

Manual de SQL para Oracle 9i

Manual de referencia

Autor: Jorge Sánchez (www.jorgesanchez.net) año 2004
e-mail: info@jorgesanchez.net

SOME RIGHTS RESERVED



Este trabajo está protegido bajo una licencia de **Creative Commons** del tipo **Attribution-NonCommercial-ShareAlike**.

Para ver una copia de esta licencia visite:

<http://creativecommons.org/licenses/by-nc-sa/2.0/>

o envíe una carta a:

**Creative Commons, 559 Nathan Abbott Way, Stanford, California
94305, USA.**



Los contenidos de este documento están protegidos bajo una licencia de **Creative Commons** del tipo **Attribution-Noncomercial-Share Alike**. Con esta licencia:

Eres libre de:

- Copiar, distribuir y mostrar este trabajo
- Realizar modificaciones de este trabajo

Bajo las siguientes condiciones:



Attribution (Reconocimiento). Debe figurar siempre el autor original de este trabajo



Noncommercial (No comercial). No puedes utilizar este trabajo con propósitos comerciales.



Share Alike (Compartir igual). Si modificas, alteras o construyes nuevos trabajos a partir de este, debes distribuir tu trabajo con una licencia idéntica a ésta

- Si estas limitaciones son incompatible con tu objetivo, puedes contactar con el autor para solicitar el permiso correspondiente
- No obstante tu derecho a un uso justo y legítimo de la obra, así como derechos no se ven de manera alguna afectados por lo anteriormente expuesto.

Esta nota no es la licencia completa de la obra, sino una traducción del resumen en formato comprensible del texto legal. La licencia original completa (jurídicamente válida y pendiente de su traducción oficial al español) está disponible en <http://creativecommons.org/licenses/by-nc-sa/2.0/legalcode>

índice

índice	5
notas previas	7
introducción	9
Historia del lenguaje SQL.....	9
estructura del lenguaje SQL.....	12
normas de escritura	12
tablas	13
esquemas de usuario y objetos	13
creación de tablas	13
orden DESCRIBE	14
orden INSERT	14
consultar las tablas del usuario	14
borrar tablas	14
tipos de datos	15
modificar tablas.....	18
valor por defecto	19
restricciones	19
consultas SELECT	27
capacidades	27
sintaxis sencilla	27
cálculos	27
condiciones.....	28
ordenación	31
funciones	31
obtener datos de múltiples tablas.....	39
agrupaciones	43
subconsultas	46
combinaciones especiales	48
comandos internos en SQL e iSQL*Plus	50
variables de sustitución	50
comando SET.....	51
encabezado y pie de informe.....	52

COLUMN	53
BREAK.....	54
COMPUTE	55
DML	59
introducción	59
inserción de datos	59
actualización de registros.....	60
borrado de registros	61
comando MERGE	61
transacciones.....	63
objetos de la base de datos	65
vistas.....	65
secuencias.....	67
índices	69
sinónimos.....	70
consultas avanzadas	73
consultas con ROWNUM.....	73
consultas sobre estructuras jerárquicas.....	73
subconsultas avanzadas.....	76
consultas de agrupación avanzada.....	77

notas previas

En este manual en muchos apartados se indica sintaxis de comandos. Esta sintaxis sirve para aprender a utilizar el comando, e indica la forma de escribir dicho comando en el programa utilizado para escribir SQL.

En el presente manual la sintaxis de los comandos se escribe en párrafos sombreados de gris con el reborde en gris oscuro. Ejemplo:

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

Otras veces se describen códigos de ejemplo de un comando. Los ejemplos se escriben también con fondo gris, pero sin el reborde. Ejemplo:

```
SELECT nombre FROM cliente;
```

Los ejemplos sirven para escenificar una instrucción concreta, la sintaxis se utiliza para indicar las posibilidades de un comando. Para indicar la sintaxis de un comando se usan símbolos especiales. Los símbolos que utiliza este libro (de acuerdo con la sintaxis que se utiliza normalmente en cualquier documentación de este tipo) son:

- ⦿ **PALABRA** Cuando en la sintaxis se utiliza una palabra en negrita, significa que es un comando que hay que escribir literalmente.
- ⦿ *texto* El texto que aparece en cursiva sirve para indicar que no hay que escribirle literalmente, sino que se refiere a un tipo de elemento que se puede utilizar en el comando. Ejemplo:

```
SELECT columna FROM tabla;
```

El texto *columna* hay que cambiarlo por un nombre concreto de columna (nombre, apellidos,...) , al igual que *tabla* se refiere a un nombre de tabla concreto.

- ⦿ **[] (corchetes)**. Los corchetes sirven para encerrar texto que no es obligatorio en el comando, es decir para indicar una parte opcional.
- ⦿ **| (barra vertical)**. Este símbolo (|) , la barra vertical, indica opción, es decir que se puede elegir entre varias opciones
- ⦿ **... (puntos suspensivos)** Indica que se puede repetir el texto anterior en el comando continuamente (significaría, *y así sucesivamente*)
- ⦿ **{ } (llaves)** Las llaves sirven para indicar opciones mutuamente exclusivas pero obligatorias. Es decir, opciones de las que sólo se puede elegir una opción, pero de las que es obligado elegir una. Ejemplo:

```
SELECT { * | columna | expresión }  
FROM tabla;
```

El ejemplo anterior indicaría que se debe elegir obligatoriamente el asterisco o un nombre de columna o una expresión. Si las llaves del ejemplo fueran corchetes, entonces indicarían que incluso podría no aparecer ninguna opción.

introducción

Historia del lenguaje SQL

El nacimiento del lenguaje SQL data de 1970 cuando E. F. Codd publica su libro: "*Un modelo de datos relacional para grandes bancos de datos compartidos*". Ese libro dictaría las directrices de las bases de datos relacionales. Apenas dos años después IBM (para quien trabajaba Codd) utiliza las directrices de Codd para crear el **Standard English Query Language (Lenguaje Estándar Inglés para Consultas)** al que se le llamó **SEQUEL**. Más adelante se le asignaron las siglas SQL (aunque en inglés se siguen pronunciando SEQUEL, en español se le llama *esecuele*).

Poco después se convertía en un estándar en el mundo de las bases de datos avalando por los organismos ISO y ANSI. Aún hoy sigue siendo uno de los estándares más importantes de la industria informática.

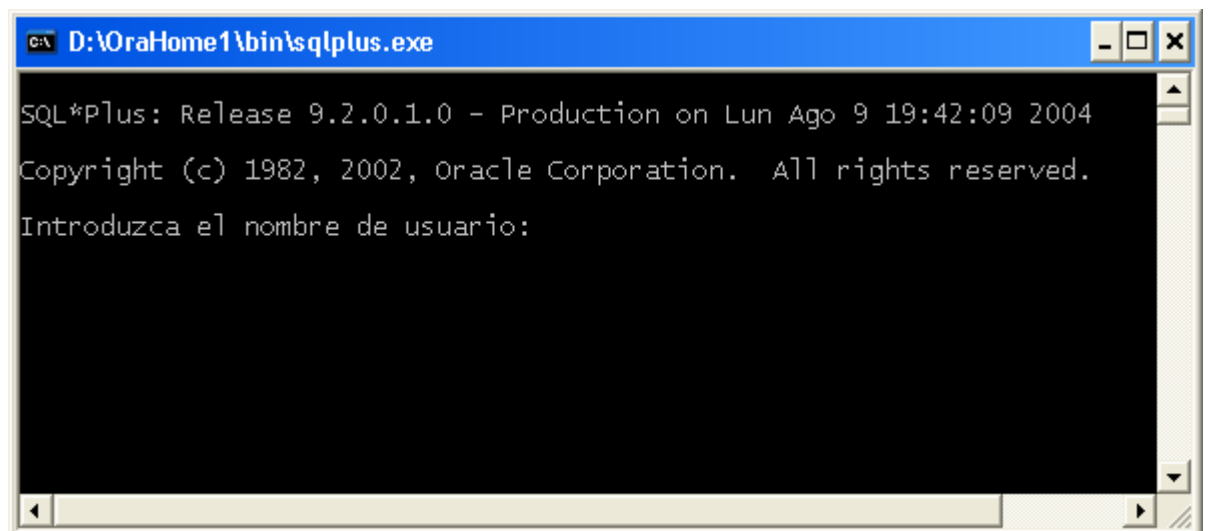
Actualmente el último estándar es el SQL del año 1999 que amplió el anterior estándar conocido como SQL 92. El SQL de Oracle es compatible con el SQL del año 1999 e incluye casi todo lo dictado por dicho estándar.

SQL*Plus

Para poder escribir sentencias SQL al servidor Oracle, éste incorpora la herramienta SQL*Plus. Toda instrucción SQL que el usuario escribe, es verificada por este programa. Si la instrucción es válida es enviada a Oracle, el cual enviará de regreso la respuesta a la instrucción; respuesta que puede ser transformada por el programa SQL*Plus para modificar su salida.

Para que el programa SQL*Plus funcione en el cliente, el ordenador cliente debe haber sido configurado para poder acceder al servidor Oracle. En cualquier caso al acceder a Oracle con este programa siempre preguntará por el nombre de usuario y contraseña. Estos son datos que tienen que nos tiene que proporcionar el administrador (DBA) de la base de datos Oracle.

Para conectar mediante SQL*Plus podemos ir a la línea de comandos y escribir el texto **sqlplus**. A continuación aparecerá la pantalla:



```
D:\OraHome1\bin\sqlplus.exe
SQL*Plus: Release 9.2.0.1.0 - Production on Lun Ago 9 19:42:09 2004
Copyright (c) 1982, 2002, Oracle Corporation. All rights reserved.
Introduzca el nombre de usuario:
```

En esa pantalla se nos pregunta el nombre de usuario y contraseña para acceder a la base de datos (información que deberá indicarnos el administrador o DBA). Tras indicar esa información conectaremos con Oracle mediante SQL*Plus, y veremos aparecer el símbolo:

```
SQL>
```

Tras el cual podremos comenzar a escribir nuestros comandos SQL. Ese símbolo puede cambiar por un símbolo con números 1, 2, 3, etc.; en ese caso se nos indica que la instrucción no ha terminado y la línea en la que estamos.

Otra posibilidad de conexión consiste en llamar al programa SQL*Plus indicando contraseña y base de datos a conectar. El formato es:

```
sqlplus usuario/contraseña@nombreServicioBaseDeDatos
```

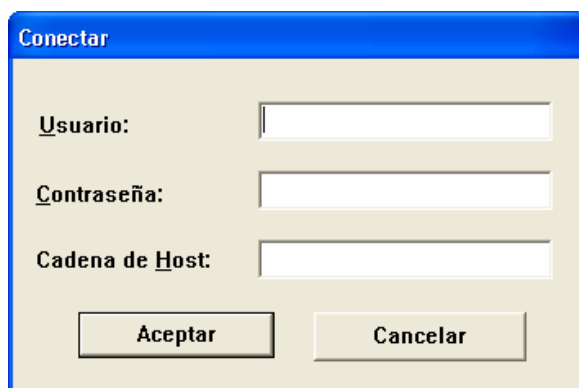
Ejemplo:

```
sqlplus usr1/miContra@inicial.forempa.net
```

En este caso conectamos con SQL*Plus indicando que somos el usuario *usr1* con contraseña *miContra* y que conectamos a la base de datos *inicial* de la red *forempa.net*. El nombre de la base de datos no tiene porque tener ese formato, habrá que conocer como es el nombre que representa a la base de datos como servicio de red en la red en la que estamos.

versión gráfica de SQL*Plus

Oracle incorpora un programa gráfico para Windows para utilizar SQL*Plus. Se puede llamar a dicho programa desde las herramientas instaladas en el menú de programas de Windows, o desde la línea de programas escribiendo **sqlplusw**. Al llamarle aparece esta pantalla:



Como en el caso anterior, se nos solicita el nombre de usuario y contraseña. La *cadena de Host* es el nombre completo de red que recibe la instancia de la base de datos a la que queremos acceder en la red en la que nos encontramos.

También podremos llamar a este entorno desde la línea de comandos utilizando la sintaxis comentada anteriormente. En este caso:

```
sqlplusw usuario/contraseña@nombreServicioBaseDeDatos
```

Esta forma de llamar al programa permite entrar directamente sin que se nos pregunte por el nombre de usuario y contraseña.

iSQL*Plus

Es un producto ideado desde la versión 9i de Oracle. Permite acceder a las bases de datos Oracle desde un navegador. Para ello necesitamos tener configurado un servidor web Oracle que permita la conexión con la base de datos. Utilizar iSQL*Plus es indicar una dirección web en un navegador, esa dirección es la de la página iSQL*Plus de acceso a la base de datos.

Desde la página de acceso se nos pedirá nombre de usuario, contraseña y nombre de la base de datos con la que conectamos (el nombre de la base de datos es el nombre con el que se la conoce en la red). Si la conexión es válida aparece esta pantalla:

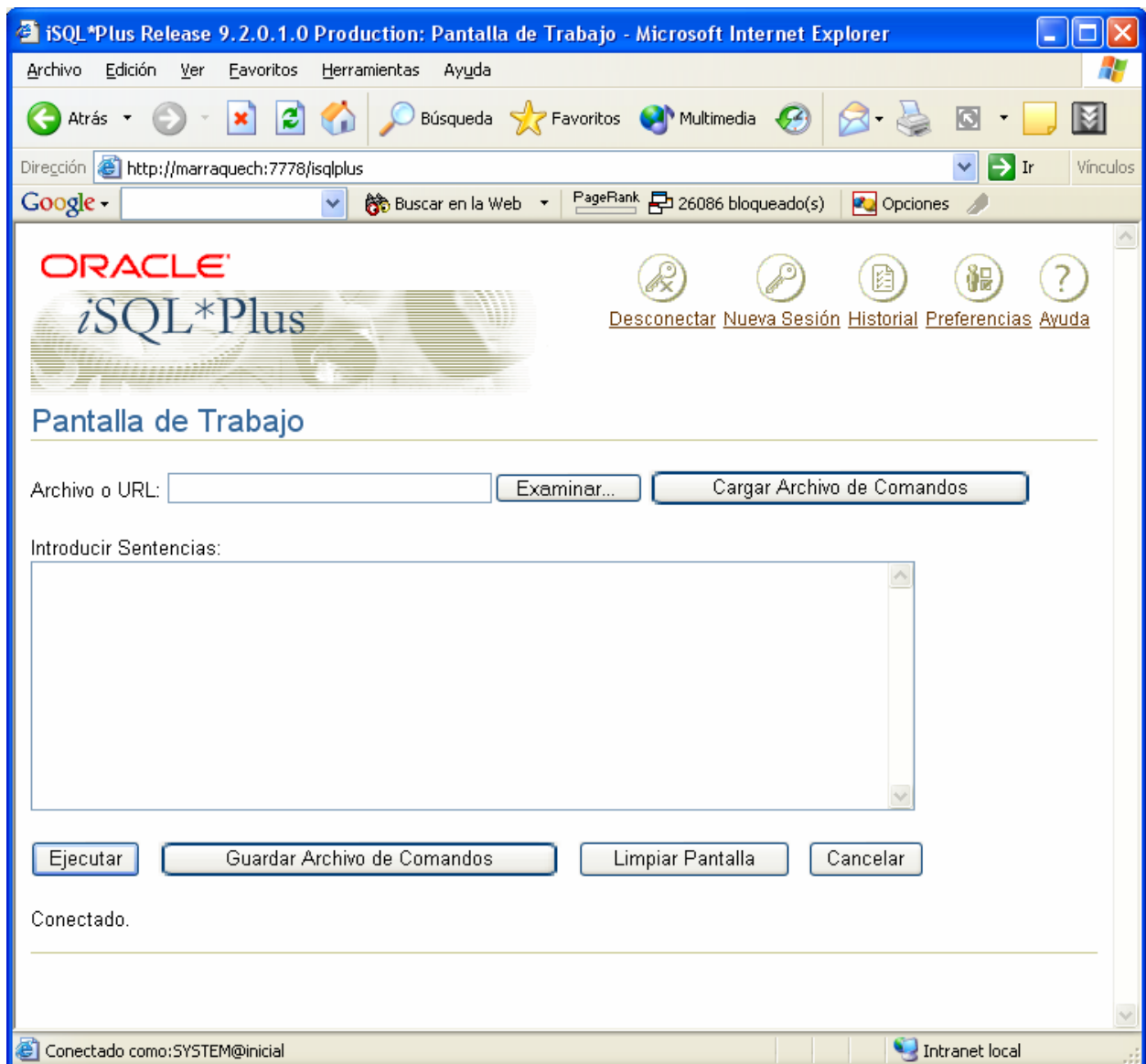


Ilustración 1, Pantalla de iSQL*Plus una vez conectados a una base de datos

En esa pantalla en el apartado **Introducir Sentencias**, se escribe la sentencia que deseamos enviar. El botón Ejecutar hace que se valide y se envíe a Oracle.

Se pueden almacenar sentencias SQL usando el botón **Examinar** y cargar sentencias previamente guardadas mediante **Cargar archivos de comandos**.

estructura del lenguaje SQL

- **SELECT**. Se trata del comando que permite realizar consultas sobre los datos de la base de datos. Obtiene datos de la base de datos.
- **DML, *Data Manipulation Language* (Lenguaje de manipulación de datos)**. Modifica filas (registros) de la base de datos. Lo forman las instrucciones **INSERT, UPDATE, MERGE y DELETE**.
- **DDL, *Data Definition Language* (Lenguaje de definición de datos)**. Permiten modificar la estructura de las tablas de la base de datos. Lo forman las instrucciones **CREATE, ALTER, DROP, RENAME y TRUNCATE**.
- **Instrucciones de transferencia**. Administran las modificaciones creadas por las instrucciones DML. Lo forman las instrucciones **ROLLBACK, COMMIT y SAVEPOINT**
- **DCL, *Data Control Language* (Lenguaje de control de datos)**. Administran los derechos y restricciones de los usuarios. Lo forman las instrucciones **GRANT y REVOKE**.

normas de escritura

- En SQL no se distingue entre mayúsculas y minúsculas. Da lo mismo como se escriba.
- El final de una instrucción lo calibra el signo del punto y coma
- Los comandos SQL (SELECT, INSERT,...) no pueden ser partidos por espacios o saltos de línea antes de finalizar la instrucción. El intérprete SQL plus indica
- Se pueden tabular líneas para facilitar la lectura si fuera necesario
- Los comentarios en el código SQL comienzan por /* y terminan por */

tablas

esquemas de usuario y objetos

Cada usuario de una base de datos posee un **esquema**. El esquema tiene el mismo nombre que el usuario y sirve para almacenar los objetos de esquema, es decir los objetos que posee el usuario.

Esos objetos pueden ser: tablas, vistas, secuencias, índices, sinónimos e instantáneas. Esos objetos son manipulados y creados por los usuarios. En principio sólo los administradores y los usuarios propietarios pueden acceder a cada objeto, salvo que se modifiquen los privilegios del objeto para permitir su acceso por parte de otros usuarios.

creación de tablas

nombre de las tablas

Deben cumplir las siguientes reglas:

- ⦿ Deben comenzar con una letra
- ⦿ No deben tener más de 30 caracteres
- ⦿ Sólo se permiten utilizar letras del alfabeto (inglés), números o el signo de subrayado (también el signo \$ y #, pero esos se utilizan de manera especial por lo que no son recomendados)
- ⦿ No puede haber dos tablas con el mismo nombre para el mismo usuario (pueden coincidir los nombres si están en distintos esquemas)
- ⦿ No puede coincidir con el nombre de una palabra reservada de Word

orden CREATE TABLE

Es la orden SQL que permite crear una tabla. Por defecto será almacenada en el tablespace por defecto del usuario que crea la tabla. Sintaxis:

```
CREATE TABLE [esquema.] nombreDeTabla  
(nombreDeLaColumna1 tipoDeDatos [, ...]);
```

Ejemplo:

```
CREATE TABLE proveedores (nombre varchar2(25));
```

Crea una tabla con un solo campo de tipo **varchar2**.

Sólo se podrá crear la tabla si el usuario posee los permisos necesarios para ello. Si la tabla pertenece a otro esquema (suponiendo que el usuario tenga permiso para grabar tablas en ese otro esquema), se antepone al nombre de la tabla, el nombre del esquema:

```
CREATE TABLE otroUsuario.proveedores (nombre varchar2(25));
```

orden DESCRIBE

El comando DESCRIBE, permite obtener la estructura de una tabla. Ejemplo:

```
DESCRIBE proveedores;
```

Y aparecerán los campos de la tabla proveedores.

orden INSERT

Permite añadir datos a las tablas. Más adelante se comenta de forma más detallada. Su sintaxis básica es:

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

Indicando la tabla se añaden los datos que se especifiquen tras el apartado **values** en un nuevo registro. Los valores deben corresponderse con el orden de las columnas. Si no es así se puede indicar tras el nombre de la tabla y entre paréntesis. Ejemplo:

```
INSERT INTO proveedores(nombre, CIF)
VALUES ('Araja SA', '14244223Y');
```

consultar las tablas del usuario

En el diccionario de datos hay una entrada que permite consultar las tablas de cada usuario. Esa entrada es **USER_TABLES**. De forma que `SELECT * FROM USER_TABLES` obtiene una vista de las tablas del usuario actual. Hay diversas columnas que muestran datos sobre cada tabla, entre ellas la columna **TABLES_NAME** muestra el nombre de la tabla.

borrar tablas

La orden **DROP TABLE** seguida del nombre de una tabla, permite eliminar la tabla en cuestión.

Al borrar una tabla:

- ⦿ Desaparecen todos los datos
- ⦿ Cualquier vista y sinónimo referente a la tabla seguirán existiendo, pero ya no funcionarán (conviene eliminarlos)
- ⦿ Las transacciones pendientes son aceptadas (COMMIT)
- ⦿ Sólo es posible realizar esta operación si se es el propietario de la tabla o se posee el privilegio DROP ANY TABLE

El borrado de una tabla es irreversible, y no hay ninguna petición de confirmación, por lo que conviene ser muy cuidadoso con esta operación.

tipos de datos

equivalentes ANSI SQL con el SQL de Oracle

Hay diferencias entre los tipos de datos del estándar ANSI con respecto al SQL de Oracle. Aunque Oracle es capaz de utilizar bases de datos con formato ANSI y tipos compatibles con el mismo, la equivalencia ANSI / Oracle la dicta esta tabla:

Tipos ANSI SQL	Equivalente Oracle SQL
CHARACTER (<i>n</i>) CHAR (<i>n</i>)	CHAR (<i>n</i>)
CHARACTER VARYING (<i>n</i>) CHAR VARYING (<i>n</i>)	VARCHAR (<i>n</i>)
NATIONAL CHARACTER (<i>n</i>) NATIONAL CHAR (<i>n</i>) NCHAR (<i>n</i>)	NCHAR (<i>n</i>)
NATIONAL CHARACTER VARYING (<i>n</i>) NATIONAL CHAR VARYING (<i>n</i>) NCHAR VARYING (<i>n</i>)	NVARCHAR2 (<i>n</i>)
NUMERIC (<i>p,s</i>) DECIMAL (<i>p,s</i>)	NUMBER (<i>p,s</i>)
INTEGER INT SMALLINT	NUMBER (38)
FLOAT (<i>b</i>) DOUBLE DOUBLE PRECISION REAL	NUMBER
LONG VARCHAR (<i>n</i>)	LONG

textos

Para los textos disponemos de los siguientes tipos:

- ⊙ **VARCHAR2**. Para textos de longitud variable de hasta 4000 caracteres
- ⊙ **CHAR**. Para textos de longitud fija de hasta 2000 caracteres.
- ⊙ **NCHAR**. Para el almacenamiento de caracteres nacionales de texto fijo
- ⊙ **NVARCHAR2**. Para el almacenamiento de caracteres nacionales de longitud variable.

En todos estos tipos se indican los tamaños entre paréntesis tras el nombre del tipo. Ese tamaño en el caso de los tipos VARCHAR2 es obligatorio, en el caso de los tipos CHAR son opcionales (de no ponerlos se toma el uno).

Conviene poner suficiente espacio para almacenar los valores. En el caso de los VARCHAR, Oracle no malgasta espacio por poner más espacio del deseado ya que si el texto es más pequeño que el tamaño indicado, el resto del espacio se ocupa.

números

El tipo NUMBER es un formato versátil que permite representar todo tipo de números. Su rango recoge números de entre 10^{-130} y $9,99999999999 * 10^{128}$. Fuera de estos rangos Oracle devuelve un error.

Los números decimales (números de coma fija) se indican con **NUMBER(p,s)**, donde *p* es la precisión máxima y *s* es la escala (número de decimales a la derecha de la coma). Por ejemplo, NUMBER (8,3) indica que se representan números de ocho cifras de precisión y tres decimales. Los decimales en Oracle se presenta con el **punto y no con la coma**.

Para números enteros se indica **NUMBER(p)** donde *p* es el número de dígitos. Eso es equivalente a NUMBER(*p*,0).

Para números de coma flotante (equivalentes a los **flota** o **double** de muchos lenguajes de programación) simplemente se indica el texto **NUMBER** sin precisión ni escala.

precisión y escala

La cuestión de la precisión y la escala es compleja. Para entenderla mejor, se muestran estos ejemplos:

Formato	Número escrito por el usuario	Se almacena como...
NUMBER	345255.345	345255.345
NUMBER(9)	345255.345	345255
NUMBER(9,2)	345255.345	345255.36
NUMBER(7)	345255.345	Da error de precisión
NUMBER(7,-2)	345255.345	345300
NUMBER(7,2)	345255.345	Da error de precisión

En definitiva, la precisión debe incluir todos los dígitos del número (puede llegar hasta 38 dígitos). La escala sólo indica los decimales que se respetarán del número, pero si es negativa indica ceros a la izquierda del decimal.

tipo LONG

Se trata de un tipo de datos que actualmente se mantiene por compatibilidad. Se recomienda encarecidamente utilizar en su lugar el tipo CLOB (que se comentará más adelante). En cualquier caso este tipo permite almacenar textos de hasta 2 GB de tamaño. Pero no puede formar clave, ni índice, ni ser parte de la cláusula WHERE, ni GROUP BY, ni SELECT con DISTINCT, ni pueden ser UNIQUE y sólo puede haber un campo de este tipo en una misma tabla entre otras limitaciones.

fechas y horas

DATE

El tipo **DATE** permite almacenar fechas. Las fechas se pueden escribir en formato día, mes y año entre comillas. El separador puede ser una barra de dividir, un guión y casi cualquier símbolo.

Para almacenar la fecha actual basta con utilizar la función **SYSDATE** que devuelve esa fecha.

TIMESTAMP

Es una extensión del anterior, almacena valores de día, mes y año, junto con hora, minuto y segundos (incluso con decimales). Con lo que representa un instante concreto en el tiempo. Un ejemplo de **TIMESTAMP** sería '2/2/2004 18:34:23,34521'. En este caso si el formato de fecha y hora del sistema está pensado para el idioma español, el separador decimal será la coma (y no el punto).

intervalos

Hay unos cuantos tipos de datos en Oracle que sirven para almacenar intervalos de tiempo (no fechas, sino una suma de elementos de tiempo).

INTERVAL YEAR TO MONTH

Este tipo de datos almacena años y meses. Tras la palabra **YEAR** se puede indicar la precisión de los años (cifras del año), por defecto es de dos. Ejemplo:

```
CREATE TABLE tiempos (meses INTERVAL YEAR(3) TO MONTH);  
INSERT INTO tiempos VALUES('3-2');
```

En el ejemplo se inserta un registro que representa 3 años y dos meses.

INTERVAL DAY TO SECOND

Representa intervalos de tiempo que expresan días, horas, minutos y segundos. Se puede indicar la precisión tras el texto **DAY** y el número de decimales de los segundos tras el texto **SECOND**. Ejemplo:

```
CREATE TABLE tiempos (días INTERVAL DAY(3) TO SECOND(0));  
INSERT INTO tiempos VALUES('2 7:12:23');
```

RAW

Sirve para almacenar valores binarios de hasta 2000 bytes (se puede especificar el tamaño máximo entre paréntesis). El valor **LONG RAW** almacena hasta 2GB.

LOB

Son varios tipos de datos que permiten almacenar valores muy grandes. Más adelante se comentan en su totalidad. Incluye a **BLOB**, **CLOB**, **NCLOB** y **BFILE**.

ROWID

Valor hexadecimal que representa la dirección única de una fila en su tabla.

modificar tablas

cambiar de nombre

La orden **RENAME** permite el cambio de nombre de cualquier objeto. Sintaxis:

```
RENAME nombreViejo TO nombreNuevo
```

borrar contenido de tablas

La orden **TRUNCATE TABLE** seguida del nombre de una tabla, hace que se elimine el contenido de la tabla, pero no la tabla en sí. Incluso borra del archivo de datos el espacio ocupado por la tabla.

Esta orden no puede anularse con un **ROLLBACK**.

modificar tablas

La versátil **ALTER TABLE** permite hacer cambios en la estructura de una tabla.

añadir columnas

```
ALTER TABLE nombreTabla ADD(nombreColumna TipoDatos  
[Propiedades]  
[,columnaSiguiete tipoDatos [propiedades]...)
```

Permite añadir nuevas columnas a la tabla. Se deben indicar su tipo de datos y sus propiedades si es necesario (al estilo de **CREATE TABLE**).

Las nuevas columnas se añaden al final, no se puede indicar otra posición.

borrar columnas

```
ALTER TABLE nombreTabla DROP(columna);
```

Elimina la columna indicada de manera irreversible e incluyendo los datos que contenía. No se puede eliminar la última columna (habrá que usar **DROP TABLE**).

modificar columna

Permite cambiar el tipo de datos y propiedades de una determinada columna. Sintaxis:

```
ALTER TABLE nombreTabla MODIFY(columna tipo [propiedades]  
[columnaSiguiete tipo [propiedades] ...])
```

Los cambios que se permiten son:

- ⊙ Incrementar precisión o anchura de los tipos de datos
- ⊙ Sólo se puede reducir la anchura si la anchura máxima de un campo si esa columna posee nulos en todos los registros, o todos los valores so o no hay registros

- ⊙ Se puede pasar de CHAR a VARCHAR2 y viceversa (si no se modifica la anchura)
- ⊙ Se puede pasar de DATE a TIMESTAMP y viceversa

añadir comentarios a las tablas

Se le pueden poner comentarios a las tablas y las columnas. Un comentario es un texto descriptivo utilizado para documentar la tabla. Sintaxis:

```
COMMENT ON { TABLE NombreTabla | COLUMN tabla.nombreColumna }  
IS 'Comentario'
```

Para mostrar los comentarios puestos se usan las siguientes vistas del diccionario de datos mediante la instrucción SELECT:

- ⊙ **USER_TAB_COMMENTS.** Comentarios de las tablas del usuario actual.
- ⊙ **USER_COL_COMMENTS.** Comentarios de las columnas del usuario actual.
- ⊙ **ALL_TAB_COMMENTS.** Comentarios de las tablas de todos los usuarios (sólo administradores)
- ⊙ **ALL_COL_COMMENTS.** Comentarios de las columnas de todos los usuarios (sólo administradores).

valor por defecto

A cada columna se le puede asignar un valor por defecto durante su creación mediante la propiedad DEFAULT. Se puede poner esta propiedad durante la creación o modificación de la tabla, añadiendo la palabra DEFAULT tras el tipo de datos del campo y colocando detrás el valor que se desea por defecto.

Ejemplo:

```
CREATE TABLE articulo (cod NUMBER(7), nombre VARCHAR2(25),  
precio NUMBER(11,2) DEFAULT 3.5);
```

restricciones

Una restricción es una condición de obligado cumplimiento para una o más columnas de la tabla. A cada restricción se le pone un nombre, en el caso de no poner un nombre (en las que eso sea posible) entonces el propio Oracle le coloca el nombre que es un mnemotécnico con el nombre de tabla, columna y tipo de restricción.

Su sintaxis general es:

```
{CREATE TABLE nombreTabla |  
ALTER TABLE nombreTabla {ADD | MODIFY}}  
(campo tipo [propiedades] [...]  
CONSTRAINT nombreRestricción tipoRestricción (columnas)  
[,CONSTRAINT nombrerestricción tipoRestricción (columnas) ...])
```

Las restricciones tienen un nombre, se puede hacer que sea Oracle el que les ponga nombre, pero entonces será críptico. Por eso es mejor ponerle uno mismo.

Los nombres de restricción no se pueden repetir para el mismo esquema, por lo que es buena idea incluir de algún modo el nombre de la tabla, los campos involucrados y el tipo de restricción en el nombre de la misma. Por ejemplo *pieza_id_pk* podría indicar que el campo *id* de la tabla *pieza* tiene una clave principal (**PRIMARY KEY**).

prohibir nulos

La restricción NOT NULL permite prohibir los nulos en una determinada tabla. Eso obliga a que la columna tenga que tener obligatoriamente un valor para que sea almacenado el registro.

Se puede colocar durante la creación (o modificación) del campo añadiendo la palabra NOT NULL tras el tipo:

```
CREATE TABLE cliente(dni VARCHAR2(9) NOT NULL);
```

En ese caso el nombre le coloca Oracle. La otra forma (que admite nombre) es:

```
CREATE TABLE cliente(dni VARCHAR2(9)
    CONSTRAINT dni_sinnulos NOT NULL(dni));
```

valores únicos

Las restricciones de tipo UNIQUE obligan a que el contenido de uno o más campos no puedan repetir valores. Nuevamente hay dos formas de colocar esta restricción:

```
CREATE TABLE cliente(dni VARCHAR2(9) UNIQUE);
```

En ese caso el nombre de la restricción la coloca el sistema Oracle. Otra forma es:

```
CREATE TABLE cliente(dni VARCHAR2(9) CONSTRAINT dni_u UNIQUE);
```

Esta forma permite poner un nombre a la restricción. Si la repetición de valores se refiere a varios campos, la forma sería:

```
CREATE TABLE alquiler(dni VARCHAR2(9),
    cod_pelicula NUMBER(5),
    CONSTRAINT alquiler_uk UNIQUE(dni,cod_pelicula) ;
```

La coma tras la definición del campo *cod_pelicula* hace que la restricción sea independiente de ese campo. Eso obliga a que, tras UNIQUE se indique la lista de campos.

Los campos UNIQUE son las claves candidatas de la tabla (que habrán sido detectadas en la fase de diseño de la base de datos).

clave primaria

La clave primaria de una tabla la forman las columnas que indican a cada registro de la misma. La clave primaria hace que los campos que la forman sean NOT NULL (sin posibilidad de quedar vacíos) y que los valores de los campos sean de tipo UNIQUE (sin posibilidad de repetición).

Si la clave está formada por un solo campo basta con:

```
CREATE TABLE cliente(
  dni VARCHAR2(9) PRIMARY KEY,
  nombre VARCHAR(50)) ;
```

O, poniendo un nombre a la restricción:

```
CREATE TABLE cliente(
  dni VARCHAR2(9) CONSTRAINT cliente_pk PRIMARY KEY,
  nombre VARCHAR(50)) ;
```

Si la clave la forman más de un campo:

```
CREATE TABLE alquiler(dni VARCHAR2(9),
  cod_pelicula NUMBER(5),
  CONSTRAINT alquiler_pk PRIMARY KEY(dni,cod_pelicula) ;
```

clave secundaria o foránea

Una clave secundaria o foránea, es uno o más campos de una tabla que están relacionados con la clave principal de los campos de otra tabla. La forma de indicar una clave foránea es:

```
CREATE TABLE alquiler(dni VARCHAR2(9),
  cod_pelicula NUMBER(5),
  CONSTRAINT alquiler_pk PRIMARY KEY(dni,cod_pelicula),
  CONSTRAINT dni_fk FOREIGN KEY (dni)
    REFERENCES clientes(dni),
  CONSTRAINT pelicula_fk FOREIGN KEY (cod_pelicula)
    REFERENCES peliculas(cod)
);
```

Esta completa forma de crear la tabla alquiler incluye sus claves foráneas, el campo *dni* hace referencia al campo *dni* de la tabla *clientes* y el campo *cod_pelicula* que hace referencia al campo *cod* de la tabla *peliculas*. También hubiera bastado con indicar sólo la tabla a la que hacemos referencia, si no se indican los campos relacionados de esa tabla, se toma su clave principal (que es lo normal).

Esto forma una relación entre dichas tablas, que además obliga al cumplimiento de la **integridad referencial**. Esta integridad obliga a que cualquier *dni* incluido en la tabla *alquiler* tenga que estar obligatoriamente en la tabla de *clientes*. De no ser así el registro no será insertado en la tabla (ocurrirá un error).

Otra forma de crear claves foráneas (sólo válida para claves de un solo campo) es:

```
CREATE TABLE alquiler(
  dni VARCHAR2(9) CONSTRAINT dni_fk
    REFERENCES clientes(dni),
  cod_pelicula NUMBER(5) CONSTRAINT pelicula_fk
    REFERENCES peliculas(cod)
  CONSTRAINT alquiler_pk PRIMARY KEY(dni,cod_pelicu));
```

Esta definición de clave secundario es idéntica a la anterior, sólo que no hace falta colocar el texto FOREIGN KEY.

La integridad referencial es una herramienta imprescindible de las bases de datos relacionales. Pero provoca varios problemas. Por ejemplo, si borramos un registro en la tabla principal que está relacionado con uno o varios de la secundaria ocurrirá un error, ya que de permitírse nos borrar el registro ocurrirá fallo de integridad (habrá claves secundarios refiriéndose a una clave principal que ya no existe).

Por ello Oracle nos ofrece dos soluciones a añadir tras la cláusula REFERENCES:

- ⊙ **ON DELETE SET NULL.** Coloca nulos todas las claves secundarias relacionadas con la borrada.
- ⊙ **ON DELETE CASCADE.** Borra todos los registros cuya clave secundaria es igual que la clave del registro borrado.

Si no se indica esta cláusula, no se permite el borrado de registros relacionados.

El otro problema ocurre si se desea cambiar el valor de la clave principal en un registro relacionado con claves secundarias. En muchas bases de datos se implementan soluciones consistentes en añadir ON UPDATE CASCADE o ON UPDATE SET NULL. Oracle no implementa directamente estas soluciones. Por lo que hay que hacerlo de otra forma. Las soluciones son:

- ⊙ Implementar un TRIGGER para que cuando se actualice el registro se actualicen las claves secundarias (el mecanismo de funcionamiento es parecido al que se muestra en el siguiente párrafo).
- ⊙ Añadir un registro igual que el que se quiere cambiar en la tabla principal, pero con el nuevo valor de la clave. Mediante una instrucción UPDATE actualizar a ese valor de clave todos los registros de la tabla secundaria cuyo valor coincida con la antigua clave. Finalmente borrar el registro en la tabla principal con el valor antiguo de la clave.

La sintaxis completa para añadir claves foráneas es:

```
CREATE TABLE tabla(lista_de_campos
    CONSTRAINT nombreRestriccion FOREIGN KEY (listaCampos)
    REFERENCES tabla(clavePrincipalRelacionada)
    [ON UPDATE {SET NULL | CASCADE}]
);
```

Si es de un solo campo existe esta alternativa:

```
CREATE TABLE tabla(lista_de_campos tipos propiedades,
    nombreCampoClaveSecundaria
    CONSTRAINT nombreRestriccion
    REFERENCES tabla(clavePrincipalRelacionada)
    [ON UPDATE {SET NULL | CASCADE}]
);
```

restricciones de validación

Son restricciones que dictan una condición que deben cumplir los contenidos de una columna. La expresión de la condición es cualquier expresión que devuelva verdadero o falso, pero si cumple estas premisas:

- ⦿ No puede hacer referencia a números de fila
- ⦿ No puede hacer referencia a objetos de SYSTEM o SYS
- ⦿ No se permiten usar las funciones SYSDATE, UID, USER y USERENV
- ⦿ No se permiten referencias a columnas de otras tablas (si a las de la misma tabla)

Una misma columna puede tener múltiples CHECKS en su definición (se pondrían varios CONSTRAINT seguidos, sin comas). Ejemplo:

```
CREATE TABLE ingresos(cod NUMBER(5) PRIMARY KEY,
    concepto VARCHAR2(40) NOT NULL,
    importe NUMBER(11,2) CONSTRAINT importe_min
        CHECK (importe>0)
    CONSTRAINT importe_max
        CHECK (importe<8000)
);
```

Para poder hacer referencia a otras columnas hay que construir la restricción de forma independiente a la columna:

```
CREATE TABLE ingresos(cod NUMBER(5) PRIMARY KEY,
    concepto VARCHAR2(40) NOT NULL,
    importe_max NUMBER(11,2),
    importe NUMBER(11,2),
    CONSTRAINT importe_maximo
        CHECK (importe<importe_max)
);
```

añadir restricciones

Es posible querer añadir restricciones tras haber creado la tabla. En ese caso se utiliza la siguiente sintaxis:

```
ALTER TABLE tabla
ADD [CONSTRAINT nombre] tipoDeRestricción(columnas);
```

tipoRestricción es el texto CHECK, PRIMARY KEY o FOREIGN KEY. Las restricciones NOT NULL deben indicarse mediante ALTER TABLE .. MODIFY colocando NOT NULL en el campo que se modifica.

borrar restricciones

Sintaxis:

```
ALTER TABLE tabla  
DROP PRIMARY KEY | UNIQUE(campos) |  
CONSTRAINT nombreRestricción [CASCADE]
```

La opción PRIMARY KEY elimina una clave principal (también quitará el índice UNIQUE sobre las campos que formaban la clave. UNIQUE elimina índices únicos. La opción CONSTRAINT elimina la restricción indicada.

La opción CASCADE hace que se eliminen en cascada las restricciones de integridad que dependen de la restricción eliminada. Por ejemplo en:

```
CREATE TABLE curso(  
    cod_curso CHAR(7) PRIMARY KEY,  
    fecha_inicio DATE,  
    fecha_fin DATE,  
    tItulo VARCHAR2(60),  
    cod_siguientecurso CHAR(7),  
    CONSTRAINT fecha_ck CHECK(fecha_fin>fecha_inicio),  
    CONSTRAINT cod_ste_fk FOREIGN KEY(cod_siguientecurso)  
        REFERENCES curso ON DELETE SET NULL);
```

Tras esa definición de tabla, esta instrucción:

```
ALTER TABLE curso DROP PRIMARY KEY;
```

Produce este error:

```
ORA-02273: a esta clave única/primaria hacen referencia  
algunas claves ajenas
```

Para ello habría que utilizar esta instrucción:

```
ALTER TABLE curso DROP PRIMARY KEY CASCADE;
```

Esa instrucción elimina la clave secundaria antes de eliminar la principal.

También produce error esta instrucción:

```
ALTER TABLE curso DROP(fecha_inicio);  
ERROR en línea 1:  
ORA-12991: se hace referencia a la columna en una restricción  
de multicolumna
```


El error se debe a que no es posible borrar una columna que forma parte de la definición de una instrucción. La solución es utilizar **CASCADE CONSTRAINT** elimina las restricciones en las que la columna a borrar estaba implicada:

```
ALTER TABLE curso DROP(fecha_inicio) CASCADE CONSTRAINTS;
```

Esta instrucción elimina la restricción de tipo **CHECK** en la que aparecía la *fecha_inicio* y así se puede eliminar la columna.

desactivar restricciones

A veces conviene temporalmente desactivar una restricción para saltarse las reglas que impone. La sintaxis es:

```
ALTER TABLE tabla DISABLE CONSTRAINT nombre [CASCADE]
```

La opción **CASCADE** hace que se desactiven también las restricciones dependientes de la que se desactivó.

activar restricciones

Anula la desactivación. Formato:

```
ALTER TABLE tabla ENABLE CONSTRAINT nombre [CASCADE]
```

Sólo se permite volver a activar si los valores de la tabla cumplen la restricción que se activa. Si hubo desactivado en cascada, habrá que activar cada restricción individualmente.

cambiar de nombre a las restricciones

Para hacerlo se utiliza este comando:

```
ALTER TABLE table RENAME CONSTRAINT
nombreViejo TO nombreNuevo;
```

mostrar restricciones

La vista del diccionario de datos **USER_CONSTRAINTS** permite identificar las restricciones colocadas por el usuario (**ALL_CONSTRAINTS** permite mostrar las restricciones de todos los usuarios, pero sólo está permitida a los administradores). En esa vista aparece toda la información que el diccionario de datos posee sobre las restricciones. En ella tenemos las siguientes columnas interesantes:

Columna	Tipo de datos	Descripción
OWNER	VARCHAR2(20)	Indica el nombre del usuario propietario de la tabla
CONSTRAINT_NAME	VARCHAR2(30)	Nombre de la restricción

Columna	Tipo de datos	Descripción
CONSTRAINT_TYPE	VARCHAR2(1)	Tipo de restricción: <input type="radio"/> C. De tipo CHECK o NOT NULL <input type="radio"/> P. PRIMARY KEY <input type="radio"/> R. FOREIGN KEY <input type="radio"/> U. UNIQUE
TABLE_NAME	VARCHAR2(30)	Nombre de la tabla en la que se encuentra la restricción

En el diccionario de datos hay otra vista que proporciona información sobre restricciones, se trata de **USER_CONS_COLUMNS**, en dicha tabla se muestra información sobre las columnas que participan en una restricción. Así si hemos definido una clave primaria formada por los campos *uno* y *dos*, en la tabla **USER_CONS_COLUMNS** aparecerán dos entradas, una para el primer campo del índice y otra para el segundo. Se indicará además el orden de aparición en la restricción. Ejemplo:

OWNER	CONSTRAINT_NAME	TABLE_NAME	COLUMN_NAME	POSITION
JORGE	EXIS_PK	EXISTENCIAS	TIPO	1
JORGE	EXIS_PK	EXISTENCIAS	MODELO	2
JORGE	EXIS_PK	EXISTENCIAS	N_ALMACEN	3
JORGE	PIEZA_FK	EXISTENCIAS	TIPO	1
JORGE	PIEZA_FK	EXISTENCIAS	MODELO	2
JORGE	PIEZA_PK	PIEZA	TIPO	1
JORGE	PIEZA_PK	PIEZA	MODELO	2

En esta tabla **USER_CONS_COLUMNS** aparece una restricción de clave primaria sobre la tabla *existencias*, esta clave está formada por las columnas (*tipo*, *modelo* y *n_almacen*) y en ese orden. Una segunda restricción llamada *pieza_fk* está compuesta por *tipo* y *modelo* de la tabla *existencias*. Finalmente la restricción *pieza_pk* está formada por *tipo* y *modelo*, columnas de la tabla *pieza*.

Para saber de qué tipo son esas restricciones, habría que acudir a la vista **USER_CONSTRAINTS**.

consultas SELECT

capacidades

Sin duda el comando más versátil del lenguaje SQL es el comando SELECT. Este comando permite:

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

sintaxis sencilla

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

Donde:

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

Ejemplos:

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

cálculos

aritméticos

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

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

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

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

Los nombres pueden llevar espacios si se ponen con comillas dobles:

```
SELECT nombre, precio, precio*1.16 AS "precio con iva"  
FROM articulos;
```

Esas comillas dobles cumplen otra función y es la de hacer que se respeten las mayúsculas y minúsculas del nombre (de otro modo el nombre de la columna aparece siempre en mayúsculas)

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

Cuando una expresión aritmética se calcula sobre valores NULL, el resultado de la expresión es siempre NULL.

concatenación

El operador || es el de la concatenación. Sirve para unir textos. Ejemplo:

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

El resultado puede ser:

TIPO	MODELO	Clave Pieza
AR	6	AR-6
AR	7	AR-7
AR	8	AR-8
AR	9	AR-9
AR	12	AR-12
AR	15	AR-15
AR	20	AR-20
AR	21	AR-21
BI	10	BI-10
BI	20	BI-20
BI	38	BI-38
BI	57	BI-57

condiciones

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

Ejemplo:

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

operadores de comparación

Se pueden utilizar en la cláusula WHERE, son:

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

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

Así la letra Ñ y las vocales acentuadas nunca quedan bien ordenadas ya que figuran con códigos más altos. Las mayúsculas figuran antes que las minúsculas (la letra 'Z' es menor que la 'a').

valores lógicos

Son:

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

Ejemplo:

```
/* Obtiene a las personas de entre 25 y 50 años
SELECT nombre,apellidos FROM personas
WHERE edad>=25 AND edad<=50;
/*Obtiene a la gente de más de 60 años o de menos de 20
SELECT nombre,apellidos FROM personas
WHERE edad>60 OR edad<20;
```

BETWEEN

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

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

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

IN

Permite obtener registros cuyos valores estén en una lista:

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

Obtiene piezas cuyos precios sea 3, 5 u 8, sólo uno de esos tres.

LIKE

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

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

Ejemplos:

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

IS NULL

Devuelve verdadero si una expresión contiene un nulo:

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

Esa instrucción selecciona a la gente que no tiene teléfono

Precedencia de operadores

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

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

ordenación

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

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

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

Sintaxis completa de SELECT:

```
SELECT expresiones
FROM tabla
[WHERE condición]
ORDER BY listaDeCamposOAlias;
```

funciones

Oracle incorpora una serie de instrucciones que permiten realizar cálculos avanzados, o bien facilitar la escritura de ciertas expresiones. Todas las funciones reciben datos para poder operar (parámetros) y devuelven un resultado (que depende de los parámetros enviados a la función. Los argumentos se pasan entre paréntesis:

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

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

Las hay de dos tipos:

- ⊙ Funciones que operan con una sola fila
- ⊙ Funciones que operan con varias filas.

Sólo veremos las primeras (más adelante se comentan las de varias filas).

funciones de caracteres

conversión del texto a mayúsculas y minúsculas

Son:

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

funciones de transformación

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

Función	Descripción
<i>caracterDeRelleno)</i>	el texto se recorta.

funciones numéricas

redondeos

Función	Descripción
ROUND (<i>n,decimales</i>)	Redondea el número al siguiente número con el número de decimales indicado más cercano. ROUND(8.239,2) devuelve 8.3
TRUNC (<i>n,decimales</i>)	Los decimales del número para que sólo aparezca el número de decimales indicado
FLOOR (<i>n</i>)	Obtiene el entero más grande o igual que <i>n</i>
CEIL (<i>n</i>)	Entero más pequeño o igual que <i>n</i>

matemáticas

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

otras

Función	Descripción
BITAND (<i>n1,n2</i>)	Realiza una operación AND de bits sobre los valores <i>n1</i> y <i>n2</i> que tienen que ser enteros sin

Función	Descripción
	signo (el resultado también es un entero)
VSIZE (<i>valor</i>)	Tamaño en bytes que gasta Oracle en almacenar ese valor

funciones de trabajo con nulos

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

Función	Descripción
NVL (<i>valor,sustituto</i>)	Si el <i>valor</i> es NULL, devuelve el valor <i>sustituto</i> ; de otro modo, devuelve valor
NVL2 (<i>valor,sustituto1,sustituto2</i>)	Variante de la anterior, devuelve el valor <i>sustituto1</i> si <i>valor</i> no es nulo. Si <i>valor</i> es nulo devuelve el <i>sustituto2</i>
NULLIF (<i>valor1,valor2</i>)	Función que devuelve nulo si el <i>valor1</i> y el <i>valor2</i> sean iguales. En el caso de que no lo sean devuelve el <i>valor1</i>
COALESCE (<i>valor1,valor2</i> [, <i>valor3...</i>])	Devuelve el <i>valor1</i> si no es nulo; si lo es devuelve el <i>valor2</i> si no es nulo. Si son nulos el 1 y el 2 devuelve el tres si este no es nulo, y así sucesivamente

funciones de fecha

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

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

intervalos

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

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

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

La precisión en el caso de indicar tanto años como meses, se indica sólo en el año.

En intervalos de días a segundos los intervalos se pueden indicar como:

```

/* 4 días 10 horas 12 minutos y 7 con 352 segundos */
INTERVAL '4 10:12:7,352' DAY TO SECOND(3)
/* 4 días 10 horas 12 minutos */
INTERVAL '4 10:12' DAY TO MINUTE
/* 4 días 10 horas */
INTERVAL '4 10' DAY TO HOUR
/* 4 días*/
INTERVAL '4' DAY
/*10 horas*/
INTERVAL '10' HOUR
/*25 horas*/
INTERVAL '253' HOUR
/*12 minutos*/
INTERVAL '12' MINUTE
/*30 segundos */
INTERVAL '30' SECOND
/*8 horas y 50 minutos */
INTERVAL ('8:50') HOUR TO MINUTE;
/*7 minutos 6 segundos*/
INTERVAL ('7:06') MINUTE TO SECOND;
/*8 horas 7 minutos 6 segundos*/
INTERVAL ('8:07:06') HOUR TO SECOND;
    
```

Esos intervalos se pueden sumar a valores de tipo DATE o TIMESTAMP

obtener la fecha y hora actual

Función	Descripción
SYSDATE	Obtiene la fecha y hora actuales
SYSTIMESTAMP	Obtiene la fecha y hora actuales en formato TIMESTAMP
DBTIMEZONE	Devuelve la zona horaria actual
CURRENT_DATE	Obtiene la fecha y hora actuales e incluye la zona horaria
CURRENT_TIMESTAMP	Obtiene la fecha y hora actuales en formato TIMESTAMP e incluye la zona horaria

calcular fechas

Función	Descripción
ADDMONTHS(<i>fecha,n</i>)	Añade a la fecha el número de meses indicado por <i>n</i>
MONTHS_BETWEEN(<i>fecha1, fecha2</i>)	Obtiene la diferencia en meses entre las dos fechas (puede ser decimal)

Función	Descripción
NEXT_DAY (<i>fecha, día</i>)	Indica cual es el día que corresponde a añadir a la fecha el día indicado. El día puede ser el texto 'Lunes', 'Martes', 'Miércoles',... (si la configuración está en español) o el número de día de la semana (1=lunes, 2=martes,...)
LAST_DAY (<i>fecha</i>)	Obtiene el último día del mes al que pertenece la fecha. Devuelve un valor DATE
EXTRACT (<i>valor FROM fecha</i>)	Extrae un valor de una fecha concreta. El valor puede ser <i>day</i> (día), <i>month</i> (mes), <i>year</i> (año), etc.
GREATEST (<i>fecha1, fecha2,..</i>)	Devuelve la fecha más moderna la lista
LEAST (<i>fecha1, fecha2,..</i>)	Devuelve la fecha más antigua la lista
ROUND (<i>fecha</i> [, ' <i>formato</i> ']	Redondea la fecha al valor de aplicar el formato a la fecha. El formato puede ser: <ul style="list-style-type: none"> ⊙ 'YEAR' Hace que la fecha refleje el año completo ⊙ 'MONTH' Hace que la fecha refleje el mes completo más cercano a la fecha ⊙ 'HH24' Redondea la hora a las 00:00 más cercanas ⊙ 'DAY'
TRUNC (<i>fecha</i> [<i>formato</i>])	Igual que el anterior pero trunca la fecha en lugar de redondearla.

funciones de conversión

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

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

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

Pero en diversas ocasiones queremos realizar conversiones explícitas.

TO_CHAR

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

fechas

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

Símbolo	Significado
YY	Año en formato de dos cifras

Símbolo	Significado
YYY	Últimas tres cifras del año
YYYY	Año en formato de cuatro cifras
SYYYY	igual que el anterior, pero si la fecha es anterior al nacimiento de Cristo el año aparece en negativo
MM	Mes en formato de dos cifras
MON	Las tres primeras letras del mes
MONTH	Nombre completo del mes
DY	Día de la semana en tres letras
DAY	Día completo de la semana
DD	Día en formato de dos cifras
Q	Semestre
WW	Semana del año
D	Día de la semana (del 1 al 7)
DDD	Día del año
AD A.D.	Indicador de periodo <i>Anno Domini</i> (después de Cristo)
BC B.C.	Indicador de periodo, antes de Cristo. Aparece en fechas anteriores al año cero (en español se pone AC)
J	Año juliano
RN	Método Romano de numeración
AM	Indicador AM
PM	Indicador PM
HH12	Hora de 1 a 12
HH24	Hora de 0 a 23
MI	Minutos (0 a 59)
SS	Segundos (0 a 59)
SSSS	Segundos desde medianoche
/.,	Posición de los separadores

Ejemplos:

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

números

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

Símbolo	Significado
9	Posición del número

Símbolo	Significado
O	Posición del número (muestra ceros)
S	En esa posición se coloca el signo del número (tanto el negativo como el positivo)
\$	Formato dólar
L	Símbolo local de la moneda
C	Símbolo internacional de moneda (según la configuración local de Oracle)
D	Posición del símbolo decimal (en español, la coma)
G	Posición del separador de grupo (en español el punto)
RN	Numeración romana en mayúsculas
rn	Numeración romana en minúsculas
PR	Se muestran los negativos entre símbolos < y >
.	Posición del decimal
,	Posición del separador de miles

TO_NUMBER

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

TO_DATE

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

funciones condicionales

CASE

Es una instrucción incorporada a la versión 9 de Oracle que permite establecer condiciones de salida (al estilo *if-then-else* de muchos lenguajes). Sintaxis:

```
CASE expresión WHEN valor1 THEN resultado1
[ WHEN valor2 THEN resultado2 ....
...
ELSE resultadoElse
]
END
```

El funcionamiento es el siguiente:

- 1 > Se evalúa la expresión indicada

- 2> Se comprueba si esa expresión es igual al valor del primer **WHEN**, de ser así se devuelve el primer resultado (cualquier valor excepto nulo)
- 3> Si la expresión no es igual al valor 1, entonces se comprueba si es igual que el segundo. De ser así se escribe el resultado 3. De no ser así se continúa con el siguiente **WHEN**
- 4> El resultado indicado en la zona **ELSE** sólo se escribe si la expresión no vale ningún valor de los indicados.

Ejemplo:

```
SELECT
  CASE cotizacion WHEN 1 THEN salario*0.85
                  WHEN 2 THEN salario * 0.93
                  WHEN 3 THEN salario * 0.96
                  ELSE salario
  END
FROM empleados;
```

función DECODE

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

Sintaxis:

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

Ejemplo:

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

Este ejemplo es idéntico al utilizado para la instrucción **CASE**

obtener datos de múltiples tablas

Es más que habitual necesitar en una consulta datos que se encuentran distribuidos en varias tablas. Las bases de datos relacionales se basan en que los datos se distribuyen en

tablas que se pueden relacionar mediante un campo. Ese campo es el que permite integrar los datos de las tablas.

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

producto cruzado o cartesiano de tablas

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

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

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

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

asociando tablas

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

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

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

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

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

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


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

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

relaciones sin igualdad

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

A veces esto no ocurre, en las tablas:

EMPLEADOS	
Empleado	Sueldo
Antonio	18000
Marta	21000
Sonia	15000
Andrés	11000
...	

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

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

La forma sería:

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

obtener registros no relacionados

En el ejemplo visto anteriormente de las tareas y los empleados. Podría ocurrir que un empleado no hubiera realizado una tarea todavía, con lo que habría empleados que no aparecerían en la consulta al no tener una tarea relacionada.

La forma de conseguir que salgan todos los registros de una tabla aunque no estén relacionados con las de otra es realizar una asociación lateral o unión externa (también llamada *outer join*). En esas asociaciones, el signo (+) indica que se desean todos los registros de la tabla estén o no relacionados.

Sintaxis:

```
SELECT tabla1.columna1, tabla1.columna2,...
       tabla2.columna1, tabla2.columna2,...
FROM tabla1, tabla2
WHERE tabla1.columnaRelacionada(+)=tabla2.columnaRelacionada
```

Eso obtiene los registros relacionados entre las tablas y además los registros no relacionados de la tabla2. Se podría usar esta otra forma:

```
SELECT tabla1.columna1, tabla1.columna2,...
       tabla2.columna1, tabla2.columna2,...
FROM tabla1, tabla2
WHERE tabla1.columnaRelacionada=tabla2.columnaRelacionada(+)
```

En ese caso salen los relacionados y los de la primera tabla que no estén relacionados con ninguno de la primera.

sintaxis SQL 1999

En la versión SQL de 1999 se ideó una nueva sintaxis para consultar varias tablas. La razón fue separar las condiciones de asociación respecto de las condiciones de selección de registros. La sintaxis completa es:

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

Se describen sus posibilidades

CROSS JOIN

Utilizando la opción CROSS JOIN se realiza un producto cruzado entre las tablas indicadas

NATURAL JOIN

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

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

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

JOIN USING

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

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

JOIN ON

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

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

relaciones externas

La última posibilidad es obtener relaciones laterales o externas (*outer join*). Para ello se utiliza la sintaxis:

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

En esta consulta además de las relacionadas, aparecen las piezas no relacionadas en existencias. Si el LEFT lo cambiamos por un RIGHT, aparecerán las existencias no presentes en piezas.

La condición FULL OUTER JOIN produciría un resultado en el que aparecen los registros no relacionados de ambas tablas.

agrupaciones

Es muy común utilizar consultas en las que se desee agrupar los datos a fin de realizar cálculos en vertical, es decir calculados a partir de datos de distintos registros. Para ello se utiliza la cláusula GROUP BY que permite indicar en base a qué registros se realiza la agrupación. Con GROUP BY la instrucción SELECT queda de esta forma:

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

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

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

Si la tabla de existencias sin agrupar es:

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

La consulta anterior creará esta salida:

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

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

```
SELECT tipo,modelo, cantidad
FROM existencias GROUP BY tipo,modelo;
SELECT tipo,modelo, cantidad
      *
ERROR en línea 1:
ORA-00979: no es una expresión GROUP BY
```

funciones de cálculo con grupos

Lo interesante de la creación de grupos es las posibilidades de cálculo que ofrece.

Para ello se utilizan funciones que permiten trabajar con los registros de un grupo son:

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

Todos esos valores se calculan para cada elemento del grupo, así la expresión:

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

Obtiene este resultado:

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

Se suman las cantidades para cada grupo

condiciones HAVING

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

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

Pero Oracle devolvería este error:

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

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

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

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

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

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

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

Pasos en la ejecución de una instrucción de agrupación por parte del gestor de bases de datos:

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

subconsultas

Se trata de una técnica que permite utilizar el resultado de una tabla SELECT en otra consulta SELECT. Permite solucionar problemas en los que el mismo dato aparece dos veces.

La sintaxis es:

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

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

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

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

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

La última consulta obtiene los empleados cuyas pagas estén entre lo que gana Luis y lo que gana Martina.

Una subconsulta que utilice los valores >, <, >=, ... tiene que devolver un único valor, de otro modo ocurre un error. Pero a veces se utilizan consultas del tipo: *mostrar el sueldo y nombre de los empleados cuyo sueldo supera al de cualquier empleado del departamento de ventas.*

La subconsulta necesaria para ese resultado mostraría los sueldos del departamento de ventas. Pero no podremos utilizar un operador de comparación directamente ya que compararíamos un valor con muchos valores. La solución a esto es utilizar instrucciones especiales entre el operador y la consulta. Esas instrucciones son:

Instrucción	Significado
ANY	Compara con cualquier registro de la consulta. La instrucción es válida si hay un registro en la subconsulta que permite que la comparación sea cierta
ALL	Compara con todos los registros de la consulta. La instrucción resulta cierta si es cierta toda

Instrucción	Significado
	comparación con los registros de la subconsulta
IN	No usa comparador, ya que sirve para comprobar si un valor se encuentra en el resultado de la subconsulta
NOT IN	Comprueba si un valor no se encuentra en una subconsulta

Ejemplo:

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

Esa consulta obtiene el empleado que más cobra. Otro ejemplo:

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

En ese caso se obtienen los nombres de los empleados cuyos *dni* están en la tabla de directivos.

combinaciones especiales

uniones

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

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

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

intersecciones

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

diferencia

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

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

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

comandos internos en SQL e iSQL*Plus

Lo que se comenta en este apartado son comandos y operaciones que no pertenecen al lenguaje SQL, sino que son comandos que sirven para dar instrucciones al programa SQL*Plus o iSQL*Plus.

Las operaciones que se comentan aquí son interpretadas por el cliente SQL*Plus y no por Oracle. Estas operaciones sirven sobre todo para variar la forma en la que se muestran los resultados de las consultas SQL.

Hay que tener en cuenta que hay cierta diferencia entre los comandos SQL*Plus e iSQL*Plus.

variables de sustitución

Se utilizan para poder dar parámetros a una consulta. Por ejemplo si a menudo se realiza un listado de clientes en el que queremos mostrar los datos de un cliente identificado por su DNI, entonces se puede utilizar una variable de sustitución para el DNI, de modo que cada vez que se ejecute esa consulta se pedirá el nuevo valor de la variable.

operador &

La primera forma de utilizar variables de sustitución es mediante el comando &. Este símbolo utilizado en cualquier parte de la consulta, permite rellenar el contenido de una variable de sustitución. Por ejemplo:

```
SELECT * FROM Piezas WHERE modelo=&mod;
```

Al ejecutar esa sentencia, desde el cliente SQL*Plus se nos pedirá rellenar el valor de la variable **mod**. Esa variable no se puede volver a usar, si se usa se nos invitará a indicar el valor que la damos. La ventaja de esta técnica está en que cada vez que ejecutemos podremos dar un valor a la variable, lo que nos permite reutilizar consultas una y otra vez para distintos valores, sin tener que reescribirla.

En el caso de que la variable sea de texto, hay que colocar el símbolo & dentro de las comillas que delimitan el texto. Ejemplo:

```
SELECT * FROM Piezas WHERE tipo='&tip';
```

Es decir se trata de una macro-sustitución en la que el contenido de la variable se sustituye por su contenido antes de pasar la instrucción a Oracle, de ahí que sea necesario colocar las comillas, de otro modo Oracle indicaría que la instrucción es errónea.

define

Se pueden utilizar variables de sustitución que se definan como variables de usuario mediante la operación DEFINE. La sintaxis de este comando es:

```
DEFINE variable=valor;
```

La variable se sobreentiende que es de tipo texto. El valor es el contenido inicial de la variable. La variable así creada tiene vigencia durante toda la sesión de usuario. Se elimina

en el cierre de la sesión o si se usa el comando **UNDEFINE** indicando el nombre de la variable a eliminar.

Para cambiar el valor de la variable se debe utilizar otra vez el comando **DEFINE**. La ventaja respecto al método anterior está en que la misma variable de sustitución se puede utilizar para varios **SELECT**. La desventaja está en que requiere tocar el código para cambiar el valor de la variable. Ejemplo:

```
DEFINE tip='TU';
SELECT * FROM piezas where tipo='&tip';
SELECT * FROM existencias where tipo='&tip';
```

En el ejemplo, los dos **SELECT** muestran piezas cuyo tipo sea TU. **SQL*Plus** no preguntará por el valor de la variable **tip**.

listar variables

El comando **DEFINE** sin nada más permite mostrar una lista de todas las variables definidas en ese momento.

operador &&

Se trata de una mezcla entre las anteriores. Cuando en una consulta se utiliza una variable de sustitución mediante dos símbolos *ampersand*, entonces al ejecutar la consulta se nos preguntará el valor. Pero luego ya no, la variable queda definida como si se hubiera declarado con **DEFINE**.

El resto de veces que se utilice la variable, se usa con un solo **&**. El cambio de valor de la variable habrá que realizarle con **DEFINE**.

comando SET

Este comando permite cambiar el valor de las variables de entorno del programa. Su uso es:

```
SET nombreVariable valor
```

las variables más interesantes a utilizar son:

variable	posibles valores	explicación
ECHO	ON (activado) y OFF (desactivado)	Repite el comando SQL antes de mostrar su resultado
TIMING	ON (activado) y OFF (desactivado)	Permite mostrar estadísticas sobre el tiempo de ejecución en cada consulta SQL que se ejecute (interesante para estadísticas)
HEADING	ON (activado) y OFF (desactivado)	Hace que el encabezado con los alias de las columnas se active o no
WRAP	ON (activado) y OFF (desactivado)	Activado, trunca un texto si sobrepasa la anchura máxima.
COMPATIBILITY	V7, V8, NATIVE	Permite indicar la versión con la que se comprueba la compatibilidad de los

variable	posibles valores	explicación
		comandos. NATIVE indica que el propio servidor Oracle decide la compatibilidad
DEFINE	& , <i>carácter</i> , ON (<i>activado</i>) y OFF (<i>desactivado</i>)	Permite activar y desactivar la posibilidad de usar variables de sustitución. Permite indicar el carácter utilizado para la sustitución de variables
PAGESIZE	<i>n</i>	Indica el número de filas que se muestran antes de repetir el encabezado de la consulta
LINESIZE	<i>n</i>	Indica la anchura máxima de la línea de la consulta. Si una línea de la consulta sobrepasa este valor, los datos pasan a la siguiente. También influye sobre los tamaños y posiciones de los encabezados y pies de los informes
NULL	<i>valor</i>	Indica qué valor se muestra cuando hay nulos
NUMFORMAT	<i>formato</i>	Permite especificar un formato que se aplicará a todos los números. (Véase formato de columnas, más adelante)
NUMWIDTH	<i>valor</i>	Indica la anchura máxima utilizada para mostrar números. Si un número sobrepasa esta anchura, es redondeado
FEEDBACK	<i>n</i> , ON (<i>activado</i>) y OFF (<i>desactivado</i>)	Hace que se muestren el número total de registros de la consulta cuando el resultado supera los <i>n</i> registros.
LONG	<i>ancho</i>	Anchura máxima para los campos de tipo LONG

SHOW

El comando SHOW seguido de el nombre de uno de los parámetros de la tabla anterior, permite mostrar el estado actual del parámetro indicado. Si se usa **SHOW ALL**, entonces se muestran todos

encabezado y pie de informe

Los parámetros **BTITLE** y **TTITLE** permiten, respectivamente, indicar un texto de pie y de encabezado para la consulta. El formato es

```
{B|T}TITLE {texto|ON|OFF}
```

El texto es lo que se desea en el encabezado o pie. Ese texto puede incluir las palabras LEFT, RIGHT o CENTER para hacer que el texto vaya a izquierda, centro o derecha respectivamente.

Se pueden indicar incluso las tres cosas a la vez:

```
TTITLE LEFT 'informe1' RIGHT 'estudio de clientes';
```

Se puede también usar la palabra COL seguida del número de columna en el que se desea el texto:

```
TTITLE COL 50 'informe1';
```

También se puede indicar la palabra TAB seguida de un número que representará tabulaciones, haciendo que SQL*Plus deje ese espacio en encabezado o pie.

COLUMN

Permite especificar un formato de columna. Si se usa sin modificador (sólo COLUMNS o su abreviatura COL) se muestran las configuraciones de formato de columnas actualmente en uso. Si se añade el nombre de una columna, se indica el formato actual (si lo hay) para esa columna.

añadir formato de columna

HEADING

Si se usa COLUMN (o COL) con el parámetro HEADING seguido de un texto, el texto se convierte en la cabecera de la columna (sustituyendo al alias de la columna). Ese texto sólo sirve para ser mostrado, no puede formar parte de una sentencia SQL.

```
COLUMN precio_venta HEADING 'Precio de|venta';
SELECT tipo, modelo, precio_venta FROM piezas;
```

En el ejemplo, la barra vertical provoca un salto de línea en el texto.

FORMAT

Permite indicar una máscara de formato para el texto. Se usan códigos especiales para ello.

textos

A los textos se les puede colocar una anchura máxima de columna. Eso se hace con una **A** seguida del número que indica esa anchura.

```
COLUMN tipo FORMAT 'A10';
SELECT tipo, modelo, precio_venta FROM piezas;
```

números

Se usan códigos de posición:

código	significado
9	Posición para un número. Si el valor es 0 o vacío, entonces no se muestra nada
0	Posición para un número. Si el valor es 0 o vacío, entonces se muestra 0

código	significado
\$	Posición para el signo de dólar
MI	Muestra un signo menos tras el número, si el número es negativo (ejemplo: '999MI')
S	Muestra el signo del número (+ ó -) en la posición en la que se coloca el signo
PR	Muestra los números negativos entre < y >
D	Muestra el signo decimal en esa posición
G	Muestra el signo de grupo en esa posición
.	Muestra el punto (separador decimal)
,	Muestra la coma (separador de miles)
L	Muestra el símbolo de moneda nacional en esa posición
RN	Muestra el número en romano (mayúsculas)
rn	Muestra el número en romano (minúsculas)

fechas

Las fechas deben ser formateadas desde la propia instrucción SQL mediante la función TO_CHAR (vista en el tema anterior)

LIKE

Permite copia atributos de una columna a otra:

```
COLUMN precio_venta FORMAT '9G990D00L';  
COLUMN precio_compra LIKE precio_venta;
```

Las dos columnas tendrán el mismo formato (separador de miles, decimales y moneda tras los dos decimales)

NULL

Indica un texto que sustituirá a los valores nulos.

CLEAR

Elimina el formato de la columna

Lógicamente se pueden combinar varias acciones a la vez

BREAK

Es uno de los comandos más poderosos. Permite realizar agrupaciones en las consultas, consiguiendo verdaderos informes (en especial si se combina con COMPUTE).

Permite dividir la vista en secciones en base al valor de un campo al que se le pueden incluso quitar los duplicados. El comando:

```
BREAK ON tipo;
```

Hace que las columnas con alias *tipo* no muestren los duplicados, mostrando una vez cada valor duplicado. Para el buen funcionamiento de la orden, el resultado debe de estar ordenado por esa columna.

Se pueden hacer varios grupos a la vez:

```
BREAK ON tipo ON modelo;
```

La orden **CLEAR BREAK** elimina todos los **BREAK** anteriormente colocados.

SKIP

A la instrucción anterior se le puede añadir la palabra **SKIP** seguida de un número. Ese número indica las líneas que se dejan tras el valor del grupo al imprimir. Si se indica **SKIP PAGE**, significa que se salta una página completa cada vez que cambie el valor del grupo.

ON REPORT

Permite (en unión con **COMPUTE**) realizar cálculos de totales sobre el informe completo.

DUPLICATES y NODUPLICATES

Permiten mostrar o no los duplicados de cada sección. La opción inicial es **NODUPLICATES**.

COMPUTE

Permite en unión con **BREAK** realizar cálculos para las secciones de una consulta. Todo **COMPUTE** está asociado a un apartado **ON** de una instrucción **BREAL** previa. Sintaxis:

```
COM[PUTE] [función [LAB[EL] texto] OF columnaOAlias  
ON {columnaOAlias|REPORT}
```

- ⊙ *función*. Es el nombre de la función de cálculo que se usa (SUM, AVG, MIN, MAX, NUM (esta es como COUNT) , STD o VAR)
- ⊙ **LABEL**. Permite indicar un texto previo al resultado del cálculo, si no se utiliza se pone el nombre de la función utilizada.
- ⊙ **OF**. Indica el nombre de columna o alias utilizado en la instrucción **SELECT** a partir de la que se realiza el cálculo
- ⊙ **ON**. Indica el nombre de columna o alias que define la sección sobre la que se realizará el cálculo. Este nombre debe haber sido indicado en un **BREAK** anterior.
- ⊙ **REPORT**. Indica que el cálculo se calculará para toda la consulta.

```

CLEAR BREAK;
COLUMN TIPO FORMAT A20;
BREAK ON tipo SKIP PAGE ON modelo ON REPORT;
COMPUTE SUM LABEL 'Total' MAX LABEL 'Máximo' -
  OF cantidad ON tipo;
COMPUTE SUM LABEL 'Total' OF cantidad ON modelo;
COMPUTE SUM LABEL 'Total absoluto' OF cantidad ON REPORT;

SELECT tipo, modelo, n_almacen, cantidad FROM existencias
WHERE tipo='AR' OR tipo='TU'
ORDER BY tipo, modelo;

```

Resultado:

TIPO	MODELO	N_ALMACEN	CANTIDAD
AR	6	1	2500
		2	5600
		3	2430
	*****		-----
	Total		10530
	9	1	250
		2	4000
		3	678
	*****		-----
	Total		4928
	15	1	5667
	*****		-----
	Total		5667
	20	3	43
	*****		-----
	Total		43
*****			-----
Máximo			5667
Total			21168
TIPO	MODELO	N_ALMACEN	CANTIDAD
TU	6	2	234
		3	43
	*****		-----
	Total		277
	9	2	876
	*****		-----
	Total		876
	10	2	1023

	*****		-----
	Total		1023
	12	2	234
	*****		-----
	Total		234
	16	2	654
	*****		-----
	Total		654
*****			-----
Máximo			1023
Total			3064
TIPO	MODELO	N_ALMACEN	CANTIDAD

Total absoluto			24232

DML

introducción

Es una de las partes fundamentales del lenguaje SQL. El DML (*Data Manipulation Language*) lo forman las instrucciones capaces de modificar los datos de las tablas. Al conjunto de instrucciones DML que se ejecutan consecutivamente, se las llama **transacciones** y se pueden anular todas ellas o aceptar, ya que una instrucción DML no es realmente efectuada hasta que no se acepta (**commit**).

En todas estas consultas, el único dato devuelto por Oracle es el número de registros que se han modificado.

inserción de datos

La adición de datos a una tabla se realiza mediante la instrucción **INSERT**. Su sintaxis fundamental es:

```
INSERT INTO tabla [(listaDeCampos)]  
VALUES (valor1 [,valor2 ...])
```

La *tabla* representa la tabla a la que queremos añadir el registro y los valores que siguen a **VALUES** son los valores que damos a los distintos campos del registro. Si no se especifica la lista de campos, la lista de valores debe seguir el orden de las columnas según fueron creados (es el orden de columnas según las devuelve el comando **DESCRIBE**).

La lista de campos a rellenar se indica si no queremos rellenar todos los campos. Los campos no rellenados explícitamente con la orden **INSERT**, se rellenan con su valor por defecto (**DEFAULT**) o bien con **NULL** si no se indicó valor alguno. Si algún campo tiene restricción de tipo **NOT NULL**, ocurrirá un error si no rellenamos el campo con algún valor.

Por ejemplo, supongamos que tenemos una tabla de clientes cuyos campos son: dni, nombre, apellido1, apellido2, localidad y dirección; supongamos que ese es el orden de creación de los campos de esa tabla y que la localidad tiene como valor por defecto *Palencia* y la dirección no tiene valor por defecto. En ese caso estas dos instrucciones son equivalentes:

```
INSERT INTO clientes  
VALUES('11111111','Pedro','Gutiérrez','Crespo',DEFAULT,NULL);  
  
INSERT INTO clientes(dni,nombre,apellido1,apellido2)  
VALUES('11111111','Pedro','Gutiérrez','Crespo')
```

Son equivalentes puesto que en la segunda instrucción los campos no indicados se rellenan con su valor por defecto y la dirección no tiene valor por defecto. La palabra **DEFAULT** fuerza a utilizar ese valor por defecto.

El uso de los distintos tipos de datos debe de cumplir los requisitos ya comentados en temas anteriores (véase tipos de datos, página 15).

relleno de registros a partir de filas de una consulta

Hay un tipo de consulta, llamada de adición de datos, que permite rellenar datos de una tabla copiando el resultado de una consulta.

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

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

Ejemplo:

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

actualización de registros

La modificación de los datos de los registros lo implementa la instrucción **UPDATE**. Sintaxis:

```
UPDATE tabla
  SET columna1=valor1 [,columna2=valor2...]
  [WHERE condición]
```

Se modifican las columnas indicadas en el apartado **SET** con los valores indicados. La cláusula **WHERE** permite especificar qué registros serán modificados.

Ejemplos:

```
UPDATE clientes SET provincia='Ourense'
  WHERE provincia='Orense';

UPDATE productos SET precio=precio*1.16;
```

El primer dato actualiza la provincia de los clientes de Orense para que aparezca como Ourense. El segundo **UPDATE** incrementa los precios en un 16%. La expresión para el valor puede ser todo lo compleja que se desee:

```
UPDATE partidos SET fecha= NEXT_DAY(SYSDATE,'Martes')
  WHERE fecha=SYSDATE;
```

Incluso se pueden utilizar subconsultas:

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

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

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

borrado de registros

Se realiza mediante la instrucción DELETE:

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

Es más sencilla que el resto, elimina los registros de la tabla que cumplan la condición indicada. Ejemplos:

```
DELETE FROM empleados
WHERE seccion=23;

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

Hay que tener en cuenta que el borrado de un registro no puede provocar fallos de integridad y que la opción de integridad ON DELETE CASCADE (véase página 21, clave secundaria o foránea) hace que no sólo se borren los registros indicados en el SELECT, sino todos los relacionados.

comando MERGE

Sin duda alguna el comando más poderoso del lenguaje de manipulación de Oracle es MERGE. Este comando sirve para actualizar los valores de los registros de una tabla a partir de valores de registros de otra tabla o consulta. Permite pues combinar los datos de dos tablas a fin de actualizar la primera

Supongamos que poseemos una tabla en la que queremos realizar una lista de localidades con su respectiva provincia. Las localidades están ya rellenas, nos faltan las provincias. Resulta que tenemos otra tabla llamada clientes en la que tenemos datos de localidades y provincias, gracias a esta tabla podremos rellenar los datos que faltan en la otra. La situación se muestra en la ilustración siguiente.

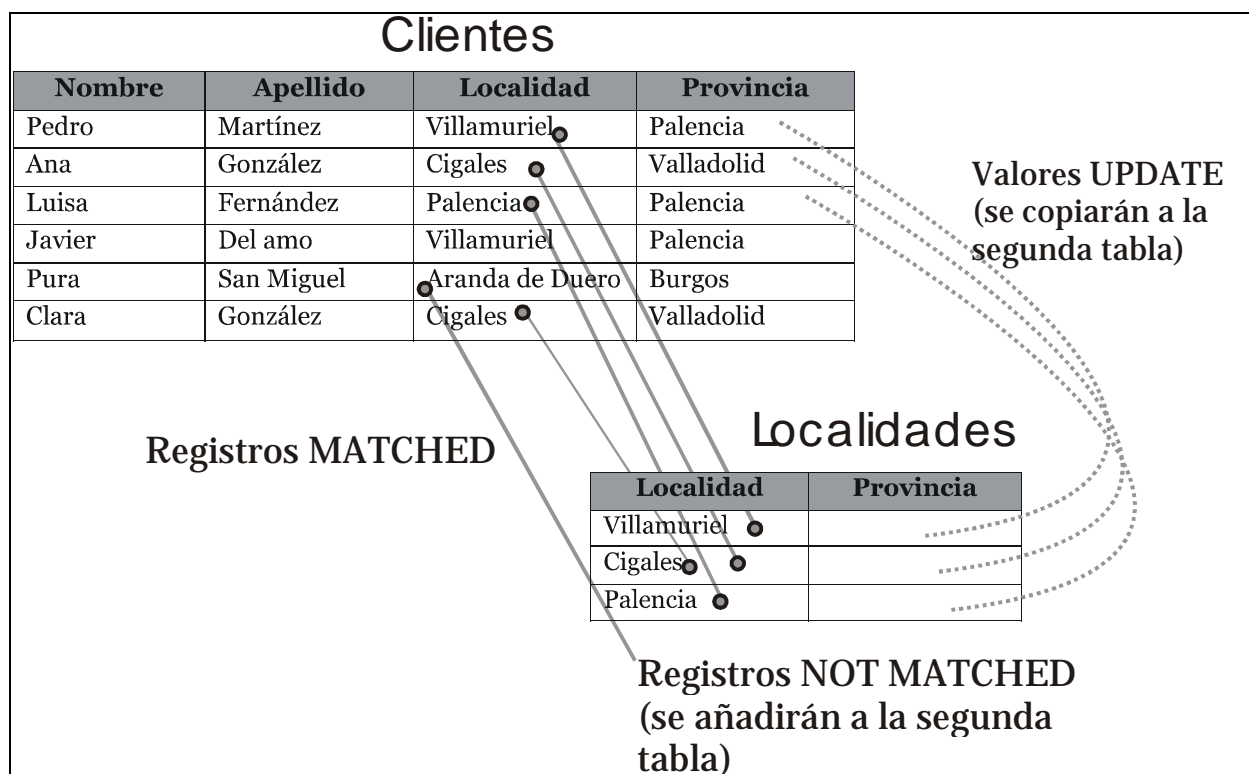


Ilustración 2, Ejemplo de uso de MERGE

La sintaxis del comando MERGE es:

```

MERGE INTO tabla alias
USING (instrucción SELECT) alias
ON (condición de unión)
WHEN MATCHED THEN
    UPDATE SET col1=valor1 [col2=valor2]
WHEN NOT MATCHED THEN
    INSERT (listaDeColumnas)
    VALUES (listaDeValores)
    
```

MERGE compara los registros de ambas tablas según la condición indicada en el apartado ON. Compara cada registro de la tabla con cada registro del SELECT. Los apartados de la sintaxis significan lo siguiente:

- *tabla* es el nombre de la tabla que queremos modificar
- **USING.** En esa cláusula se indica una instrucción SELECT tan compleja como queramos que muestre una tabla que contenga los datos a partir de los cuales se modifica la tabla
- **ON.** permite indicar la condición que permite relacionar los registros de la tabla con los registros de la consulta SELECT

- ⦿ **WHEN MATCHED THEN.** El UPDATE que sigue a esta parte se ejecuta cuando la condición indicada en el apartado ON sea cierta para los dos registros actuales.
- ⦿ **WHEN NOT MATCHED THEN.** El INSERT que sigue a esta parte se ejecuta para cada registro de la consulta SELECT que no pudo ser relacionado con ningún registro de la tabla.

Para el ejemplo descrito antes la instrucción MERGE sería:

```
MERGE INTO localidades l
USING (SELECT * FROM clientes) c
ON (l.localidad=clientes.localidad)
WHEN MATCHED THEN
    UPDATE SET l.provincia=c.provincia
WHEN NOT MATCHED THEN
    INSERT (localidad, provincia)
VALUES (c.localidad, c.provincia)
```

El resultado es la siguiente tabla de localidades:

Localidad	Provincia
Villamuriel	Palencia
Cigales	Valladolid
Palencia	Palencia
Aranda de Duero	Burgos

transacciones

Como se ha comentado anteriormente, una transacción está formada por una serie de instrucciones DML. Una transacción comienza con la primera instrucción DML que se ejecute y finaliza con alguna de estas circunstancias:

- ⦿ Una operación **COMMIT** o **ROLLBACK**
- ⦿ Una instrucción DDL (como ALTER TABLE por ejemplo)
- ⦿ Una instrucción DCL (como GRANT)
- ⦿ El usuario abandona la sesión
- ⦿ Caída del sistema

Hay que tener en cuenta que cualquier instrucción DDL o DCL da lugar a un COMMIT implícito, es decir todas las instrucciones DML ejecutadas hasta ese instante pasan a ser definitivas.

COMMIT

La instrucción COMMIT hace que los cambios realizados por la transacción sean definitivos, irrevocables. Sólo se debe utilizar si estamos de acuerdo con los cambios, conviene asegurarse mucho antes de realizar el COMMIT ya que las instrucciones ejecutadas pueden afectar a miles de registros.

Además el cierre correcto de la sesión da lugar a un COMMIT, aunque siempre conviene ejecutar explícitamente esta instrucción a fin de asegurarnos de lo que hacemos.

ROLLBACK

Esta instrucción regresa a la instrucción anterior al inicio de la transacción, normalmente el último COMMIT, la última instrucción DDL o DCL o al inicio de sesión. Anula definitivamente los cambios, por lo que conviene también asegurarse de esta operación.

Un abandono de sesión incorrecto o un problema de comunicación o de caída del sistema dan lugar a un ROLLBACK implícito.

SAVEPOINT

Esta instrucción permite establecer un punto de ruptura. El problema de la combinación ROLLBACK/COMMIT es que un COMMIT acepta todo y un ROLLBACK anula todo. SAVEPOINT permite señalar un punto intermedio entre el inicio de la transacción y la situación actual. Su sintaxis es:

```
...instrucciones DML...  
SAVEPOINT nombre  
...instrucciones DML...
```

Para regresar a un punto de ruptura concreto se utiliza ROLLBACK TO SAVEPOINT seguido del nombre dado al punto de ruptura. Cuando se vuelve a un punto marcado, las instrucciones que siguieron a esa marca se anulan definitivamente.

estado de los datos durante la transacción

Si se inicia una transacción usando comandos DML hay que tener en cuenta que:

- Se puede volver a la instrucción anterior a la transacción cuando se desee
- Las instrucciones de consulta SELECT realizadas por el usuario que inició la transacción muestran los datos ya modificados por las instrucciones DML
- El resto de usuarios ven los datos tal cual estaban antes de la transacción, de hecho los registros afectados por la transacción aparecen bloqueados hasta que la transacción finalice. Esos usuarios no podrán modificar los valores de dichos registros.

Tras la transacción todos los usuarios ven los datos tal cual quedan tras el fin de transacción. Los bloqueos son liberados y los puntos de ruptura borrados.

objetos de la base de datos

vistas

introducción

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

Las vistas se emplean para:

- ⊙ Realizar consultas complejas más fácilmente
- ⊙ Proporcionar tablas con datos completos
- ⊙ Utilizar visiones especiales de los datos

Hay dos tipos de vistas:

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

creación de vistas

Sintaxis:

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

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

- **WITH READ ONLY.** Hace que la vista sea de sólo lectura. Permite grabar un nombre para esta restricción.

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

Ejemplo:

```
CREATE VIEW resumen
/* alias */
(id_localidad, localidad, poblacion, n_provincia, provincia,
superficie, capital_provincia,
id_comunidad, comunidad, capital_comunidad)

AS
(
  SELECT l.id_localidad, l.nombre, l.poblacion,
         n_provincia, p.nombre, p.superficie, l2.nombre,
         id_comunidad, c.nombre, l3.nombre
  FROM localidades l
  JOIN provincias p USING (n_provincia)
  JOIN comunidades c USING (id_comunidad)
  JOIN localidades l2 ON (p.id_capital=l2.id_localidad)
  JOIN localidades l3 ON (c.id_capital=l3.id_localidad)
)

SELECT DISTINCT (comunidad, capital_comunidad) FROM resumen;
```

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

ejecución de comandos DML sobre vistas

Las instrucciones DML ejecutadas sobre las vistas permiten añadir o modificar los datos de las tablas relacionados con las filas de la vista. Ahora bien, no es posible ejecutar instrucciones DML sobre vistas que:

- Utilicen funciones de grupo (SUM, AVG,...)
- Usen GROUP BY o DISTINCT
- Posean columnas con cálculos (PRECIO * 1.16)

Además no se pueden añadir datos a una vista si en las tablas referencias en la consulta SELECT hay campos NOT NULL que no aparecen en la consulta (es lógico ya que al añadir el dato se tendría que añadir el registro colocando el valor NULL en el campo). Ejemplo (sobre la vista anterior):

```
INSERT INTO resumen(id_localidad, localidad, poblacion)
VALUES (10000, 'Sevilla', 750000)
```

mostrar la lista de vistas

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

```
SELECT * FROM USER_VIEWS;
```

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

borrar vistas

Se utiliza el comando **DROP VIEW**:

```
DROP VIEW nombreDeVista;
```

secuencias

Una secuencia sirve para generar automáticamente números distintos. Se utilizan para generar valores para campos que se utilizan como clave forzada (claves cuyo valor no interesa, sólo sirven para identificar los registros de una tabla).

Es una rutina interna de Oracle la que realiza la función de generar un número distinto cada vez. Las secuencias se almacenan independientemente de la tabla, por lo que la misma secuencia se puede utilizar para diversas tablas.

creación de secuencias

Sintaxis:

```
CREATE SEQUENCE secuencia
  [INCREMENT BY n]
  [START WITH n]
  [ {MAXVALUE n | NOMAXVALUE } ]
  [ {MINVALUE n | NOMINVALUE } ]
  [ {CYCLE | NOCYCLE } ]
```

Donde:

- ⊙ *secuencia*. Es el nombre que se le da al objeto de secuencia
- ⊙ **INCREMENT BY**. Indica cuánto se incrementa la secuencia cada vez que se usa. Por defecto se incrementa de uno en uno
- ⊙ **START WITH**. Indica el valor inicial de la secuencia (por defecto 1)
- ⊙ **MAXVALUE**. Máximo valor que puede tomar la secuencia. Si no se toma **NOMAXVALUE** que permite llegar hasta el 10^{27}
- ⊙ **MINVALUE**. Mínimo valor que puede tomar la secuencia. Por defecto -10^{26}

- **CYCLE.** Hace que la secuencia vuelva a empezar si se ha llegado al máximo valor.

Ejemplo:

```
CREATE SEQUENCE numeroPlanta
INCREMENT 100
STARTS WITH 100
MAXVALUE 2000
```

ver lista de secuencias

La vista del diccionario de datos **USER_SEQUENCES** muestra la lista de secuencias actuales. La columna **LAST_NUMBER** muestra cual será el siguiente número de secuencia disponible

uso de la secuencia

Los métodos **NEXTVAL** y **CURRVAL** se utilizan para obtener el siguiente número y el valor actual de la secuencia respectivamente. Ejemplo de uso:

```
SELECT numeroPlanta.NEXTVAL FROM DUAL;
```

Eso muestra en pantalla el siguiente valor de la secuencia. Realmente **NEXTVAL** incrementa la secuencia y devuelve el valor actual. **CURRVAL** devuelve el valor de la secuencia, pero sin incrementar la misma.

Ambas funciones pueden ser utilizadas en:

- Una consulta **SELECT** que no lleve **DISTINCT**, ni grupos, ni sea parte de una vista, ni sea subconsulta de otro **SELECT**, **UPDATE** o **DELETE**
- Una subconsulta **SELECT** en una instrucción **INSERT**
- La cláusula **VALUES** de la instrucción **INSERT**
- La cláusula **SET** de la instrucción **UPDATE**

No se puede utilizar (y siempre hay tentaciones para ello) como valor para la cláusula **DEFAULT** de un campo de tabla.

Su uso más habitual es como apoyo al comando **INSERT**:

```
INSERT INTO plantas(num, uso)
VALUES(numeroPlanta.NEXTVAL, 'Suites');
```

modificar secuencias

Se pueden modificar las secuencias, pero la modificación sólo puede afectar a los futuros valores de la secuencia, no a los ya utilizados. Sintaxis:

```
ALTER SEQUENCE secuencia
  [ INCREMENT BY n ]
  [ START WITH n ]
  [ { MAXVALUE n | NOMAXVALUE } ]
  [ { MINVALUE n | NOMINVALUE } ]
  [ { CYCLE | NOCYCLE } ]
```

borrar secuencias

Lo hace el comando **DROP SEQUENCE** seguido del nombre de la secuencia a borrar.

índices

Los índices son esquemas que hacen que Oracle acelere las operaciones de consulta y ordenación sobre los campos a los que el índice hace referencia.

Se almacenan aparte de la tabla a la que hace referencia, lo que permite crearles y borrarles en cualquier momento.

Lo que realizan es una lista ordenada por la que Oracle puede acceder para facilitar la búsqueda de los datos. cada vez que se añade un nuevo registro, los índices involucrados se actualizan a fin de que su información esté al día. De ahí que cuantos más índices haya, más le cuesta a Oracle añadir registros, pero más rápidas se realizan las instrucciones de consulta.

La mayoría de los índices se crean de manera implícita, como consecuencia de las restricciones PRIMARY KEY (que obliga a crear un índice único sobre los campos clave) , UNIQUE (crea también un índice único) y FOREIGN KEY (crea un índice con posibilidad de repetir valores, índice con duplicados). Estos son índices obligatorios, por los que les crea el propio Oracle.

creación de índices

Aparte de los índices obligatorios comentados anteriormente, se pueden crear índices de forma explícita. Éstos se crean para aquellos campos sobre los cuales se realizarán búsquedas e instrucciones de ordenación frecuente.

Sintaxis:

```
CREATE INDEX nombre
ON tabla (columna1 [,columna2...])
```

Ejemplo:

```
CREATE INDEX nombre_completo
ON clientes (apellido1, apellido2, nombre);
```

El ejemplo crea un índice para los campos apellido1, apellido2 y nombre. Esto no es lo mismo que crear un índice para cada campo, este índice es efectivo cuando se buscan u ordenan clientes usando los tres campos (*apellido1, apellido2, nombre*) a la vez.

Se aconseja crear índices en campos que:

- Contengan una gran cantidad de valores
- Contengan una gran cantidad de nulos
- Son parte habitual de cláusulas WHERE, GROUP BY u ORDER BY
- Son parte de listados de consultas de grandes tablas sobre las que casi siempre se muestran como mucho un 4% de su contenido.

No se aconseja en campos que:

- Pertenecan a tablas pequeñas
- No se usan a menudo en las consultas
- Pertenecen a tablas cuyas consultas muestran más de un 6% del total de registros
- Pertenecen a tablas que se actualizan frecuentemente
- Se utilizan en expresiones

Los índices se pueden crear utilizando expresiones complejas:

```
CREATE INDEX nombre_complejo
ON clientes (UPPER(nombre));
```

Esos índices tienen sentido si en las consultas se utilizan exactamente esas expresiones.

lista de índices

Para ver la lista de índices se utiliza la vista **USER_INDEXES** . Mientras que la vista **USER_IND_COLUMNS** Muestra la lista de columnas que son utilizadas por índices.

borrar índices

La instrucción DROP INDEX seguida del nombre del índice permite eliminar el índice en cuestión.

sinónimos

Un sinónimo es un nombre que se asigna a un objeto cualquiera. Normalmente es un nombre menos descriptivo que el original a fin de facilitar la escritura del nombre del objeto en diversas expresiones.

creación

Sintaxis:

```
CREATE [PUBLIC] SYNONYM nombre FOR objeto;
```

objeto es el objeto al que se referirá el sinónimo. La cláusula PUBLIC hace que el sinónimo esté disponible para cualquier usuario (sólo se permite utilizar si disponemos de privilegios administrativos).

borrado

```
DROP SYNONYM nombre
```

lista de sinónimos

La vista **USER_SYNONYMS** permite observar la lista de sinónimos del usuario, la vista **ALL_SYNONYMSE** permite mostrar la lista completa de sinónimos.

consultas avanzadas

consultas con ROWNUM

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

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

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

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

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

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

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

consultas sobre estructuras jerárquicas

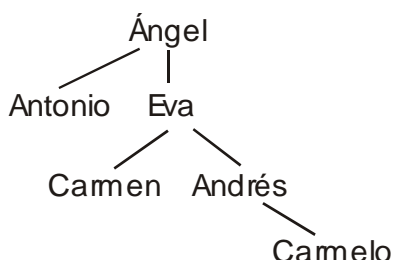
Imaginemos una tabla de empleados definida por un código de empleado, nombre del mismo y el código del jefe. Este último código está relacionado con el código de empleado que posee el jefe en cuestión. Así definido, una consulta que muestre el nombre de un empleado y el nombre de su jefe directo, sería:

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

Saldría por ejemplo:

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

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



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

```

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

```

El apartado CONNECT permite indicar que relación hay que seguir para recorrer el árbol. La palabra PRIOR indica hacia dónde se dirige el recorrido. Finalmente el apartado START indica la condición de inicio del recorrido (normalmente la condición que permita buscar el nodo del árbol por el que comenzamos el recorrido, es decir sirve para indicar desde donde comenzamos. Ejemplo:

```

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

```

Resultado:

NOMBRE
Andrés
Eva
Ángel

Sin embargo:

```

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

```

Devuelve:

NOMBRE
Andrés
Carmelo

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

NOMBRE
Ángel
Antonio
Eva
Carmen
Andrés
Carmelo

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

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

Resultado:

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

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

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

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

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

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

subconsultas avanzadas

Ya se comentó en el apartado subconsultas (página 46) como utilizar las subconsultas. Este tipo de consultas permiten comprobar si un dato se encuentra relacionado con datos que proceden de una segunda consulta.

Aquí se comentan algunas mejoras a ese tipo de consultas implementadas por Oracle.

subconsultas sobre múltiples valores

Las operaciones de subconsulta comentadas anteriormente permiten comprar un valor con el resultado de una subconsulta. También se pueden comparar varios valores:

```
SELECT * FROM piezas
WHERE (tipo, modelo) IN
      (SELECT tipo,modelo FROM EXISTENCIAS);
```

Lógicamente los valores entre paréntesis deben de coincidir, es decir si entre paréntesis se hace referencia a tres campos, el SELECT interior debe devolver tres campos exactamente del mismo tipo que los del paréntesis.

subconsultas correlacionadas

Las subconsultas correlacionadas hacen un proceso fila a fila, de modo que la subconsulta se ejecuta una vez por cada fila de la consulta principal.

Esto es absolutamente diferente respecto a la ejecución normal de una subconsulta, ya que normalmente la subconsulta se ejecuta primero, y con sus resultados se ejecuta la consulta principal. La sintaxis de este tipo de consultas es:

```
SELECT listaDeColumnas
FROM tabla alias
WHERE expresion operador (SELECT listaDeExpresiones+
                          FROM tabla2
                          WHERE expr1 = alias.expr2)
```

Ejemplo:

```
SELECT nombre, salario, cod_departamento
FROM empleados emp
WHERE salario >(SELECT AVG(salario)
                FROM empleados
                WHERE departamento = emp.departamento)
```

Este ejemplo muestra los datos de los empleados cuyo sueldo supera la media de su departamento.

consultas EXISTS

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

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

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

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

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

consultas de agrupación avanzada

ROLLUP

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

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

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

Si añadimos:

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

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

TI	MODELO	SUM(CANTIDAD)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
AR		21168
BI	10	363
BI	38	1740
BI	57	1638
BI		3741
CL	12	7000
CL	15	3068
CL	18	6612
CL		16680
EM	21	257
EM	42	534
EM		791
PU	5	12420
PU	9	7682
PU		20102
TO	6	464
TO	9	756
TO	10	987
TO	12	7740
TO	16	356
TO		10303
TU	6	277
TU	9	876
TU	10	1023
TU	12	234
TU	16	654
TU		3064
		75849

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

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

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

CUBE

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

Ejemplo:

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

Resulta:

TI	MODELO	SUM(CANTIDAD)
		75849
	5	12420
	6	11271
	9	14242
	10	2373
	12	14974
	15	8735
	16	1010
	18	6612
	20	43
	21	257
	38	1740
	42	534
	57	1638
AR		21168
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI		3741
BI	10	363
BI	38	1740
BI	57	1638
CL		16680
CL	12	7000
CL	15	3068
CL	18	6612

TI	MODELO	SUM(CANTIDAD)
EM		791
EM	21	257
EM	42	534
PU		20102
PU	5	12420
PU	9	7682
TO		10303
TO	6	464
TO	9	756
TO	10	987
TO	12	7740
TO	16	356
TU		3064
TU	6	277
TU	9	876
TU	10	1023
TU	12	234
TU	16	

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

GROUPING

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

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

Sale:

TIPO	MODELO	SUM(CANTIDAD)	GROUPING(TIPO)	GROUPING(MODELO)
		75849	1	1
	5	12420	1	0
	6	11271	1	0
	9	14242	1	0
	10	2373	1	0
	12	14974	1	0
	15	8735	1	0
	16	1010	1	0
	18	6612	1	0
	20	43	1	0

TIPO	MODELO	SUM(CANTIDAD)	GROUPING(TIPO)	GROUPING(MODELO)
	21	257	1	0
	38	1740	1	0
	42	534	1	0
	57	1638	1	0
AR		21168	0	1
AR	6	10530	0	0
AR	9	4928	0	0
AR	15	5667	0	0
AR	20	43	0	0
BI		3741	0	1
BI	10	363	0	0
BI	38	1740	0	0
BI	57	1638	0	0
CL		16680	0	1
CL	12	7000	0	0
CL	15	3068	0	0
CL	18	6612	0	0
EM		791	0	1
EM	21	257	0	0
EM	42	534	0	0
PU		20102	0	1
PU	5	12420	0	0
PU	9	7682	0	0
TO		10303	0	1
TO	6	464	0	0
TO	9	756	0	0
TO	10	987	0	0
TO	12	7740	0	0
TO	16	356	0	0
TU		3064	0	1
TU	6	277	0	0
TU	9	876	0	0
TU	10	1023	0	0
TU	12	234	0	0
TU	16	654	0	0

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

GROUPING SETS

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

```
SELECT ...
...
```

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

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

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

De esa consulta se obtiene:

TI	MODELO	N_ALMACEN	SUM(CANTIDAD)
AR	6		10530
AR	9		4928
AR	15		5667
AR	20		43
BI	10		363
BI	38		1740
BI	57		1638
CL	12		7000
CL	15		3068
CL	18		6612
EM	21		257
EM	42		534
PU	5		12420
PU	9		7682
TO	6		464
TO	9		756
TO	10		987
TO	12		7740
TO	16		356
TU	6		277
TU	9		876
TU	10		1023
TU	12		234
TU	16		654
		1	30256
		2	40112
		3	5481

Se trata de dos consultas de totales unidades

conjuntos de agrupaciones combinadas

Se pueden combinar agrupaciones de diversas formas creando consultas como:

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

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

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