes, es decir, las que aparezcan en ambos conjuntos. Las filas duplicadas resultantes se eliminarán antes de obtener el resultado final.

```
SELECT nombre FROM EMPLEADOS
  WHERE jefe = 1
INTERSECT
SELECT nombre FROM EMPLEADOS
  WHERE departamento = 5 ;
```

MINUS: Devuelve las filas que están en el primer conjunto pero no en el segundo. Las filas duplicadas en el primer conjunto se reducirán a una única antes de realizarse la comparación con el segundo conjunto.

```
SELECT nombre FROM EMPLEADOS
  WHERE jefe = 1
MINUS
SELECT nombre FROM EMPLEADOS
  WHERE departamento = 5 ;
```

Así obtenemos los empleados cuyo jefe es el 1 pero no pertenecen al departamento 5.

Estos dos ejemplos últimos se pueden realizar con una sola SELECT, pero os dejo que penséis las utilidades que le podemos dar usando diferentes tablas en cada conjunto.

9.1.1- REGLAS PARA LOS OPERADORES DE CONJUNTOS

- Pueden ser encadenados en cualquier combinación.
- Se evalúan de derecha a izquierda.
- Podemos forzar la precedencia mediante paréntesis.
- Los operadores de conjuntos pueden ser usados con conjuntos de diferentes tablas siempre que se cumpla:

    1. El numero de columnas debe ser el mismo en todos los conjuntos
    2. Los nombres de las columnas son irrelevantes
    3. Los tipos de datos deben coincidir

9.2- RECUPERACION JERARQUICA DE DATOS

Si revisamos la tabla de empleados, veremos que tenemos dos campos, codigo y jefe, que relacionan la tabla consigo misma. Con esto conseguíamos almacenar y poder recuperar el jefe de cada empleado como un codigo más de empleado (acordaros de la SELF-JOIN).

Pues bien, a esto se le llama “relación de jerarquía” y podríamos representarla mediante un árbol cuyo nodo superior fuera el “Director General” y fuera bajando a través de los diferentes jefes y empleados subordinados a su cargo.

Para recuperar los datos mediante una SELECT siguiendo una estructura de árbol, usaremos las clausulas CONNECT BY y START WITH de la sentencia SELECT.

CONNECT BY: Es el criterio que define la estructura del árbol.
START WITH: Identifica la raíz del árbol que queremos obtener.

Ejemplo:

```
SELECT codigo, nombre, departamento, jefe
FROM EMPLEADOS
CONNECT BY PRIOR codigo=jefe
START WITH codigo= 1 ;
```

Suponiendo que el empleado con código 1 fuera el Director General, obtendríamos el árbol jerárquico completo de la empresa. Si el empleado cuyo código es 1 es el jefe del departamento de Contabilidad, obtendremos el árbol jerárquico del citado departamento.

En: http://www.jlcomp.demon.co.uk/faq/connectby.html
tenéis algunos ejemplos completos sobre este tipo de recuperación, incluyendo el campo "level" interno que se puede usar para indentar la salida de la select y darle un aspecto jerárquico "auténtico".

Seguro que buscando en google encontráis muchos más ejemplos.

10.- AGRADECIMIENTOS

No me extenderé mucho, pero no puedo acabar el curso sin agradecer a mi mujer su paciencia por estar "currándome" en casa algo referente al trabajo.

También quiero agradecer a los usuarios del foro de hackxcrack su interés por el curso, ya que sino, probablemente esto hubiera acabado en unos cuantos apuntes en varios folios para poder impartirlo rápidamente en el trabajo.

Para terminar voy a dedicárselo (aunque quede "cursi") a mi futura hija Lucía y a mi compañero José Vicente González fallecido recientemente y cuyos apuntes de SQL forman parte de la bibliografía de este curso.

11.- BIBLIOGRAFIA

Curso de Introdución de SQL ANSI de IBM.- APC S.L. Jose Vte González
Oracle/SQL Tutorial .- Michael Gertz
Manuales Oracle, currículum de desarrollador.- Oracle Corp.
Apuntes de los cursos de Certificación Oracle.- Jorge Navarrete

Jorge Navarrete Olmos
jorge@navarreteolmos.com
Febrero de 2004
Valencia