Unidad 5:

El lenguaje SQL

2º de ASI

Autor: Jorge Sánchez Asenjo
http://www.jorgesanchez.net
email: info@jorgesanchez.net

5.1) características y conceptos básicos

5.1.1) 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 direcrices 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 (*Standard Query Language*, lenguaje estándar de consulta) 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 avalado por los organismos ISO y ANSI (el primer estándar es del año 1982). Aún hoy sigue siendo uno de los estándares más importantes de la industria informática.

Los estándares más seguidos son los de los años 1992 y 1999 (el último estándar). Sobre estos dos estándares giran estos apuntes.

5.1.2) objetivos

SQL pretende ser un lenguaje que simula su escritura en lenguaje normal. De ahí que se le considere un lenguaje de cuarta generación. Consta de palabras especiales y de expresiones.

Se trata de un lenguaje que intenta agrupar todas las funciones que se le pueden pedir a una base de datos

5.2) Modos de utilización

5.2.1) ejecución directa. SQL interactivo

Las instrucciones SQL se introducen a través de una herramienta que las traduce inmediatamente a la base de datos, por lo que se ejecutan al instante.

5.2.2) ejecución incrustada o embebida

Las instrucciones SQL se colocan como parte del código de otro lenguaje anfitrión (C, Java, Pascal, Visual Basic,...). Estas instrucciones están separadas del resto del código de forma conveniente. Al compilar el código se utiliza un precompilador de la propia base de datos para traducir el SQL.

5.2.3) ejecución dinámica

Se trata de SQL incrustado en módulos especiales que pueden ser invocados una y otra vez desde distintas aplicaciones.

5.3) elementos del lenguaje SQL

5.3.1) código SQL

El código SQL consta de los siguientes elementos:

- **Comandos.** Las distintas instrucciones que se pueden realizar desde 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.**
- **Cláusulas.** Son palabras especiales que permiten modificar el funcionamiento de un comando **(WHERE, ORDER BY,...)**
- **Operadores.** Permiten crear expresiones complejas. Pueden ser aritméticos (+,-,*,/,...) lógicos (>, <, !=,<>, AND, OR,...)
- **Funciones**. Para conseguir valores complejos (SUM(), DATE(),...)
- **Constantes.** Valores literales para las consultas, números, textos, caracteres,...
- **Datos**. Obtenidos de la propia base de datos

5.3.2) 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
- Se pueden tabular líneas para facilitar la lectura si fuera necesario
- Los comentarios en el código SQL comienzan por /* y terminan por */

5.3.3) nota

En estos apuntes se hace referencia al SQL estándar y sobre todo al utilizado por Oracle.

5.4) DDL

5.4.1) introducción

El DDL es la parte del lenguaje que se encarga de la definición de datos. Fundamentalmente se encarga de la creación de esquemas, tablas y vistas. Por ahora veremos como crear tablas.

Cada usuario de una base de datos posee un **esquema**. El esquema suelte tener 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, índices y otras objetos relacionados con la definición de la base de datos. 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.

Hay que tener en cuenta que **ninguna instrucción DDL puede ser anulada por una instrucción ROLLBACK** (que se verá más adelante) por lo que hay que tener mucha precaución a la hora de utilizarlas.

5.4.2) creación de bases de datos

Esta es una tarea administrativa que se comentará más profundamente en otros temas. Por ahora sólo se comenta de forma simple. Crear la base de datos implica indicar los archivos y ubicaciones que se utilizarán para la misma, además de otras indicaciones técnicas.

Lógicamente sólo es posible crear una base de datos si se tienen privilegios DBA o en el caso de Oracle SYSDBA (*DataBase Administrator)*.

El comando SQL de creación de una base de datos es **CREATE DATABASE**. Este comando crea una base de datos con el nombre que se indique. Ejemplo:

```
CREATE DATABASE prueba;
```

Pero normalmente se indican más parámetros. Ejemplo:

```
CREATE DATABASE prueba
  LOGFILE prueba.log
  MAXLOGFILES 25
  MAXINSTANCES 10
  ARCHIVELOG
  CHARACTER SET WIN1214
  NATIONAL CHARACTER SET UTF8
  DATAFILE prueba1.dbf AUTOEXTEND ON MAXSIZE 500MB;
```

5.4.3) creación de tablas

nombre de las tablas

Deben cumplir las siguientes reglas (en algunos SGBD podrían cambiar):

- 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
- En el caso de que el nombre tenga espacios en blanco o caracteres nacionales (permitido sólo en algunas bases de datos), entonces se suele entrecomillar (en Oracle con comillas dobles, en SQLServer con comillas simples)

orden CREATE TABLE

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

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

Ejemplo:

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

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

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));
```

5.4.4) orden DESCRIBE

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

```
DESCRIBE proveedores;
```

Y aparecerán los campos de la tabla proveedores. Esta instrucción no es parte del SQL estándar, pero casi es considerada así ya que casi todos los SGBD la utilizan.

5.4.5) 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');
```

5.4.6) 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), en aquellas bases de datos que tengan la posibilidad de utilizar transacciones.
- Lógicamente sólo se pueden eliminar las tablas sobre las que tenemos permiso de borrado.

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

5.4.7) tipos de datos

A la hora de crear tablas, hay que indicar el tipo de datos de cada campo. Para ello describimos la siguiente tabla:

Descripción	Tipos ANSI SQL	SQL Server	Oracle SQL	Tipo MySQL
Texto de anchura fija	CHARACTER(***n***) CHAR(***n***)	CHAR(***n***)	CHAR(***n***)	CHAR(***n***)
Texto de anchura variable	CHARACTER VARYING(***n***) CHAR VARYING(***n***)	VARCHAR(***n***)	VARCHAR2(***n***)	VARCHAR(***n***)
Texto de anchura fija para	NATIONAL CHARACTER(***n***) NATIONAL	NCHAR(***n***)	NCHAR(***n***)	

Descripción	Tipos ANSI SQL	SQL Server	Oracle SQL	Tipo MySQL
caracteres nacionales	CHAR(***n***) NCHAR(***n***)			
Texto de anchura variable para caracteres nacionales	NATIONAL CHARACTER VARYING(***n***) NATIONAL CHAR VARYING(***n***) NCHAR VARYING(***n***)	NVARCHAR(***n***)	NVARCHAR2(***n***)	
Enteros	INTEGER INT SMALLINT	INT INT SMALL INT	NUMBER(***38***)	INT SMALL INT TINY INT
Decimal de coma variable	FLOAT(***b***) DOUBLE DOUBLE PRECISSION REAL	FLOAT	NUMBER	FLOAT(***m,d***) DOUBLE(***m,d***)
Decimal de coma fija	NUMERIC(***m,d***) DECIMAL(***m,d***)	NUMERIC(***m,d***) DECIMAL(***m,d***)	NUMBER(***m,d***)	 DECIMAL(***m,d***)
Fechas	DATE		DATE	DATE
Fecha y hora	TIMESTAMP	TIMESTAMP DATETIME SMALLDATETIME	TIMESTAMP	TIMESTAMP DATETIME TIME
Intervalos	INTERVAL		INTERVAL	YEAR
Lógicos	BIT	BINARY		BIT BOOL
Texto gran longitud	LOB	TEXT y NTEXT	LONG (en desuso) y CLOB	TEXT, MEDIUM TEXT y LONG TEXT
Binario de gran longitud	BLOB	IMAGE	RAW, LONG RAW BLOB	BLOB, MEDIUM BLOB y LONG BLOB

textos

Para los textos disponemos de los siguientes tipos:

- **VARCHAR.** Para textos de longitud variable. Su tamaño depende de la base de datos (en Oracle es de 4000)

- **CHAR.** Para textos de longitud fija (en Oracle hasta 2000 caracteres).
- **NCHAR.** Para el almacenamiento de caracteres nacionales de texto fijo
- **NVARCHAR.** Para el almacenamiento de caracteres nacionales de longitud variable.

En todos estos tipos se indican los tamaños entre paréntesis tras el nombre del tipo. Conviene poner suficiente espacio para almacenar los valores. En el caso de los VARCHAR, no se 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

Si son enteros se indican con INT (en Oracle es el tipo NUMBER seguido del tamaño) que equivale a los **long** del lenguaje C. Si son decimales se elige FLOAT o DOUBLE si se desea precisión doble (en Oracle es el tipo NUMBER a secas).

También existe la posibilidad de utilizar el tipo DECIMAL que representa decimales de longitud fija (gastan más espacio, pero son más precisos). En ese caso se indica el tamaño completo del número y el número de decimales. Por ejemplo, DECIMAL (8,3) indica que se representan números de ocho cifras de precisión y tres decimales. En Oracle se usa igual pero sustituyendo la palabra DECIMAL por la palabra NUMBER

tipos LONG y LOB

Se trata de la posibilidad de utilizar textos muy grandes (varias MB o incluso GB). Dependiendo de la base de datos son tipos LONG, LOB o CLOB. Pero el uso es el mismo.

tipos BLOB y RAW

Parecidos a los anteriores pero utilizados para almacenar datos binarios (imágenes, vídeo,...).

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 la mayoría de bases de datos proporcionan funciones (como SYSDATE en Oracle) que devuelven ese valor.

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

Sirven para almacenar intervalos de tiempo (no fechas, sino una suma de elementos de tiempo). En el caso de Oracle son:

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

5.4.8) 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.

modificar tablas

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

añadir columnas

```
ALTER TABLE  nombreTabla ADD(nombreColumna TipoDatos
[Propiedades][,columnaSiguiente 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 [,columnaSiguiente,...]);
```

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]
  [columnaSiguiente tipo [propiedades] ...]
```

Los cambios que se permiten son (en Oracle):

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

renombrar columna

Esto permite cambiar el nombre de una columna. Sintaxis

```
ALTER TABLE nombreTabla
RENAME COLUMN nombreAntiguo TO nombreNuevo
```

5.4.9) 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);
```

5.4.10) 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 (**P**RIMARY **K**EY).

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 la propia base de datos. La otra forma (que admite nombre) es:

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

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 permitírsenos borrar el registro ocurrirá fallo de integridad (habrá claves secundarios refiriéndose a una clave principal que ya no existe).

Por ello se nos pueden ofrecer 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.
- **ON DELETE SET DEFAULT.** Coloca en el registro relacionado el valor por defecto en la columna relacionada
- **ON DELETE NOTHING.** No hace nada.

En esas cuatro cláusulas se podría sustituir la palabra DELETE por la palabra UPDATE, haciendo que el funcionamiento se refiera a cuando se modifica un registro de la tabla principal.

En la base de datos Oracle sólo se permite utilizar ON DELETE SET NULL o ON DELETE CASCADE.

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}]
);
```

Ejemplo (no válido para Oracle, por el uso de ON UPDATE):

```
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)
                    ON DELETE SET NULL ON UPDATE CASCADE,
        CONSTRAINT pelicula_fk FOREIGN KEY (cod_pelicula)
              REFERENCES peliculas(cod)
                    ON DELETE CASCADE
);
```

restricciones de validación

Son restricciones que dictan una condición que deben cumplir los contenidos de una columna. 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_max<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 (en la base de datos Oracle):

```
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 restricción de 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

El trabajo con restricciones ya se ha visto que es complejo. Por eso todas las bases de datos suelen proporcionar una vista (o más) del diccionario de datos que permite consultar las restricciones. En el caso de Oracle, se puede utilizar la vista del diccionario de datos USER_CONSTRAINTS.

Esta vista 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**

Columna	Tipo de datos	Descripción
CONSTRAINT_NAME	VARCHAR2(30)	**Nombre de la restricción**
CONSTRAINT_TYPE	VARCHAR2(1)	**Tipo de restricción:** ◉ C. **De tipo CHECK o NOT NULL** ◉ P. **PRIMARY KEY** ◉ R. **FOREIGN KEY** ◉ 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 (resultado de la instrucción SELECT * FROM USER_CONS_COLUMNS):

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.

5.5) DQL

5.5.1) capacidades

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

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

5.5.2) sintaxis sencilla del comando SELECT

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

Donde:

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

Ejemplos:

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

relación con el álgebra relacional

El comando SELECT permite implementar de forma exacta todas las consultas del álgebra relación. En concreto la proyección se implementa así. Ejemplo:

$$\Pi_{\text{nombre, apellidos}}(\text{Cliente})$$

La proyección anterior en SQL se escribe como:

```
SELECT nombre, apellidos FROM Cliente;
```

El renombrado es lo que permite realizar la cláusula AS, sólo que en SQL se utiliza exclusivamente para abreviar las referencias a una tabla. Ejemplo:

```
SELECT c.nombre, c.apellidos, a.fecha
FROM Clientes AS c, ALQUILERES AS a
WHERE c.dni=a.dni;
```

La palabra AS se puede obviar:

```
SELECT c.nombre, c.apellidos, a.fecha
FROM Clientes c, ALQUILERES a
WHERE c.dni=a.dni;
```

5.5.3) 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 un nueva columna. Ejemplo:

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

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

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

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

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

concatenación de textos

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

```
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

5.5.4) 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;
```

relación con el álgebra relacional

La cláusula WHERE es la encargada de implementar la operación de selección del álgebra relacional. Ejemplo:

$$\sigma_{\text{nombre=”Pepe”} \wedge \text{edad>25}}(\text{Cliente})$$

Se corresponde con el SQL:

```
SELECT * FROM Clientes WHERE nombre='Pepe' AND edad>25
```

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**

Operador	Significado
=	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.

Ejemplos:

```
/* 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. Se puede usar la expresión IS NOT NULL que devuelve verdadero cuando una determinada columna no es nula

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 (tomada de Oracle):

Orden de precedencia	Operador
1	*(Multiplicar) / (dividir)
2	+ (Suma) - (Resta)

Orden de precedencia	Operador
3	|| (Concatenación)
4	Comparaciones (>, <, !=, ...)
5	IS [NOT] NULL, [NOT]LIKE, IN
6	NOT
7	AND
8	OR

5.5.5) 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;
```

5.5.6) funciones

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

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.

En realidad hay dos tipos de funciones:

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

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

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

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

Muestra una tabla con el contenido de ese cálculo. DUAL es una tabla interesante para hacer pruebas.

funciones de caracteres

conversión del texto a mayúsculas y minúsculas

Son:

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

funciones de transformación

Función	Descripción
RTRIM(*texti*)	Elimina los espaciosa la derecha del texto
LTRIM(*texto*)	Elimina los espacios a la izquierda que posea el texto
TRIM(*texto*)	Elimina los espacios en blanco a la izquierda y la derecha del texto y los espacios dobles del interior.
TRIM(*caracteres* FROM *texto*)	Elimina del texto los caracteres indicados. Por ejemplo TRIM('h' FROM nombre) elimina las haches de la columna *nombre* que estén a la izquierda y a la derecha
SUBSTR(*texto*,*n*[,*m*])	Obtiene los *m* siguientes caracteres del texto a partir de la posición *n* (si *m* no se indica se cogen desde *n* hasta el final).

Función	Descripción
LENGTH(*texto*)	Obtiene el tamaño del texto
INSTR(*texto, textoBuscado* [,*posInicial* [, *nAparición*]])	Obtiene la posición en la que se encuentra el texto buscado en el texto inicial. Se puede empezar a buscar a partir de una posición inicial concreta e incluso indicar el número de aparición del texto buscado. Ejemplo, si buscamos la letra *a* y ponemos 2 en *nAparición*, devuelve la posición de la segunda letra a del texto). Si no lo encuentra devuelve 0
REPLACE(*texto, textoABuscar, textoReemplazo*)	Buscar el texto a buscar en un determinado texto y lo cambia por el indicado como texto de reemplazo

funciones numéricas

redondeos

Función	Descripción
ROUND(*n,decimales*)	Redondea el número al siguiente número con el número de decimales indicado más cercano. ROUND(8.239,2) devuelve 8.3
TRUNC(*n,decimales*)	Los decimales del número se cortan para que sólo aparezca el número de decimales indicado

matemáticas

Función	Descripción
MOD(*n1,n2*)	Devuelve el resto resultado de dividir *n1* entre *n2*
POWER(*valor,exponente*)	Eleva el valor al exponente indicado
SQRT(*n*)	Calcula la raíz cuadrada de n
SIGN(*n*)	Devuelve 1 si *n* es positivo, cero si vale cero y -1 si es negativo
ABS(*n*)	Calcula el valor absoluto de *n*
EXP(*n*)	Calcula e^n, es decir el exponente en base *e* del número n
LN(*n*)	Logaritmo neperiano de *n*
LOG(*n*)	Logaritmo en base 10 de *n*

Función	Descripción
SIN(*n*)	Calcula el seno de *n* (*n* tiene que estar en radianes)
COS(*n*)	Calcula el coseno de *n* (*n* tiene que estar en radianes)
TAN(*n*)	Calcula la tangente de *n* (*n* tiene que estar en radianes)
ACOS(*n*)	Devuelve en radianes el arcocoseno de *n*
ASIN(*n*)	Devuelve en radianes el arcoseno de *n*
ATAN(*n*)	Devuelve en radianes el arcotangente de *n*
SINH(*n*)	Devuelve el seno hiperbólico de *n*
COSH(*n*)	Devuelve el coseno hiperbólico de *n*
TANH(*n*)	Devuelve la tangente hiperbólica de *n*

funciones de trabajo con nulos

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

Función	Descripción
NVL(*valor,sustituto*)	Si el *valor* es NULL, devuelve el valor *sustituto*; de otro modo, devuelve valor
NVL2(*valor,sustituto1, sustituto2*)	Variante de la anterior, devuelve el valor *sustituto1* si *valor* no es nulo. Si *valor* es nulo devuelve el *sustituto2*

funciones de fecha y manejo de fechas

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

calcular fechas

Función	Descripción
ADDMONTHS(*fecha,n*)	Añade a la fecha el número de meses indicado por *n*
MONTHS_BETWEEN(*fecha1, fecha2*)	Obtiene la diferencia en meses entre las dos fechas (puede ser decimal)
NEXT_DAY(*fecha,día*)	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(*fecha*)	Obtiene el último día del mes al que pertenece la fecha. Devuelve un valor DATE
EXTRACT(*valor* FROM *fecha*)	Extrae un valor de una fecha concreta. El valor puede ser *day* (día), *month* (mes), *year* (año), etc.
GREATEST(*fecha1, fecha2,..*)	Devuelve la fecha más moderna la lista
LEAST(*fecha1, fecha2,..*)	Devuelve la fecha más antigua la lista
ROUND(*fecha* [,'*formato*']	Redondea la fecha al valor de aplicar el formato a la fecha. El formato puede ser: ◉ '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' Redondea al día más cercano
TRUNC(*fecha* [*formato*])	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 querremos realizar conversiones explícitas.

TO_CHAR

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

fechas

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

Símbolo	Significado
YY	Año en formato de dos cifras
YYYY	Año en formato de cuatro cifras
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
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, donde se pongan estos símbolos aparecerán en el resultado

Ejemplos:

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

números

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

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

TO_NUMBER

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

TO_DATE

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

5.5.7) 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.

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

EMPLEADOS	
Empleado	**Sueldo**
Antonio	**18000**
Marta	**21000**
Sonia	**15000**

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

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

La forma sería:

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

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, tabl1.columna2,...
    tabla2.columna1, tabla2.columna2,... FROM tabla1
[CROSS JOIN tabla2]|
[NATURAL JOIN tabla2]|
[JOIN tabla2 USING(columna)]|
[JOIN tabla2 ON (tabla1.columa=tabla2.columna)]|
[LEFT|RIGHT|FULL OUTER JOIN tabla2 ON
(tabla1.columa=tabla2.columna)]
```

Se describen sus posibilidades

CROSS JOIN

Utilizando la opción CROSS JOIN se realiza un producto cruzado entre las tablas indicadas. Esta orden es equivalente al producto del álgebra relacional. Ejemplo:

- **álgebra relacional:** $\mathbf{R} \times \mathbf{S}$
- **SQL.** SELECT * FROM R CROSS JOIN S;

NATURAL JOIN

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

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

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

Equivalencia con el álgebra relacional:

- **álgebra relacional:** $R \infty S$
- **SQL:** SELECT * FROM R NATURAL JOIN S;

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);
```

Equivalencia con el álgebra relacional:

- **álgebra relacional:**

  $$\text{cliente} \underset{\text{dni}}{\infty} \text{alquiler}$$

- **SQL.** SELECT * FROM clientes JOIN alquiler USING(dni);

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);
```

Equivalencia con el álgebra relacional:

- **álgebra relacional:**

  $$localidades \infty provincias$$
  provincias.id_capital=localidades.id_localidad

- **SQL.** SELECT * FROM localidades JOIN provincias
  USING(provincias.id_capital=localidades.id_capital);

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 este consulta además de las relacionadas, aparecen los datos de los registros de la tabla piezas que no están en existencias. Si el LEFT lo cambiamos por un RIGHT, aparecerán las existencias no presentes en la tabla piezas (además de las relacionadas en ambas tablas).

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

5.5.8) 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(*expresión*)	Suma los valores de la expresión
AVG(*expresión*)	Calcula la media aritmética sobre la expresión indicada
MIN(*expresión*)	Mínimo valor que toma la expresión indicada
MAX(*expresión*)	Máximo valor que toma la expresión indicada
STDDEV(*expresión*)	Calcula la desviación estándar
VARIANCE(*expresión*)	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.

5.5.9) 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 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.

5.5.10) 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*/
```

5.6) DML

5.6.1) 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.

5.6.2) 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 9).

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;
```

5.6.3) 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.

5.6.4) 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 15, clave secundaria o foránea) hace que no sólo se borren los registros indicados en el SELECT, sino todos los relacionados.

5.6.5) 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.

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.

5.7) vistas

5.7.1) introducción

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

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.

5.7.2) creación de vistas

Sintaxis:

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

- **OR REPLACE.** Si la vista ya existía, la cambia por la actual

- **FORCE.** Crea la vista aunque los datos de la consulta SELECT no existan
- *vista*. Nombre que se le da a la vista
- *alias*. Lista de alias que se establecen para las columnas devueltas por la consulta SELECT en la que se basa esta vista. El número de alias debe coincidir con el número de columnas devueltas por SELECT.
- **WITH CHECK OPTION.** Hace que sólo las filas que se muestran en la vista puedan ser añadidas (INSERT) o modificadas (IPDATE). 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.

5.7.3) 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)
```

5.7.4) 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).

5.7.5) borrar vistas

Se utiliza el comando DROP VIEW:

```
DROP VIEW nombreDeVista;
```