



EFA
MORATALAZ

*1º CFGS Desarrollo de
Aplicaciones Web*

BASES DE DATOS

JESÚS SANTIAGO RICO

UT4 – MANIPULACIÓN DE BBDD CON SQL



ORACLE



EFA
MORATALAZ

*1º CFGS Desarrollo de Aplicaciones
web*

BASES DE DATOS

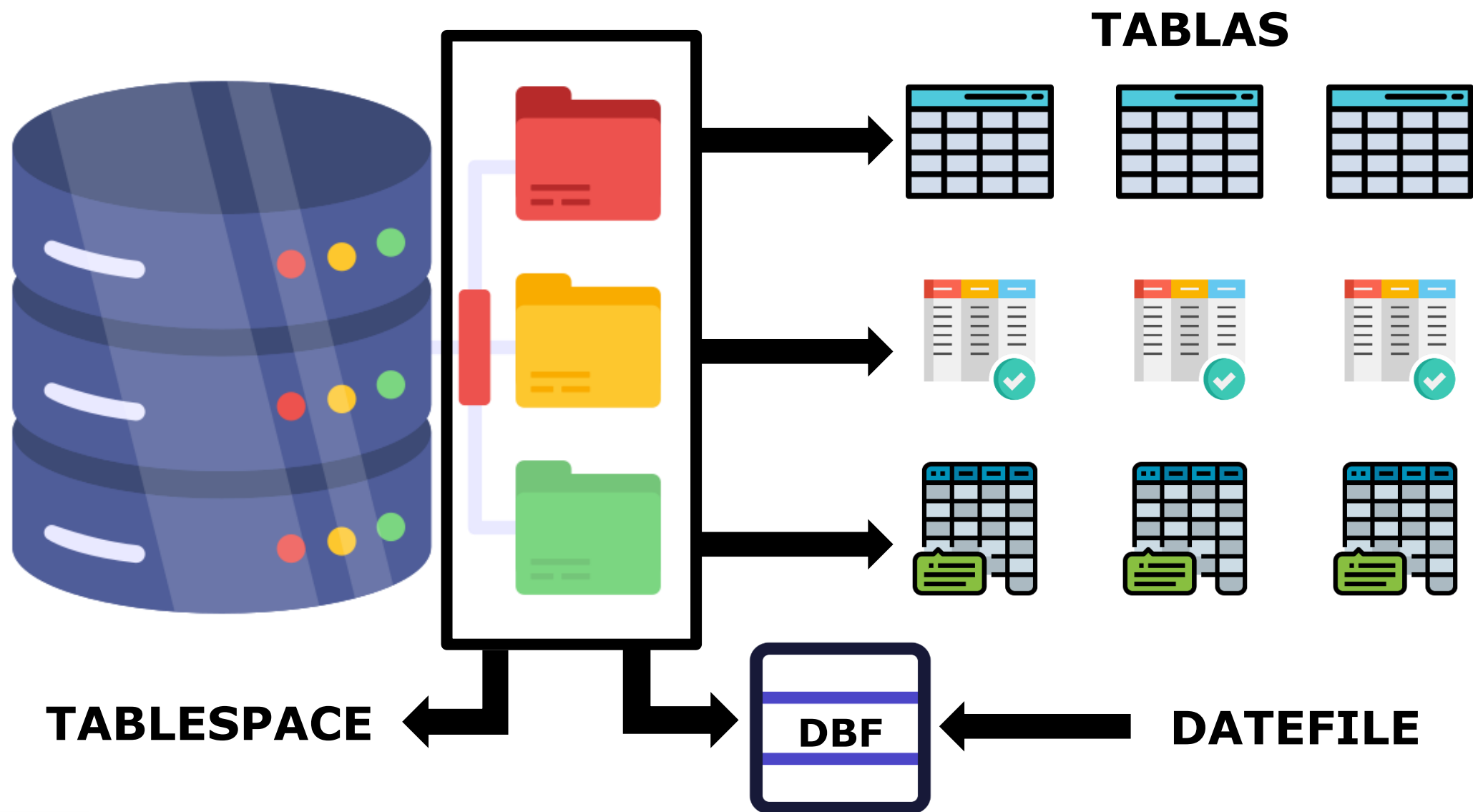
UT4 – MANIPULACIÓN DE BBDD CON SQL

1. INTRODUCCIÓN AL LENGUAJE SQL
2. SENTENCIAS DDL
3. SENTENCIAS DML
4. FUNCIONES DE ORACLE
5. VISTAS
6. ÍNDICES
7. OTRAS FUNCIONALIDADES

INTRODUCCIÓN AL LENGUAJE SQL

1

- El lenguaje **SQL** (Structured Query Language) permite la comunicación con el **SGBD**.
- Actualmente es el lenguaje estándar para la gestión de Bases de Datos relacionales para ANSI (American National Standard Institute) e ISO (International Standardization Organization).
- Entre las principales características de este lenguaje destaca que es **un lenguaje para todo tipo de usuarios** de manera que permite realizar cualquier consulta de datos por muy complicada que parezca.
- En el lenguaje SQL, dependiendo de las tareas que se quieran realizar, se distinguen dos tipos de sentencias: **sentencias DDL** (Data Definition Language) que sirven para especificar y gestionar estructuras de datos, y las **sentencias DML** (Data Manipulation Language) que sirven para trabajar con los datos almacenados en dichas estructuras.



SENTENCIAS DDL



- La sintaxis para la creación de una tabla es la siguiente:

```
CREATE [GLOBAL TEMPORARY] TABLE [usuario/esquema] <nombreTabla> (  
    columna TIPO_DATO [DEFAULT expresión] [restricción columna] | restricción  
tabla,  
    ....  
    [CONSTRAINT nombreRestricción [, ...,]]  
)  
[TABLESPACE espacioTabla]
```

```
[COMMENT ON TABLE <nombreTabla> IS 'Comentario de tabla';]
```

```
[COMMENT ON COLUMN <nombreTabla>.<campo> IS 'Comentario de columna';]
```

- **Tipos de datos**

TIPO DE DATO	DESCRIPCIÓN
CHAR(tamaño)	Almacena datos de tipo carácter de longitud fija, máximo 2000 caracteres.
VARCHAR2(tamaño)	Almacena datos de tipo carácter de longitud variable, máximo 4000 caracteres
VARCHAR	Actualmente es igual que char.
LONG	Almacenan datos de tipo carácter de longitud variable, hasta 2 GB
BLOB	Es un objeto binario de gran tamaño máximo 4GB. Imágenes, datos de voz
DATE / TIMESTAMP	Almacena fechas desde 1 de enero del 4712 a.C hasta el 31 de diciembre 4712 d.C
INTEGER	Un numero entero que no tiene parte decimal. Rango de 32 bits
SMALLINT	Un numero entero que no tiene parte decimal. Comprendido -32768 y 32767.
NUMBER(l, d) / NUMBER	Almacena datos de tipo numérico, siendo “l” la longitud y “d” el número de dígitos decimales

- La sintaxis para la creación de una restricción es la siguiente:

CONSTRAINT <nombreRestricción> columna **RESTRICCIÓN**;

RESTRICCIÓN	TIPO	DESCRIPCIÓN
NOT NULL	Columna	La columna no permitirá valores nulos
DEFAULT	Columna	La columna tendrá un valor por defecto
PRIMARY KEY	Columna	Permite indicar que la columna es clave primaria
REFERENCES	Columna	Indica que este campo es clave ajena y apunta a la clave candidata de otra tabla (sólo un campo)
UNIQUE	Columna	Obliga a que la columna tenga valores únicos. Se implementa creando un índice para esa columna.
CHECK (condicion)	Columna	Permite indicar una condición que se debe cumplir.

- La sintaxis para la creación de una restricción es la siguiente:

CONSTRAINT <nombreRestricción> columna **RESTRICCIÓN**;

RESTRICCIÓN	TIPO	DESCRIPCIÓN
PRIMARY KEY	Tabla	Permite indicar que la columna es clave primaria
FOREIGN KEY	Tabla	Indica que este campo es clave ajena y apunta a la clave candidata de otra tabla.
UNIQUE	Tabla	Obliga a que la columna tenga valores únicos.
CHECK (condición)	Tabla	Permite indicar una condición que se debe cumplir.

- Clave primaria/PRIMARY KEY

- Una clave primaria dentro de una tabla es una columna o conjunto de columnas cuyo valor **identifica unívocamente a cada fila**.
- Debe ser única, no nula, es obligatoria y como máximo podremos definir una clave primaria por tabla.

```
CREATE TABLE FAMILIA (  
  ID_FAMILIA NUMBER PRIMARY KEY,  
  DESCRIPCION VARCHAR2(100)  
);
```



NIVEL DE COLUMNA

NIVEL DE TABLA



```
CREATE TABLE FAMILIA (  
  ID_FAMILIA NUMBER,  
  DESCRIPCION VARCHAR2(100),  
  CONSTRAINT PK_FAMILIA PRIMARY KEY (ID_FAMILIA)  
);
```

- **Clave ajena/FOREIGN KEY**

- Una clave ajena está formada por una o varias columnas que hacen referencia a una clave primaria de otra o de la misma tabla.
- Se pueden definir tantas claves ajenas como sea necesario (no hay límite). El valor de la columna o columnas que son clave ajena será NULL, o bien el valor de la clave primaria de la tabla a la que hacen referencia (integridad referencial).
- A la hora de definir una clave ajena, deberemos indicar con la cláusula **REFERENCES** la tabla a que ésta hace referencia.
- Habrá que tener en cuenta que mientras que un campo definido como clave ajena haga referencia a un campo definido como clave primaria, **la columna de la segunda tabla no podrá ser eliminada hasta que no lo haga la columna que le hace referencia** (integridad referencial).
- La columna deberá ser del mismo tipo que la columna de la que es clave ajena.

- Clave ajena/FOREIGN KEY

```
CREATE TABLE TIPO (  
    ID_TIPO NUMBER,  
    DESCRIPCION VARCHAR2(200),  
    ID_FAMILIA NUMBER REFERENCES FAMILIA (ID_FAMILIA)  
);
```



NIVEL DE COLUMNA

```
CREATE TABLE TIPO (  
    ID_TIPO NUMBER,  
    DESCRIPCION VARCHAR2(200),  
    ID_FAMILIA NUMBER,  
    CONSTRAINT FK_TIPO FOREIGN KEY (ID_FAMILIA) REFERENCES FAMILIA (ID_FAMILIA)  
);
```



NIVEL DE TABLA

- **Clave ajena/FOREIGN KEY**

- Las claves foráneas tienen opciones tanto para el borrado como para la actualización.
- **ON DELETE / ON UPDATE:**
 - **CASCADE:** Al borrar/modificar una de las filas de la tabla a la que se referencia, el valor de la tabla no referenciadora se actualiza también.
 - **SET NULL:** Al borrar/modificar una de las filas de la tabla a la que se referencia, el valor de la tabla no referenciadora, se pone a nulo.
 - **SET DEFAULT:** Al borrar/modificar una de las filas de la tabla a la que se referencia, el valor de la tabla no referenciadora, se pone un valor por defecto.
 - **NO ACTION:** Al borrar/modificar una de las filas de la tabla a la que se referencia, el valor de la tabla no referenciadora, no se hace nada.

- Clave ajena/FOREIGN KEY

```
CREATE TABLE TIPO (  
  ID_TIPO NUMBER,  
  DESCRIPCION VARCHAR2(200),  
  ID_FAMILIA NUMBER,  
  CONSTRAINT FK_TIPO FOREIGN KEY (ID_FAMILIA) REFERENCES FAMILIA (ID_FAMILIA)  
  ON DELETE SET NULL  
  ON UPDATE CASCADE  
);
```



NIVEL DE TABLA

- Valor por defecto/DEFAULT

- Se puede definir el valor que una columna tomará por defecto, es decir, si al introducir una fila no se especifica valor para dicha columna. Se utiliza la palabra reservada **DEFAULT**.
- El valor por defecto debe corresponder con el tipo de dato definido:

```
CREATE TABLE FAMILIA (  
  ID_FAMILIA NUMBER DEFAULT 999,  
  DESCRIPCION VARCHAR2(100)  
);
```



NIVEL DE COLUMNA

- Valores únicos/UNIQUE

- La restricción **UNIQUE** evita valores repetidos en una misma columna.
- Al contrario que ocurre con la restricción PRIMARY KEY, la restricción de valor único se puede aplicar a varias columnas de una misma tabla y admite el valor NULL. Con respecto a esta última consideración, conviene saber que, si una columna se define como UNIQUE, sólo una de sus filas podrá contener el valor NULL.

```
CREATE TABLE ROLES(  
  ID_ROL VARCHAR2(20),  
  DESCRIPCION_ROL VARCHAR2(150) UNIQUE  
);
```



NIVEL DE COLUMNA

NIVEL DE TABLA



```
CREATE TABLE ROLES(  
  ID_ROL VARCHAR2(20),  
  DESCRIPCION_ROL VARCHAR2(150)  
  CONSTRAINT PK_UNIQUE UNIQUE(DESCRIPCION_ROL)  
);
```

- Condiciones/CHECK

- De forma más genérica, podemos forzar a que los valores de determinados campos de la tabla **cumplan unas ciertas condiciones**. En caso contrario no se permitirá la inserción de esa fila en dicha tabla.

```
CREATE TABLE PRODUCTOS(  
  ID_PRODUCTO NUMBER,  
  DESCRIPCION_PRODUCTO VARCHAR2(200),  
  ID_TIPO NUMBER,  
  PRECIO NUMBER CHECK (PRECIO > 100)  
);
```



NIVEL DE COLUMNA

NIVEL DE TABLA



```
CREATE TABLE PRODUCTOS(  
  ID_PRODUCTO NUMBER,  
  DESCRIPCION_PRODUCTO VARCHAR2(200),  
  ID_TIPO NUMBER,  
  PRECIO NUMBER,  
  CONSTRAINT CHECK_PRO CHECK (PRECIO > 100)  
);
```

- **Campos obligatorios**

- Esta restricción obliga a que se le tenga que dar valor obligatoriamente a una columna. Por tanto, no podrá tener el valor NULL. Se utiliza la palabra reservada **NOT NULL**.

- Eliminar un objeto: DROP
- Es la sentencia utilizada para eliminar objetos (tabla, usuario, vista, procedimiento,..) en una Base de Datos.
- La sintaxis para la **eliminación de tablas** es la siguiente:

DROP TABLE <nombreTabla>;

- Y para renombrar una tabla usaremos:

RENAME TABLE <nombreTablaViejo> **TO** <nombreTablaNuevo>

- Modificar un objeto: **ALTER**
- Es la sentencia utilizada para modificar objetos (tabla, usuario, vista, procedimiento, ...) en una Base de Datos.
- La sintaxis para modificar una tabla es la siguiente:

- Modificar un objeto: **ALTER**

ALTER TABLE <nombreTabla>

[**ADD** <nombreColumna> <tipoDato> <restricción>]

[**MODIFY** <nombreColumna> <tipoDato>]

[**DROP COLUMN** <nombreColumna>]

[**ADD CONSTRAINT** <nombreRestricción> <restricción>]

[**DROP CONSTRAINT** <nombreRestricción>]

[**DROP PRIMARY KEY**]

[**RENAME COLUMN** <nombreColumnaExistente> **TO** <nombreColumnaNuevo>];

- **Comentarios en lenguaje SQL**

- Existe la posibilidad de añadir comentarios al código del lenguaje SQL según la siguiente sintaxis

```
-- Esto es un comentario y Oracle no lo ejecuta
```

```
/**
```

```
    Esto también es una comentario y tampoco se  
    ejecuta
```

```
**/
```

- Mostrar la información almacenada por Oracle:
 - DESCRIBE nombre_tabla; → Muestra la información/atributos de una tabla.
- Creación de sinónimos:
CREATE SYNONYM <nombreSinonimo> **FOR** <nombreTabla>;
- Borrado de sinónimos:
DROP SYNONYM <nombreSinonimo>;

SENTENCIAS DML



- En el lenguaje SQL existen **4 sentencias que forman el DML** (Data Manipulation Language, Lenguaje de Manipulación de Datos). Son aquellas sentencias que nos permiten manipular la información que almacenamos en las Bases de Datos.
- Existen 4 instrucciones que nos permitirán:
 - Insertar datos en una tabla (**INSERT**).
 - Modificar esos datos (**UPDATE**).
 - Eliminarlos (**DELETE**).
 - Consultarlos (**SELECT**).

- **Inserción de registros**
- La inserción de nuevos registros a una tabla se efectúa con la sentencia **INSERT**, que tiene el siguiente formato:

Opción1:

INSERT INTO <nombreTabla> (columna1, columna2, ...) **VALUES**
(valorColumna1, valorColumna2, ...);

Opción 2:

INSERT INTO <nombreTabla> **VALUES** (valorColumna1, valorColumna2, ...);

- **Inserción de registros**
- Veamos algunos ejemplos:
 - Para insertar una fila en una tabla

```
INSERT INTO usuario (dni, nombre, apellidos, email, ciudad,  
fecha_nacimiento, descuento, fecha_alta)
```

```
VALUES ('123456789A', 'Antonio', 'García', 'agarcia@gmail.com',  
'Zaragoza', '1990-12-12', 0.3, '2003-02-01');
```

- **Modificación de registros**
- La modificación de registros ya insertados en la tabla se realiza con la sentencia **UPDATE**, que tiene el siguiente formato:

```
UPDATE nombre_tabla  
  
SET columna1 = valor1, columna2 = valor2, ...  
  
[ WHERE condiciones ]
```

- **Modificación de registros**
- Veamos algunos ejemplos:
 - Actualizar una columna de una fila

```
UPDATE usuario  
  
SET nombre = 'Felipe'  
  
WHERE id_usuario = 12;
```

- **Modificación de registros**
- Veamos algunos ejemplos:
 - Actualizar varias columnas de una fila

```
UPDATE usuario  
  
SET nombre = 'Felipe', dni = '123654789H'  
  
WHERE id_usuario = 15;
```

- **Modificación de registros**
- Veamos algunos ejemplos:
 - Actualizar una columna de varias filas

```
UPDATE pista  
  
SET precio = precio + precio * 0.10  
  
WHERE precio < 20 AND tipo = 'tenis';
```


- **Eliminación de registros**
- El borrado de filas de una tabla se efectúa con la sentencia **DELETE**, que tiene el siguiente formato:

```
DELETE FROM nombre_tabla  
  
[WHERE condiciones ]
```

- **Eliminación de registros**
- Veamos algunos ejemplos:
 - Elimina todos los usuarios

```
DELETE FROM usuario;
```



- Borrar una fila estableciendo una condición

```
DELETE FROM pista  
  
WHERE id_pista = 10;
```

- **COMMIT y ROLLBACK**
 - Oracle es una base de datos transaccional, por lo tanto, cuando realizamos una operación INSERT/UPDATE/DELETE esta no se hace en la base de datos de manera física, sino que se realiza en memoria hasta que la confirmamos o deshacemos la operación.
 - **COMMIT:** Confirma la transición
 - **ROLLBACK:** Deshace la transición.

- **Consulta de registros**
- La sentencia **SELECT**: La consulta de registros es la operación más compleja y también la más ejecutada de una Base de Datos.

SELECT [**ALL** | **DISTINCT** | **UNIQUE**] listaSeleccion

FROM nombreTablas

[**WHERE** condiciones]

[**GROUP BY** listaColumnas]

[**HAVING** condicion]

[**ORDER BY** nombreColumnas [**ASC** | **DESC**]]

- **Consulta de registros**

- **SELECT . . . FROM . . .**

- La cláusula **SELECT** se utiliza para seleccionar las **columnas** que se quieren visualizar como resultado de la consulta.
 - Se puede seleccionar cualquier columna de las tablas afectadas por la consulta (cláusula **FROM**), valores constantes establecidos a la hora de ejecutar la consulta, o bien el comodín '*' para indicar que se quieren visualizar todas las columnas afectadas.
 - La cláusula **FROM** permite indicar con qué tablas se trabajará en la consulta.
 - **No siempre serán tablas de las que se visualicen columnas, puesto que muchas veces sólo se utilizarán para relacionar unas tablas con otras.**
 - En cualquier caso, se usen para que se visualicen sus campos o bien para relacionar otras tablas (que no están directamente relacionadas), se deben indicar en esta cláusula.

- **Consulta de registros**
 - **SELECT ... FROM ...**
 - Consulta de una columna de una tabla

```
-- Nombre de todos los usuario (incluye repeticiones)  
  
SELECT nombre  
  
FROM usuario;
```

- **Consulta de registros**
 - **SELECT ... FROM ...**
 - Consulta de dos columnas de una tabla

```
-- Nombre y apellidos de todos los usuario
```

```
SELECT nombre, apellidos
```

```
FROM usuario;
```

- **Consulta de registros**
 - **SELECT ... FROM ...**
 - Consulta de todas las columnas de una tabla

-- Toda La información de todos Los usuario

SELECT *

FROM usuario;

- **Consulta de registros**

- **SELECT . . . FROM . . . WHERE . . .**
- La cláusula **WHERE** permite establecer condiciones sobre que filas se mostrarán en una sentencia de consulta.
- En **ausencia** de esta cláusula se **muestran todos los registros** de la tabla (aunque sólo las columnas establecidas en la cláusula SELECT).
- Si se indican condiciones mediante la cláusula **WHERE** sólo se mostrarán aquellas filas que las cumplan la o las condiciones.

- Consulta de registros
 - `SELECT ... FROM ... WHERE ...`

-- Nombre y dirección de los polideportivo de Zaragoza

```
SELECT nombre, direccion
```

```
FROM polideportivo
```

```
WHERE ciudad = 'Zaragoza';
```

- **Consulta de registros**

- **SELECT ... FROM ... WHERE ...**
- Además, nos permitirá establecer condiciones para establecer lo que se conoce como un **INNER JOIN** (implícito) entre dos o más tablas:

```
-- Código y tipo de las pistas de los polideportivos de Zaragoza  
  
SELECT pista.codigo, pista.tipo  
  
FROM pista, polideportivo  
  
WHERE pista.id_polideportivo = polideportivo.id_polideportivo  
  
AND polideportivo.ciudad = 'Zaragoza'
```

- **Consulta de registros**

- **SELECT ... FROM ... WHERE ...**

- Si utilizamos **alias** para los nombres de las tablas, podemos escribir la misma consulta algo más rápido:

```
-- Código y tipo de las pistas de los polideportivos de Zaragoza
```

```
SELECT P.codigo, P.tipo
```

```
FROM pista P, polideportivo PP
```

```
WHERE P.id_polideportivo = PP.id_polideportivo
```

```
AND PP.ciudad = 'Zaragoza'
```

- **Consulta de registros**
 - **GROUP BY / HAVING**
 - Las cláusulas **GROUP BY** y **HAVING** permiten crear agrupaciones de datos y establecer condiciones sobre dichas agrupaciones, respectivamente:

```
-- Número de polideportivos hay en cada ciudad  
  
SELECT ciudad, COUNT(*) AS cantidad  
  
FROM polideportivo  
  
GROUP BY ciudad;
```

- **Consulta de registros**
 - **GROUP BY / HAVING**
 - Las cláusulas **GROUP BY** y **HAVING** permiten crear agrupaciones de datos y establecer condiciones sobre dichas agrupaciones, respectivamente:

```
-- Número de polideportivos que hay en cada ciudad, solamente de aquellas  
  
-- ciudades donde hay más de 10.000  
  
SELECT ciudad, COUNT(*) AS cantidad  
  
FROM polideportivo  
  
GROUP BY ciudad  
  
HAVING COUNT(*) > 10000;
```

- **Consulta de registros**
 - **GROUP BY / HAVING**
 - También es posible añadir cláusulas **WHERE** para filtrar registros y agrupar el resultado final.

```
-- Número de pistas que hay de cada tipo en el polideportivo 'ACTUR 1'  
  
SELECT P.tipo, COUNT(*) AS numero_pista  
  
FROM pista P, polideportivo PP  
  
WHERE P.id_polideportivo = PP.id_polideportivo AND PP.nombre = 'ACTUR 1'  
  
GROUP BY P.tipo;
```

- **Consulta de registros**

- **OPERADORES**

- A la hora de **establecer condiciones** en una sentencia de consulta, podremos utilizar los siguientes operadores:
 - = (Igual)
 - < (Menor)
 - > (Mayor)
 - <= (Menor o igual)
 - >= (Mayor o igual)
 - <> (Distinto) !=
 - **NOT** (Operador lógico para la negación de condiciones)
 - **AND** (Operador lógico para la conjunción de condiciones)
 - **OR** (Operador lógico para la disyunción de condiciones)
 - **DISTINCT** (Se utiliza para indicar a la cláusula SELECT que no se muestren valor de columnas repetidos)

- **Consulta de registros**

- **OPERADORES**

- A la hora de **establecer condiciones** en una sentencia de consulta, podremos utilizar los siguientes operadores:

- **LIKE:** Permite comprobar si una cadena de caracteres cumple algún patrón determinado. Permite la expresión de patrones a través de dos caracteres comodín:
 - El carácter '_' para expresar un único carácter.
 - El carácter '%' para expresar cualquier secuencia de caracteres o incluso la secuencia vacía.

- Consulta de registros
 - OPERADORES

-- Nombre de Los polideportivos que están en una ciudad

-- cuyo nombre empieza por Z y tiene 8 caracteres

SELECT nombre

FROM polideportivo

WHERE ciudad **LIKE** 'Z_____';

-- Nombre de Los polideportivos que están en una ciudad

-- cuyo nombre empieza por Z

SELECT nombre

FROM polideportivo

WHERE ciudad **LIKE** 'Z%';

- **Consulta de registros**
 - **OPERADORES**
 - **IN | NOT IN:** Permite comprobar si un valor coincide (o no) con algún valor especificado como un conjunto.

```
-- Nombre y dirección de los polideportivos de Zaragoza, Huesca y Teruel  
  
SELECT nombre, direccion  
  
FROM polideportivo  
  
WHERE ciudad IN ('Zaragoza', 'Huesca', 'Teruel');
```

- **Consulta de registros**
 - **OPERADORES**
 - **IS NULL | IS NOT NULL:** Se utiliza para comprobar si un valor es igual (o no) a NULL.

```
-- Nombre y apellidos de los usuarios que no indicaron su fecha de nacimiento  
  
SELECT nombre, apellidos  
  
FROM usuario  
  
WHERE fecha_nacimiento IS NULL;
```

- **Consulta de registros**

- **OPERADORES**

- **BETWEEN:** Permite comprobar si el valor de una columna está comprendido entre dos valores determinados:

```
-- Nombre y apellidos de los usuarios que tienen un descuento  
  
-- entre 10 y 20 %  
  
SELECT nombre, apellidos  
  
FROM usuario  
  
WHERE descuento BETWEEN 0.1 AND 0.2;
```

- **Consulta de registros**

- **FUNCIONES AGREGADAS**

- Son funciones que proporciona el lenguaje SQL que permiten realizar operaciones sobre los datos de una base de datos:
 - **COUNT**: Devuelve el número de filas seleccionadas.

```
-- Número de pistas
```

```
SELECT COUNT(*)
```

```
FROM pista;
```

```
-- Número de polideportivos en Zaragoza
```

```
SELECT COUNT(*)
```

```
FROM polideportivo
```

```
WHERE ciudad = 'Zaragoza';
```

- **Consulta de registros**
 - **FUNCIONES AGREGADAS**
 - **SUM:** Devuelve la suma de todos los valores de una columna

```
-- Cuánto dinero costaría alquilar todas las pistas del  
-- polideportivo cuyo id es 23  
  
SELECT SUM(precio)  
  
FROM pista  
  
WHERE id_polideportivo = 23;
```

- **Consulta de registros**

- **FUNCIONES AGREGADAS**

- **MIN:** Devuelve el valor mínimo de una columna.

```
-- Cuánto vale la pista más barata  
  
SELECT MIN(precio)  
  
FROM pista;
```

- **MAX:** Devuelve el valor máximo de una columna.

```
-- Cuánto vale la pista más cara  
  
SELECT MAX(precio)  
  
FROM pista;
```


- **Consulta de registros**

- **FUNCIONES AGREGADAS**

- **AVG:** Devuelve el valor medio de los valores de una columna.

```
-- Valor medio de las pistas  
  
SELECT AVG(precio)  
  
FROM pista;
```

- Hay que tener en cuenta que, excepto la función COUNT, todas las demás devolverán el valor **NULL** si no hay columnas sobre las que puedan operar. La función COUNT, sin embargo, devolverá el valor 0 en ese caso.

- **Consulta de registros**

- **SUBCONSULTAS**

- La creación de subconsultas permite utilizar el resultado de una consulta como valor de entrada para la condición de otra consulta principal.

```
-- Código y tipo de la pista más barata  
  
SELECT codigo, tipo  
  
FROM tipo  
  
WHERE precio = (SELECT MIN(precio) FROM pista);
```

- Consulta de registros
 - SUBCONSULTAS

```
-- Codigo y tipo de las pistas cuyo precio está por encima de la media

SELECT codigo, tipo

FROM pista

WHERE precio > (SELECT AVG(precio) FROM pista)
```

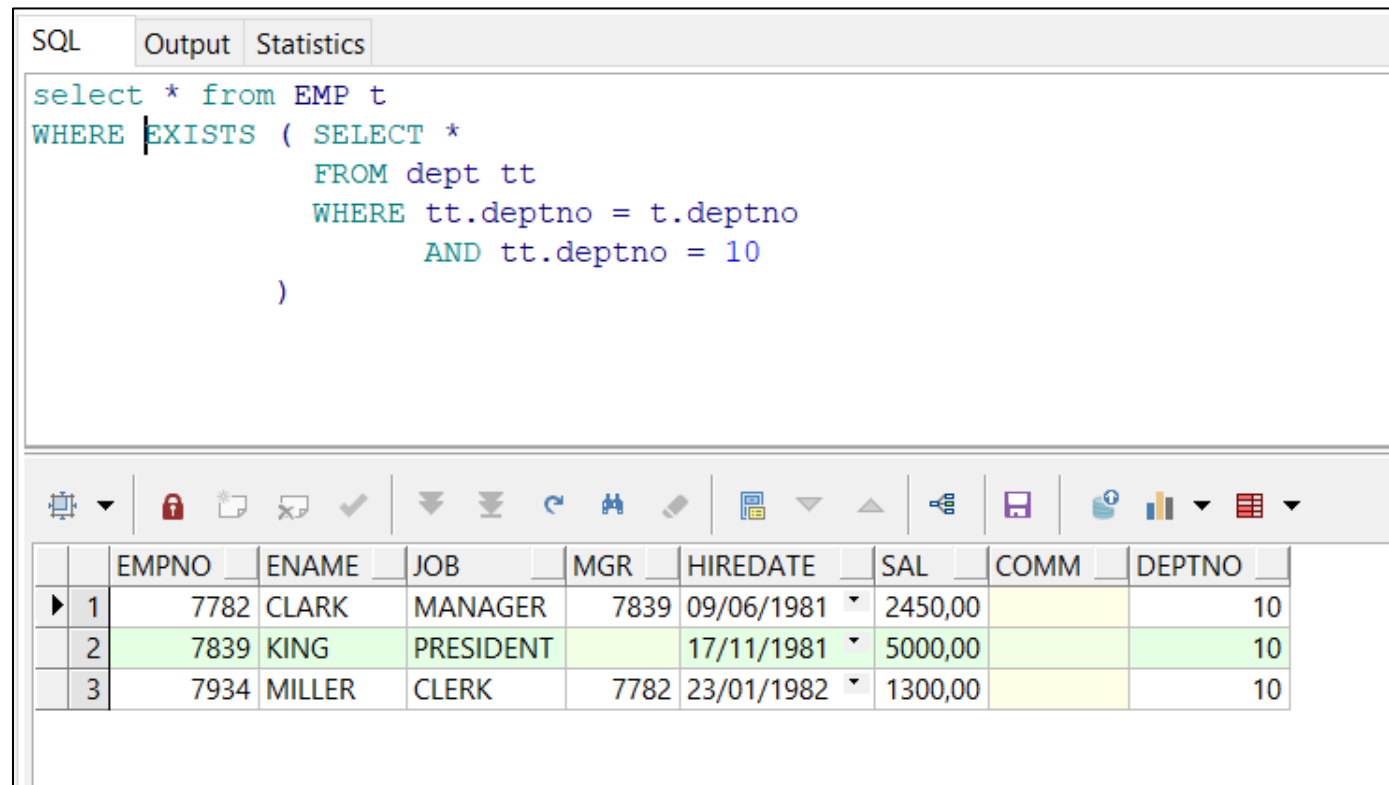
```
-- Nombre y apellidos de los usuarios que aún no han realizado
-- ninguna reserva

SELECT nombre, apellidos

FROM usuario

WHERE id_usuario NOT IN (SELECT id_usuario FROM usuario_reserva)
```

- Consulta de registros
 - SUBCONSULTAS CON EXISTS
 - Se trata de una prueba de existencia, por lo que la subconsulta no devuelve datos sino TRUE O FALSE.



The screenshot shows a SQL IDE interface with a query editor and a results table. The query is a SELECT statement with an EXISTS subquery. The results table displays three rows of employee data, with the second row highlighted in green.

```
SQL Output Statistics
select * from EMP t
WHERE EXISTS ( SELECT *
               FROM dept tt
               WHERE tt.deptno = t.deptno
                  AND tt.deptno = 10
             )
```

	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
▶ 1	7782	CLARK	MANAGER	7839	09/06/1981	2450,00		10
2	7839	KING	PRESIDENT		17/11/1981	5000,00		10
3	7934	MILLER	CLERK	7782	23/01/1982	1300,00		10

- **Consulta de registros**

- **CONSULTAS CON VARIAS TABLAS**

- Como se ha visto anteriormente, aplicando la cláusula WHERE, introducíamos una posibilidad más a la hora de realizar consultas sobre los datos de nuestra base de datos, lo que se conoce como una **consulta de varias tablas o combinación de tablas (en inglés JOIN)**.

```
-- Mostrar, para cada polideportivo, el código y tipo de las pistas  
  
-- que tiene  
  
SELECT PP.id_polideportivo, PP.nombre, P.codigo, P.tipo  
  
FROM polideportivo PP, pista P  
  
WHERE PP.id_polideportivo = P.id_polideportivo
```

- **Consulta de registros**

- **CONSULTAS CON VARIAS TABLAS**

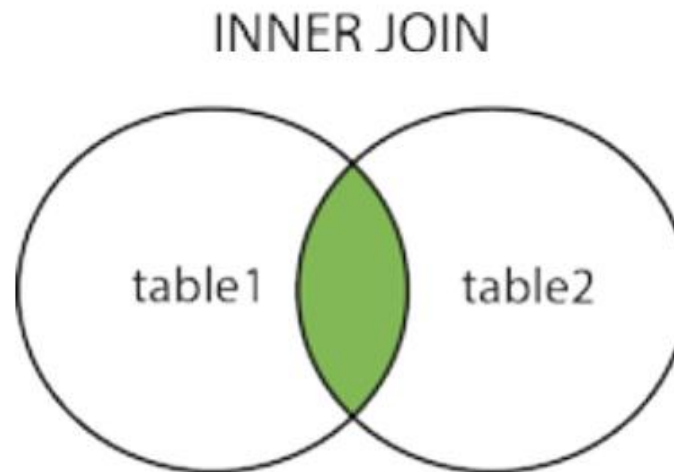
- Como se ha visto anteriormente, aplicando la cláusula WHERE, introducíamos una posibilidad más a la hora de realizar consultas sobre los datos de nuestra base de datos, lo que se conoce como una **consulta de varias tablas o combinación de tablas (en inglés JOIN)**.

```
-- Consulta equivalente  
  
SELECT PP.id_polideportivo, PP.nombre, P.codigo, P.tipo  
  
FROM polideportivo PP  
  
INNER JOIN pista P ON PP.id_polideportivo = P.id_polideportivo
```

- **Consulta de registros**

- **CONSULTAS CON VARIAS TABLAS**

- De esta forma, **al incluir a más de una tabla en la cláusula FROM estamos realizando lo que se conoce como una combinación interna (INNER JOIN)**, de forma que cabe la posibilidad de que sólo se muestren algunos datos de alguna de las tablas, puesto que la combinación interna sólo se queda con aquellos registros que están relacionadas con algún registro de la otra tabla.

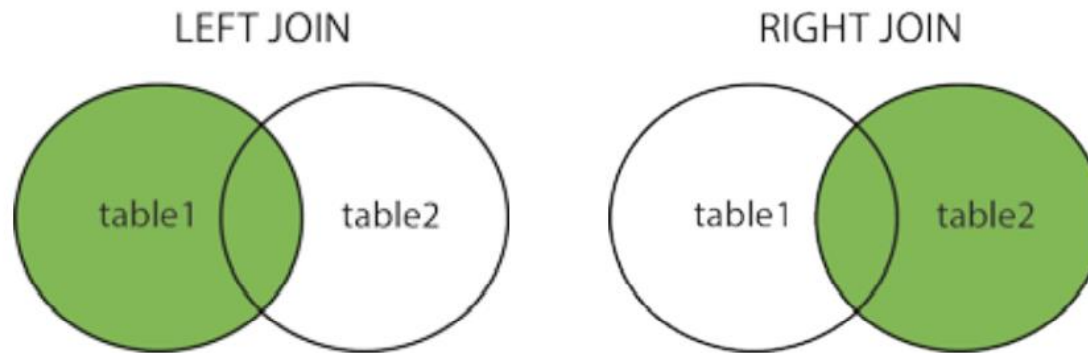


- **Consulta de registros**

- **CONSULTAS CON VARIAS TABLAS**

- Si ahora tenemos en cuenta que algún cliente puede no haber realizado pedido alguno, veremos como **no aparecen en el resultado de la consulta anterior**.
 - En algunos casos eso será lo que queramos, pero quizás **en otros casos nos interesa que su nombre aparezca**, aunque no esté vinculado con ninguno de los pedidos.
 - Si alguna fila de cualquier tabla de la consulta puede no estar relacionado con alguna de las otras, puede ser interesante utilizar un **OUTER JOIN**.
 - Decidir si utilizar un **LEFT [OUTER] JOIN** o bien un **RIGHT [OUTER] JOIN** depende de si el dato que puede no tener relación con la otra tabla está **a la izquierda o la derecha**, respectivamente, según el sentido en que se escribe el código SQL.

- Consulta de registros
 - CONSULTAS CON VARIAS TABLAS



```
-- Mostrar, para cada pista, el código de reserva que ha tenido
-- Si nunca se ha reservado, se mostrarán sólo sus datos

SELECT P.id_pista, P.codigo, P.tipo, R.id_reserva AS codigo
FROM pista P
LEFT [OUTER] JOIN reservas R ON P.id_pista = R.id_pista;

ORDER BY P.codigo
```

- **Consulta de registros**
 - **CONSULTAS CON VARIAS TABLAS**
 - De esta forma mostraremos también los datos de las pistas que no estén relacionados con ninguna reserva. Hay que tener en cuenta que, sólo en este caso, es relevante el orden en el que se especifican las tablas a la hora de definir el JOIN puesto que se incluirán aquellas filas de la tabla del lado izquierdo que no tengan relación con las de la tabla del lado derecho. Es por ello que en los INNER JOIN no se tiene que indicar el sentido de la unión.

```
-- Mostrar cuántas veces se ha reservado cada pista

SELECT P.id_pista, P.codigo, P.tipo, COUNT(R.id_reserva) AS reservas
FROM pista P
LEFT OUTER JOIN reservas R ON P.id_pista = R.id_pista
GROUP BY P.id_pista
ORDER BY P.codigo;
```

- Consulta de registros
 - CONSULTAS CON VARIAS TABLAS

```
-- Mostrar cuántas reservas ha hecho cada usuario

-- (Es posible que algún usuario no haya hecho reservas)

SELECT U.dni, U.nombre, U.apellidos, COUNT(R.id_reserva) AS
numero_reservas

FROM usuario U

LEFT OUTER JOIN usuario_reserva UR ON U.id_usuario = UR.id_usuario

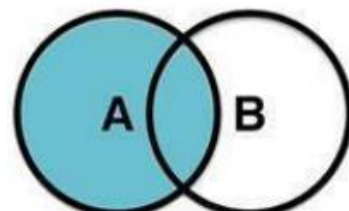
LEFT OUTER JOIN reservas R ON UR.id_reservas = R.id_reserva

LEFT OUTER JOIN pista P ON R.id_pista = P.id_pista;
```

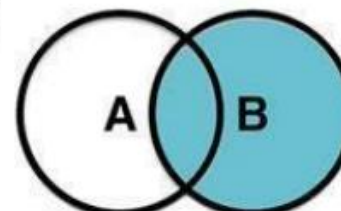
- **Consulta de registros**

- En definitiva, a la hora de construir una consulta SQL hay que añadir en la cláusula FROM todas aquellas tablas que estén involucradas en la consulta:
 - Bien porque se muestre alguna de sus columnas en la cláusula SELECT.
 - Porque se establezca alguna condición con WHERE.
 - Se agrupe por alguno de sus campos.
 - O incluso simplemente SI dicha tabla haga de puente entre dos tablas que deban estar involucradas en dicha consulta.

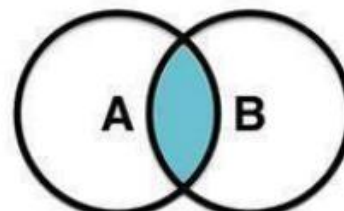
SQL JOINS



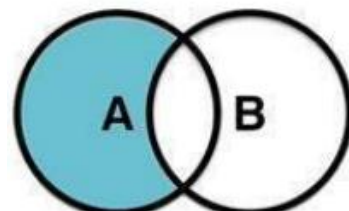
```
SELECT <fields list>  
FROM TableAA  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



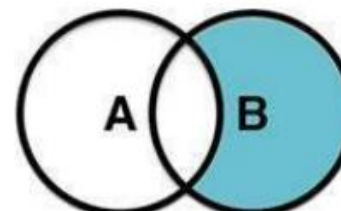
```
SELECT <fields list>  
FROM TableAA  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



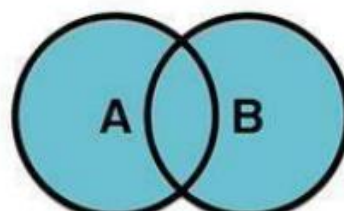
```
SELECT <fields list>  
FROM TableAA  
INNER JOIN TableB B  
ON A.Key = B.Key
```



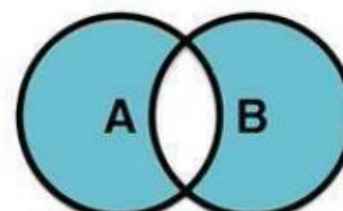
```
SELECT <fields list>  
FROM TableAA  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



```
SELECT <fields list>  
FROM TableAA  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL
```



```
SELECT <fields list>  
FROM TableAA  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <fields list>  
FROM TableAA  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL OR B.Key IS NULL
```

- **Consulta de registros**

- **UNIÓN E INTERSECCIÓN DE CONSULTAS**

- La **unión** de consultas permite unir los resultados de dos consultas totalmente diferentes como si fuera el de una sola. Se realiza mediante la instrucción **UNION** y muestra los resultados sin repeticiones.

```
-- Código y tipo de las pista abiertas y cerradas, indicando
-- el estado actual

SELECT 'abierta', codigo, tipo

FROM pista

WHERE id IN (SELECT id_pista FROM pista_abiertas)

UNION

SELECT 'cerrada', codigo, tipo

FROM pista

WHERE id IN (SELECT id_pista FROM pista_cerradas);
```

- **Consulta de registros**
 - **UNIÓN E INTERSECCIÓN DE CONSULTAS**
 - La **intersección** de consultas muestra sólo los valores que aparecen en las dos consultas que se intersecan. Se realiza mediante la instrucción **INTERSECT**:

```
-- Ciudades con polideportivo que cuentan con usuarios registrados

SELECT ciudad

FROM usuario

INTERSECT

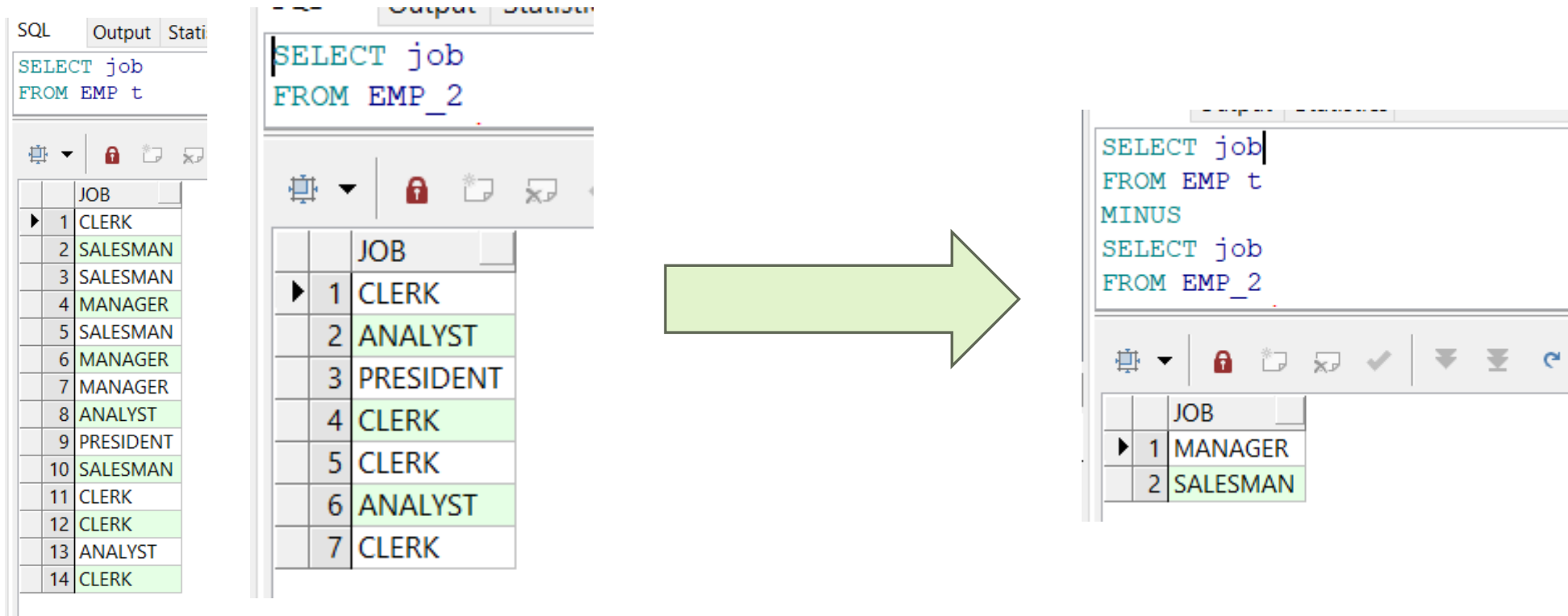
SELECT ciudad

FROM polideportivo;
```

- Consulta de registros

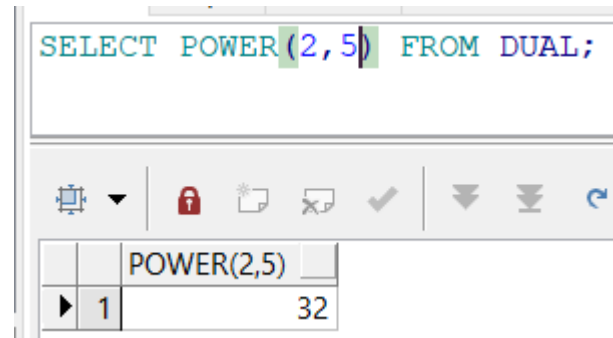
- MINUS

- La operación **minus** devuelve las filas de la primera sentencia select que no están en la segunda select.



- **Tabla dual**

- Es una tabla muy pequeña con únicamente una fila y una columna.
- Recoge los resultados de operaciones aritméticas y funciones de cálculo de Oracle.
- Suele usarse para probar funciones.



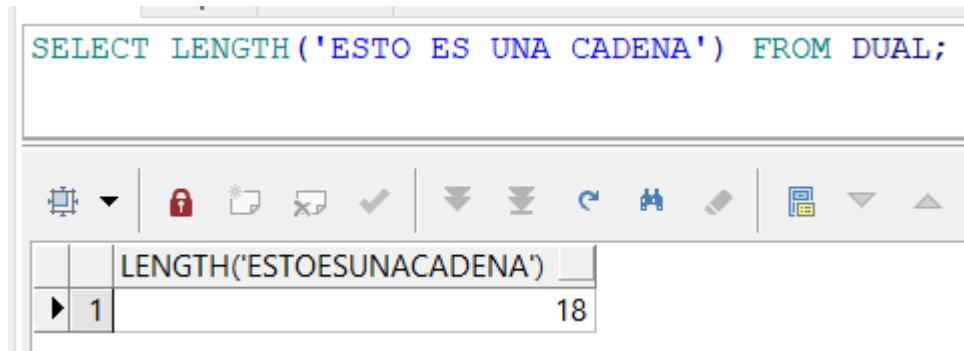
The screenshot shows an SQL IDE interface. The top pane contains the query: `SELECT POWER(2,5) FROM DUAL;`. The bottom pane shows the result of the query, which is a single row with the value 32. The result is displayed in a table with one column labeled 'POWER(2,5)' and one row with the value 32.

	POWER(2,5)
1	32

FUNCIONES DE ORACLE



- **Funciones para cadenas de caracteres**
- **LENGTH(varchar):** Devuelve la longitud, en caracteres, de una cadena de texto.



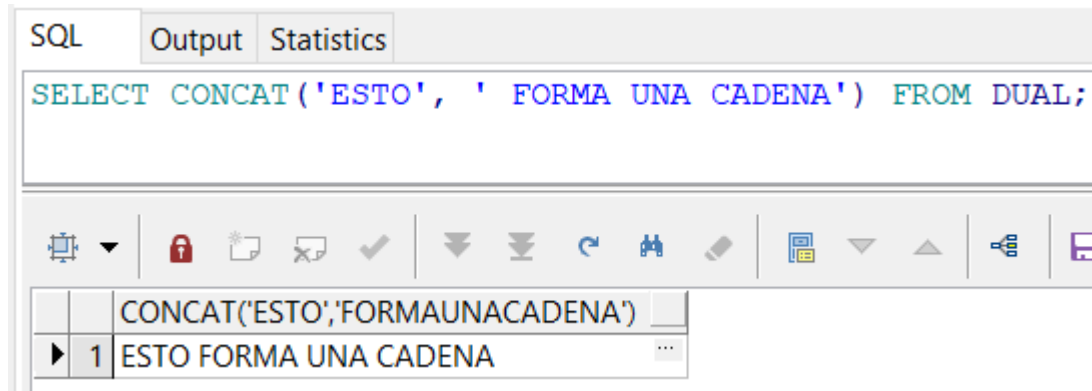
The screenshot shows a SQL query editor with the following text:

```
SELECT LENGTH('ESTO ES UNA CADENA') FROM DUAL;
```

Below the editor is a toolbar with various icons. At the bottom, a results grid displays the output of the query:

	LENGTH('ESTOESUNACADENA')
1	18

- **CONCAT(vrch1, vrch2):** Concatena las cadenas de texto que se pasan como parámetros.



The screenshot shows a SQL query editor with the following text:

```
SELECT CONCAT('ESTO', ' FORMA UNA CADENA') FROM DUAL;
```

Below the editor is a toolbar with various icons. At the bottom, a results grid displays the output of the query:

	CONCAT('ESTO','FORMAUNA CADENA')
1	ESTO FORMA UNA CADENA

- **Funciones para cadenas de caracteres**
- **RTRIM(vrch, [caracteres]):** Elimina los espacios en blanco al final de la cadena, u otros caracteres indicados.

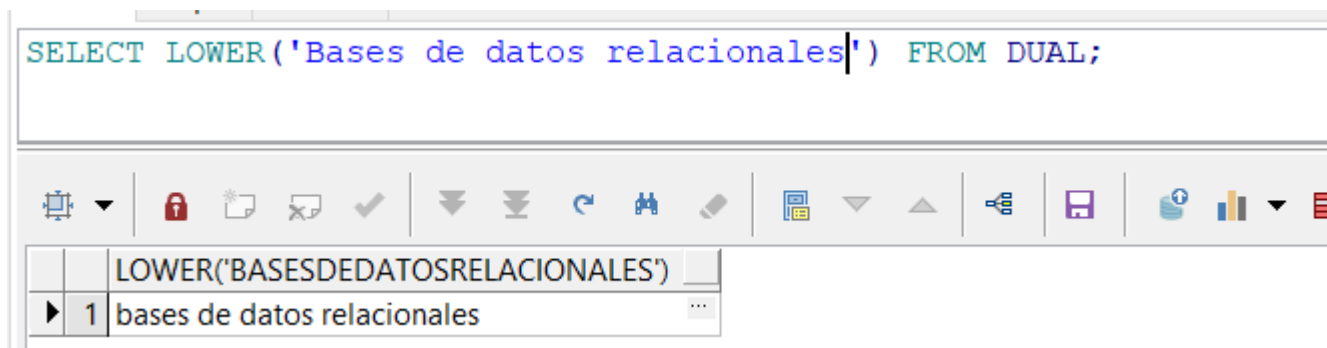
```
SELECT RTRIM('MySQL ');  
  
'MySQL '
```

- **LTRIM(vrch [caracteres]):** Elimina los espacios en blanco al comienzo de la cadena , u otros caracteres indicados.

```
SELECT LTRIM(' MySQL ');  
  
'MySQL '
```

- **Funciones para cadenas de caracteres**
- **LOWER(vrch):** Devuelve la cadena convertida a minúsculas

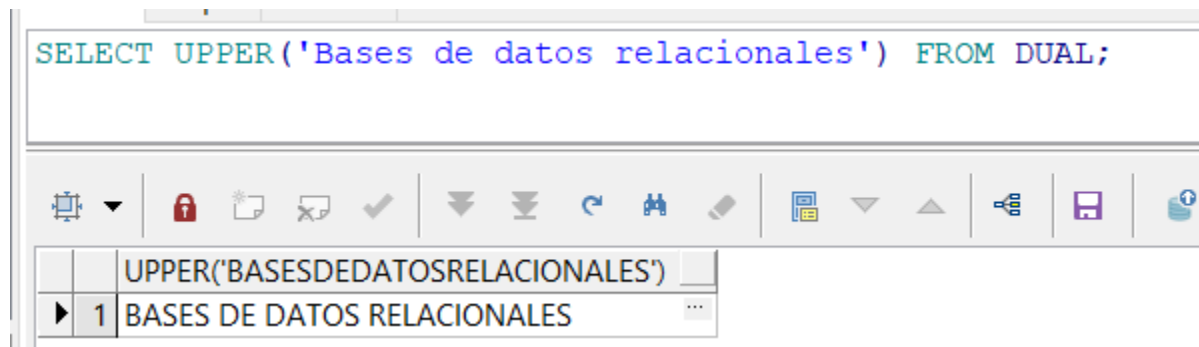
```
SELECT LOWER('Bases de datos relacionales') FROM DUAL;
```



	LOWER('BASESDEDATOSRELACIONALES')
1	bases de datos relacionales

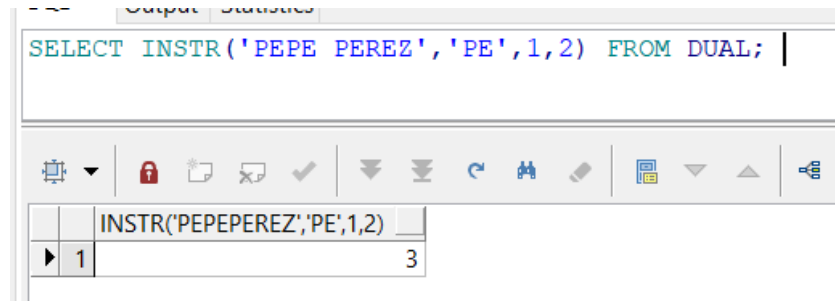
- **UPPER(vrch):** Devuelve la cadena convertida a mayúsculas

```
SELECT UPPER('Bases de datos relacionales') FROM DUAL;
```



	UPPER('BASESDEDATOSRELACIONALES')
1	BASES DE DATOS RELACIONALES

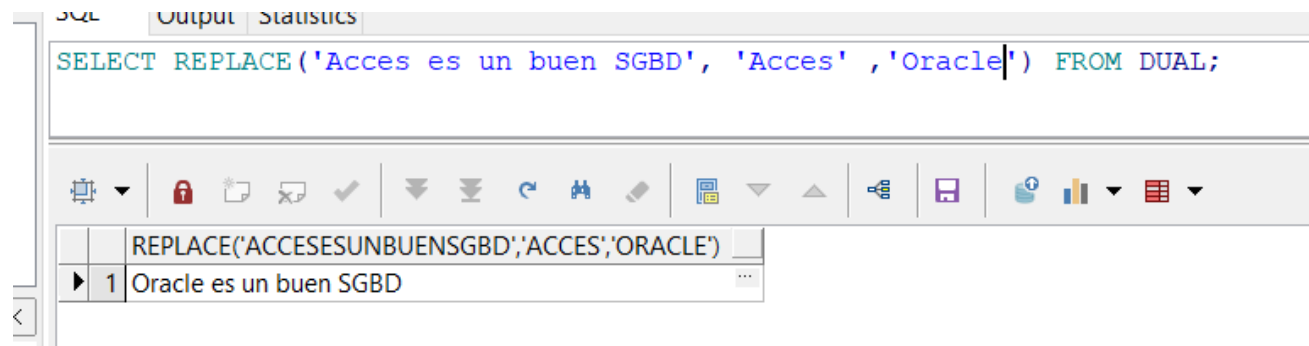
- **Funciones para cadenas de caracteres**
- **INSTR(vrch, 'caracter', inicio, n):** Busca la enésima ocurrencia de un carácter en una cadena comenzando a buscar en el lugar inicio.



The screenshot shows the SQL Developer interface. The SQL editor contains the query: `SELECT INSTR('PEPE PEREZ', 'PE', 1, 2) FROM DUAL;`. Below the editor, the result grid shows a single row with the value 3.

	INSTR('PEPEPEREZ','PE',1,2)
1	3

- **REPLACE(vrch, from_vrch, to_vrch):** Reemplaza todas las ocurrencias de 'from_vrch' por 'to_vrch' que aparezcan en la cadena 'vrch'.



The screenshot shows the SQL Developer interface. The SQL editor contains the query: `SELECT REPLACE('Acces es un buen SGBD', 'Acces', 'Oracle') FROM DUAL;`. Below the editor, the result grid shows a single row with the value Oracle es un buen SGBD.

	REPLACE('ACCESUNBUENSGBD','ACCES','ORACLE')
1	Oracle es un buen SGBD

- **Funciones para cadenas de caracteres**
- **SUBSTRING(vrch, pos), SUBSTRING(vrch, pos, len):** Devuelve una subcadena de 'vrch' comenzando en la posición 'pos'. En el segundo caso tomará caracteres hasta completar una subcadena de tamaño 'len'

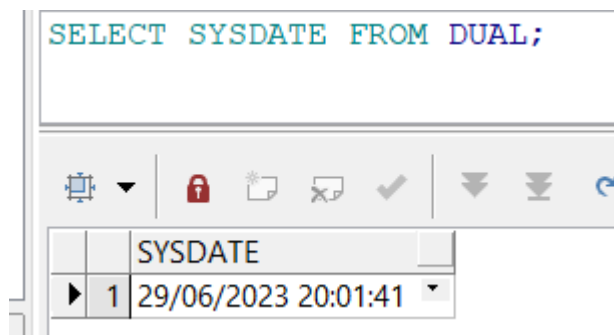
```
SELECT SUBSTRING('MySQL es un buen SGBD', 7);
```

```
'es un buen SGBD'
```

```
SELECT SUBSTRING('MySQL es un buen SGBD', 7, 2);
```

```
'es'
```

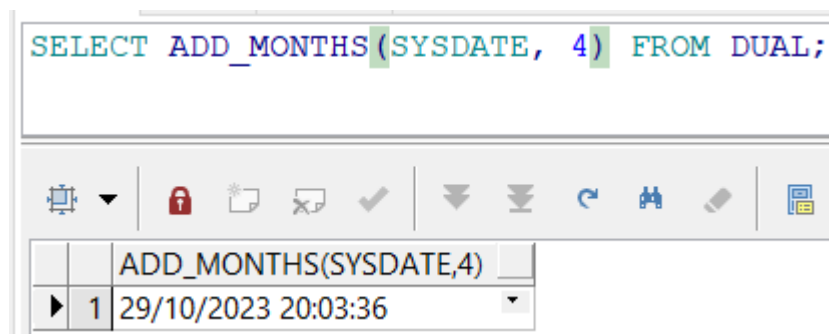
- **Funciones de fecha**
- **SYSDATE:** Devuelve la fecha actual del sistema



A screenshot of the SQL Developer interface. The top pane shows the SQL query: `SELECT SYSDATE FROM DUAL;`. The bottom pane shows the execution results in a table with one column named 'SYSDATE' and one row containing the value '29/06/2023 20:01:41'.

	SYSDATE
1	29/06/2023 20:01:41

- **ADD_MONTHS(fecha, a):** Añade meses a la fecha, si el valor de “a” negativo, en lugar de sumar meses se restan.

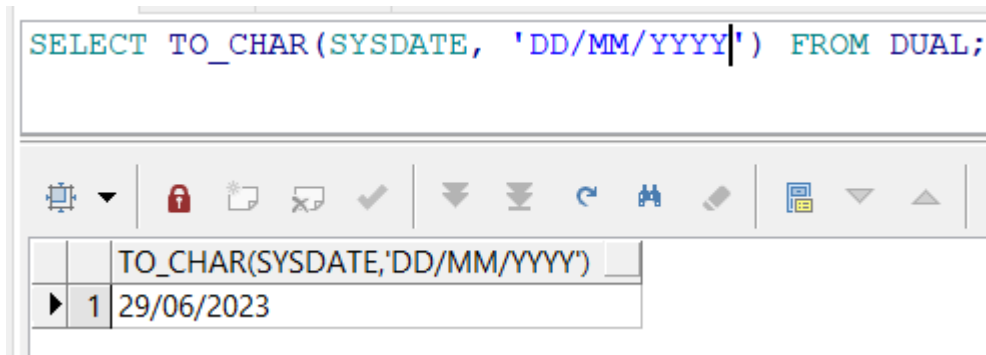


A screenshot of the SQL Developer interface. The top pane shows the SQL query: `SELECT ADD_MONTHS(SYSDATE, 4) FROM DUAL;`. The bottom pane shows the execution results in a table with one column named 'ADD_MONTHS(SYSDATE,4)' and one row containing the value '29/10/2023 20:03:36'.

	ADD_MONTHS(SYSDATE,4)
1	29/10/2023 20:03:36

- **Funciones de fecha**
- **TO_CHAR(date, mascara):** Convierte la fecha a tipo carácter, según el formato de “mascara”.

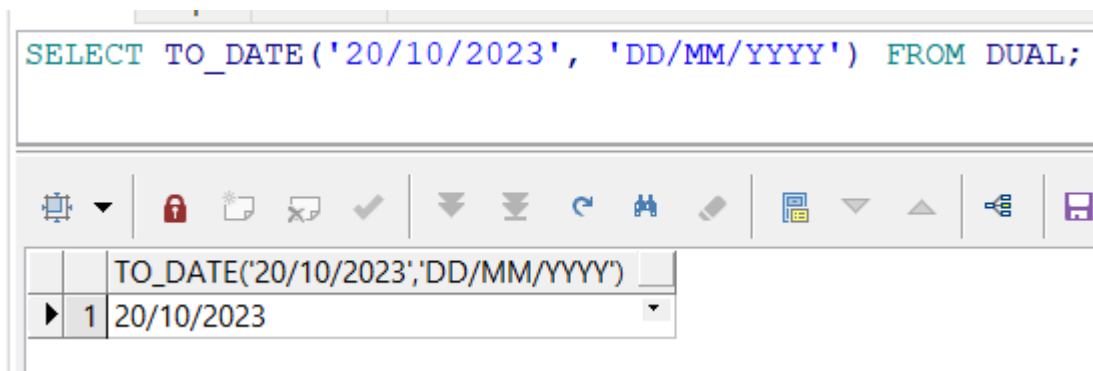
```
SELECT TO_CHAR(SYSDATE, 'DD/MM/YYYY') FROM DUAL;
```



	TO_CHAR(SYSDATE,'DD/MM/YYYY')
1	29/06/2023

- **TO_DATE(cadena, mascara):** Convierte la cadena de texto en fecha, según la máscara indicada.

```
SELECT TO_DATE('20/10/2023', 'DD/MM/YYYY') FROM DUAL;
```



	TO_DATE('20/10/2023','DD/MM/YYYY')
1	20/10/2023

- Máscaras para funciones TO_CHAR y TO_DATE

Formato	Significado	Ejemplo
MM	Número de mes	12
MON	Mes con tres letras	AGO
MONTH	Mes con todas las letras	AGOSTO
DD	Día del mes	29
YYYY	Año con cuatro dígitos	2023
HH	Hora del día (de 1 a 12)	11
H24	Hora del día (de 1 a 24)	23

VISTAS



- La creación de **vistas** permite almacenar consultas como si se trataran de nuevas tablas con la finalidad de utilizar el resultado de estas en otras consultas más complejas.
- Cuando se crea una vista se genera lo que se conoce como una '**tabla lógica**' que permite asignar un nombre al resultado de una consulta y utilizar ésta más adelante y siempre actualizada.
- Hay que tener en cuenta que realmente la consulta que se ha creado como vista **no se encuentra almacenada**, sino que tiene que ser generada cada vez que se deba utilizar.

-- Vista que almacena la consulta que mostraría el número de
-- pista que hay en cada polideportivo

```
CREATE VIEW pista_por_polideportivo
```

```
AS
```

```
SELECT PP.id_polideportivo, PP.nombre, COUNT(*) AS cantidad
```

```
FROM polideportivo PP, pista P
```

```
WHERE PP.id_polideportivo = P.id_polideportivo
```

```
GROUP BY PP.id_polideportivo
```

- Si ahora suponemos que nos pidieran conocer el polideportivo que más pistas tiene, sólo tendríamos que realizar una consulta utilizando la vista creada anteriormente.

```
-- Nombre del polideportivo que más pistas tiene  
  
SELECT nombre  
  
FROM pista_por_polideportivo  
  
WHERE cantidad = (SELECT (MAX(cantidad)  
                        FROM pista_por_polideportivo))
```

ÍNDICES



- Un **índice** es una **estructura asociada a una o más columnas de una tabla** que permite el acceso directo a una fila, tal que se reducen el número de bloques que el SGBD tiene que comprobar, haciendo que las búsquedas tengan un mejor rendimiento.
- Los índices pueden resultar **contraproducentes** si los introducimos sobre campos triviales a partir de los cuales **no se realiza ningún tipo de petición** ya que, con su creación estamos ralentizando otras tareas de la base de datos como son la edición, inserción y borrado, luego no podemos indexar todos los campos de una tabla.
- La sintaxis para la creación de un índice es:

```
CREATE [UNIQUE] INDEX <NOMBRE_INDICE>  
ON <NOMBRE_TABLA> (column1, column2, ...)
```


- Los nombres de los índices suelen empezar por IDX_.
- Son columnas candidatas a ser indexadas:
 - Los campos que han de tener **un valor único**, sin ser PK. Son un tipo especial de índices llamados **UNICOS (UNIQUE INDEX)**.
 - Las **columnas que son FK**, ya que el proceso de selección puede acelerarse sensiblemente si indexamos los campos que sirven de nexo entre dos tablas.
 - Columnas usadas como **filtros** en las consultas.

- Consideraciones para tener en cuenta:
 - Las columnas con **alta cardinalidad** (amplio rango de valores diferentes), serán **filtros más selectivos**, los cuales deberán ponerse primero en una consulta.
 - En el caso de los **índices compuestos**, el orden en el que se declaran estas columnas deberá ser de la más selectiva a la menos selectiva (siempre que sea posible).
 - En algunos casos es conveniente sustituir índices compuestos por varios índices simples.
 - No se usan los índices sobre las columnas sobre las que se aplica una función.
 - Las PKs tienen automáticamente asociado un índice.
 - Si tenemos el siguiente índice: **CREATE INDEX IDX_TABLE_01 ON TABLE (col1, col2, col3)**, las consultas que acceden sólo a col1 ó a col1 y col2 aumentan su rendimiento, pero en el caso de empezar el acceso por col2 o col3 es posible que no se use el índice.

OTRAS FUNCIONALIDADES



- Funciones de transformación:**

- **TRANSLATE(cadena1, parte,cadena2):** Mira cada carácter de cadena1. busca en ella cada carácter de “parte” y lo sustituye “en cadena2” en la misma posición en la que se encontraba en “cadena1”.

```
SELECT TRANSLATE ('el perro de san roque...','rp','gt') FROM dual;
```

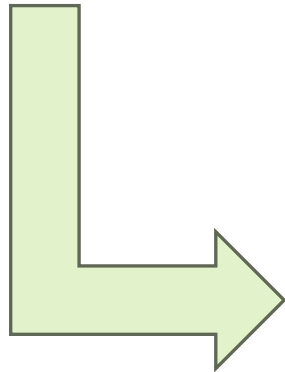
TRANSLATE('ELPERRODESANROQUE...','RP','GT')	
1	el teggo de san goque...

- **DECODE(variable, cadena1, parte1, cadena2,parte2,..., parteDefecto):** Analiza el valor de “variable” contiene el valor cadena1, se muestra parte1 y así con todos los demás casos, sino en ninguna se muestra valor por defecto.

SQL		Output	Statistics
SELECT DECODE('VALORR', 'VALOR', 'ES VALOR', 'NO ES VALOR') FROM DUAL;			
DECODE('VALORR','VALOR','ESVALOR','NOESVALOR')			
1	NO ES VALOR		

- Creación de tablas con select:

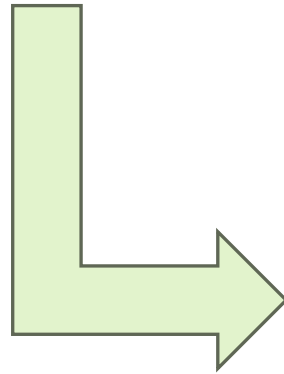
CREATE TABLE <nombreTabla> **AS**
SELECT listaSeleccion
FROM <nombreTablaOrigen>;



```
CREATE TABLE Empleados_nuevo AS  
SELECT *  
FROM EMP;
```

- Insertción con select:

INSERT INTO <nombreTabla> (
SELECT listaSeleccion
FROM <nombreTablaOrigen>);



```
INSERT INTO Empleados_nuevo (  
    SELECT *  
    FROM EMP  
);  
COMMIT;
```

Muchas gracias por vuestra atención

