

## Fastbook 06

# Tratamiento de Datos (Excel y SQL)

PostgreSQL: primeros pasos



## 06. PostgreSQL: primeros pasos

En este fastbook, daremos los primeros pasos con las sentencias de SQL de Postgres y realizaremos las primeras queries en la base de datos.

En primer lugar, conoceremos las sentencias que nos permitirán moldear nuestra base de datos (DDL). A continuación, nos centraremos en los comandos más habituales y usados en la mayoría de las consultas realizadas a una base de datos.

*Autor: Breogán Cid y Carlos Manchón*

**Contexto**

**Restricciones**

**DDL (data definition language)**

**DML (data manipulation language)**

**Posibilidades que ofrece SELECT**

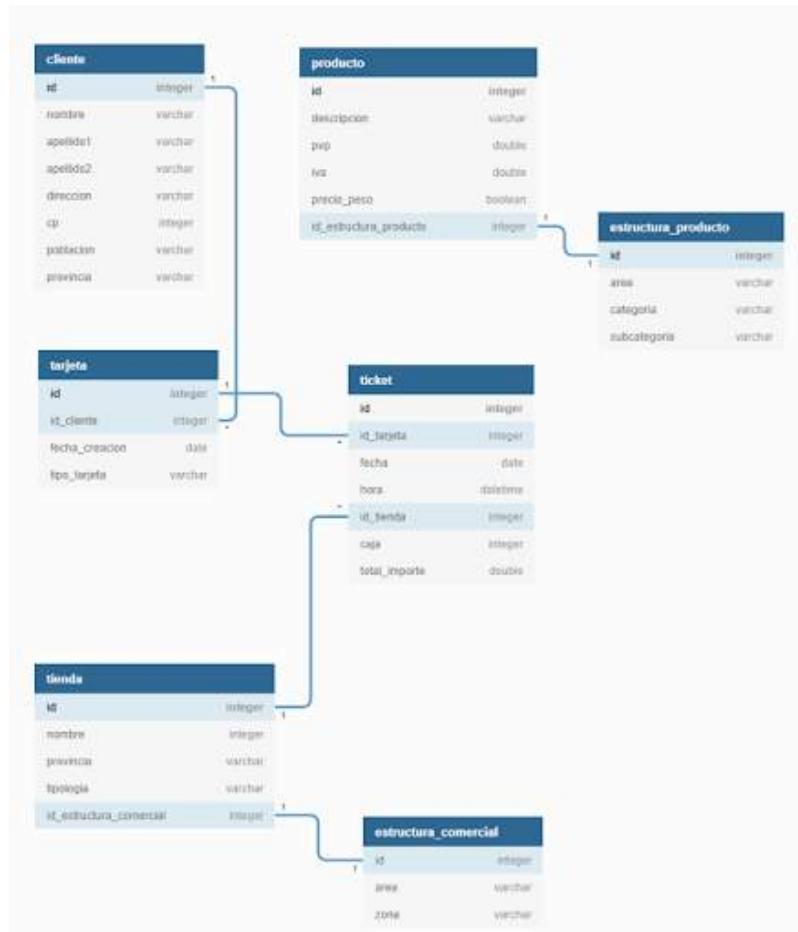
**Conclusiones**

# Contexto

Edix Educación

Para el correcto aprendizaje de este tema, se adjunta un [dump](#) de base de datos que deberás cargar en tu servidor PostgreSQL. En él se incluye todo lo necesario para que puedas ir avanzando con el temario.

En concreto, **nuestra base de datos Supermarket** contiene la información de una pequeña cadena de supermercados que acaba de implantarse en España. La información de dicha base de datos está estructurada tal y como muestra la siguiente imagen.



En el diagrama podemos observar las tablas existentes y las relaciones que hay entre ellas. Vamos a analizarlas.

## Tablas

### Cliente

Esta tabla almacena la **información personal de cada uno de los clientes fidelizados** que tiene en cartera nuestra compañía de supermercados. Para cada cliente almacenaremos su nombre, apellidos, dirección, código postal, población y provincia.

### Tarjeta

Todas las compras realizadas en nuestro supermercado se realizan a través de una tarjeta que identifica a un cliente. **Un cliente puede tener varias tarjetas.** Las tarjetas pueden ser de débito o de crédito. También se almacena la fecha de creación de la tarjeta.

### **Estructura\_comercial**

La red de tiendas de nuestra cadena de supermercado está organizada según la estructura comercial recogida en esta tabla. La **implantación de tiendas** se organiza en un primer nivel de área y, dentro de cada área, en una serie de zonas.

### **Tienda**

Para cada tienda existente dentro de la red de la compañía, **se almacena la información del nombre de la tienda, su provincia, su tipología y un identificador** a la tabla denominada 'Estructura\_comercial', que indica en qué área y zona se encuentra la tienda.

### **Estructura\_producto**

De la misma manera que las tiendas, los productos presentan una organización similar. La **jerarquía de producto** sería la siguiente:

Área > Categoría > Subcategoría > Producto

## Producto

Esta tabla **almacena cada uno de los productos que se venden dentro de la red de tiendas de la compañía**. Por cada producto almacenaremos su descripción, su precio, el IVA asociado, si se trata de un producto cuyo precio es al peso/corte. Por último, almacenamos también un identificador a la tabla 'Estructura\_producto', que nos proporcionará bajo qué área, categoría y subcategoría se sitúa el producto en cuestión.

## Ticket

En esta tabla se almacena la **información de cada uno de los tickets de compra** que se hayan realizado en alguna de las tiendas de la cadena. Por cada ticket, almacenaremos su fecha de realización, el número de caja, el importe total del ticket y dos identificadores que apuntan a la tabla de tienda y a la de cliente.

## Relaciones

Aunque ya hemos adelantado alguna de ellas, existen las siguientes relaciones:

### Cliente - Tarjeta

Relación '1-n', es decir, un cliente puede poseer varias tarjetas.

### Producto - Estructura producto

Relación '1-1', un producto tiene una única combinación de área – categoría – subcategoría.

### Tienda - Estructura comercial

Relación '1-1', una tienda está situada en una única área – zona.

### Tienda - Ticket

Relación '1-n', una tienda puede tener asociados 'n' tickets.

### Tarjeta - Ticket

Relación '1-n', una tarjeta puede tener asociados 'n' tickets.

# Restricciones

 Edix Educación

---

Antes de entrar ya en materia con nuestra nueva base de datos, conviene repasar las **posibles restricciones** que existen en nuestras tablas de base de datos.

En concreto, vimos ya las restricciones de **primary key** (o clave primaria o PK) y la de **foreign key** (o clave foránea o FK). Recordaremos estas y añadiremos otras: check, not null y unique.

1

## Primary key

Utilizaremos una restricción de clave primaria cuando queramos indicar que una columna o un grupo de columnas de una tabla van a **identificar de forma única** a cada una de las filas de una tabla.

Podemos definir una columna como clave primaria al definir la tabla, por ejemplo:

```
CREATE TABLE producto (
    id_producto integer PRIMARY KEY,
    nombre text,
    precio numeric
)
```

2

## Foreign key

Utilizaremos una restricción de clave foránea cuando queramos indicar que una columna o un grupo de columnas de una tabla deben coincidir exactamente con los valores fila a fila de otra tabla. Decimos que las dos tablas implicadas mantienen una **integridad referencial**.

Podemos definir una columna como clave foránea al definir la tabla, por ejemplo:

```
CREATE TABLE compra (
    id_compra integer PRIMARY KEY,
    id_producto integer REFERENCES producto(id_producto),
    cantidad integer
)
```

La instrucción se leería como que la columna ‘product\_number’ de la tabla ‘orders’ referencia a la columna ‘product\_number’ de la tabla ‘products’.

---

**Como supondrás, para poder ejecutar esta instrucción es necesario que exista previamente la tabla ‘products’.**

3

### Restricción check

Utilizaremos una restricción check cuando queramos que el valor de una columna de una tabla satisfaga una expresión booleana, esto es, que la expresión pueda tomar el valor de verdadero o falso al evaluarla.

Podemos definir una restricción check para una columna al crear la tabla que la contiene:

```
CREATE TABLE persona (
    dni integer,
    nombre text,
    edad numeric CHECK (edad > 0)
)
```

---

**Con esta restricción, obligamos que, al insertar datos en esa tabla, el valor que imputemos al campo ‘edad’ tenga un valor mayor a 0.**

4

## Restricción not-null

Decimos que **una columna toma un valor NULL cuando el contenido está vacío, sin informar.** Utilizando una restricción not-null podemos obligar que cierta columna de una tabla siempre tenga dicho campo informado.

La motivación puede obedecer a que dicha información es imprescindible recogerla y almacenarla.

Podemos **definir una restricción not-null** para una columna al crear la tabla que la contiene:

```
CREATE TABLE producto (
    id_producto integer NOT NULL,
    nombre text NOT NULL,
    precio numeric NOT NULL CHECK (precio > 0)
)
```

En este caso, estamos **combinando dos restricciones**, la de tipo not-null y la de tipo check.

## 5

## Restricción unique

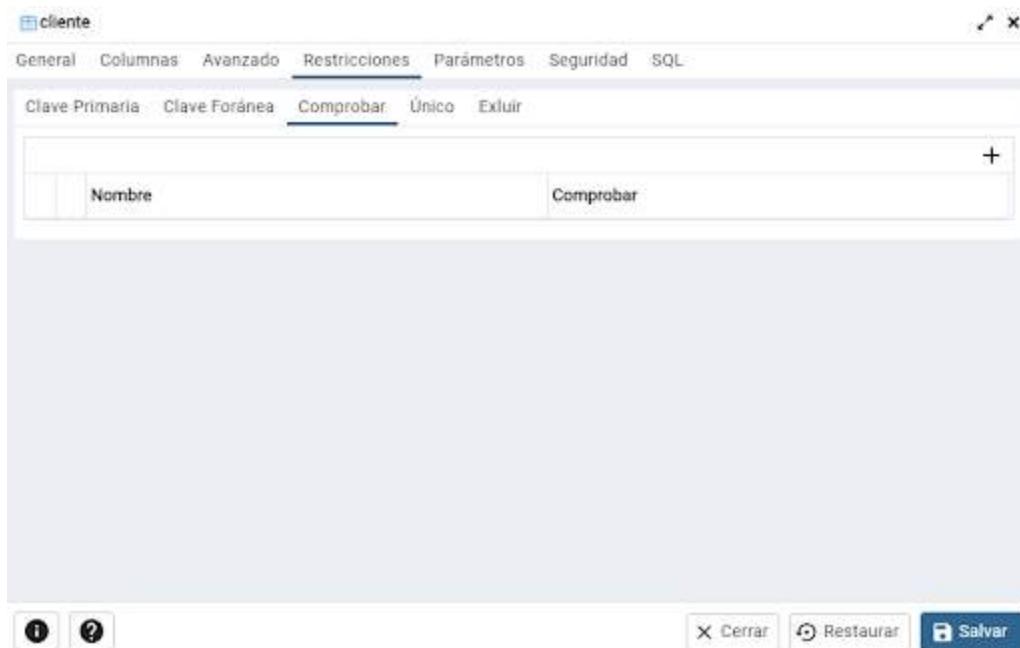
Utilizaremos una restricción unique sobre una columna o grupos de columnas cuando queremos asegurarnos de que los valores que toma en las distintas filas de la tabla son únicos.

Es importante reseñar que, si bien puede parecer que una restricción unique es igual a una restricción de clave primaria, no lo es realmente. Una clave primaria nunca puede tomar el valor NULL, una restricción unique sí (salvo que añadamos una restricción not-null). De la misma manera, una tabla no puede tener más de una clave primaria; sin embargo, una tabla sí que puede tener más de una restricción unique.

Podemos definir una restricción unique para una columna al crear la tabla que la contiene:

```
CREATE TABLE persona (
    id_persona integer PRIMARY KEY,
    dni text UNIQUE,
    nombre text,
    edad numeric
)
```

Es importante destacar que, aunque en los ejemplos anteriores hayamos definido las restricciones en las consultas de creación de nuestra base de datos, ya que es la forma más cómoda y rápida y la más usada por personas expertas, dentro de nuestro PgAdmin tendremos una interfaz gráfica para realizar las mismas opciones.



# DDL (data definition language)

X Edix Educación

---

## CREATE DATABASE

Es el comando que lanzamos cuando queremos **crear una nueva base de datos**. Su sintaxis básica es la siguiente:

```
CREATE DATABASE nombre
```

Si abrimos el fichero dump, que hemos cargado con un editor de texto, y nos vamos a la línea 27, vemos la siguiente instrucción:

```
CREATE DATABASE supermarket WITH TEMPLATE = templateo ENCODING =
'UTF8' LC_COLLATE = 'Spanish_Spain.1252' LC_CTYPE = 'Spanish_Spain.1252'.
```

## ALTER DATABASE

Es el comando que lanzamos cuando queremos **editar los atributos de una base de datos existente**. Sus posibles opciones son las mostradas en la siguiente imagen.

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]  
where option can be:  
    CONNECTION LIMIT connlimit  
  
ALTER DATABASE name RENAME TO new_name  
  
ALTER DATABASE name OWNER TO new_owner  
  
ALTER DATABASE name SET TABLESPACE new_tablespace  
  
ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }  
ALTER DATABASE name SET configuration_parameter FROM CURRENT  
ALTER DATABASE name RESET configuration_parameter  
ALTER DATABASE name RESET ALL
```

---

A nuestro nivel, noaremos uso de este comando, salvo quizás para renombrar la base de datos (2º alter database del listado). Hay que tener en cuenta que, para realizar el cambio de nombre, la base de datos no debe estar activa.

## DROP DATABASE

Es el comando que lanzamos cuando queremos **borrar una base de datos existente**. Recuerda que para realizar esta operación no debe haber conexiones abiertas Su sintaxis es la siguiente:

```
DROP DATABASE nombre
```

## CREATE TABLE

Es el comando que lanzamos cuando queremos **crear una tabla dentro de nuestra base de datos**.

La sintaxis completa puede llegar a ser muy compleja por la cantidad de parámetros que se pueden configurar, en estos momentos nos quedaremos con definiciones básicas de tablas:

```
CREATE TABLE nombre {  
    Nombre_variable1 tipo_variable1 [restriccion_variable1],  
  
    Nombre_variable2 tipo_variable2 [restriccion_variable2],  
  
    Nombre_variableN tipo_variableN [restriccion_variableN]  
}
```

Nota: [] = Opcional

---

**Existe una opción muy útil y potente de generar tablas a partir de otras ya existentes. Haremos uso de esta posibilidad cuando queramos generar una tabla a partir de otra, conteniendo un subconjunto de los datos.**

Generemos una tabla con los mismos campos que la tabla original cliente pero que solo contenga a las personas cuya provincia sea Madrid:

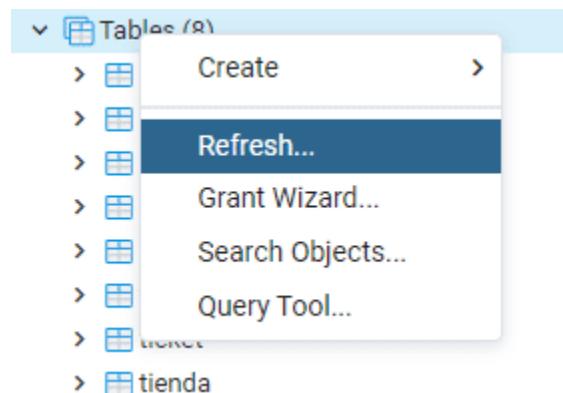
```
CREATE TABLE cliente_madrid AS  
SELECT * from cliente  
  
WHERE provincia = 'Madrid'
```



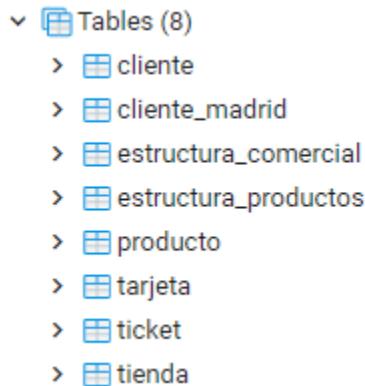
Es posible que cuando traslades el código al editor te falle al ejecutar, se debe a la comilla (") utilizada en Madrid. Sustitúyela por la comilla simple del teclado (la que está en la misma tecla que el símbolo '?').

Desde PgAdmin, si ejecutamos el código SQL de arriba en nuestro editor, veremos cómo se ha creado una nueva tabla que contiene a los clientes de la provincia de Madrid.

Para ver la nueva tabla en el árbol de la izquierda, será necesario que pulses con el botón derecho sobre el apartado ‘Tables’ dentro del schema ‘Public’ y selecciones ‘Refresh’:



Efectivamente, se ha generado una nueva tabla:



---

La única pega de esta funcionalidad es que no traspasa también las restricciones que hayamos definido en la tabla original.

---

## ALTER TABLE

¡No te preocupes! Para ese pequeño problema, existe este comando. Nos permitirá realizar **modificaciones en la definición de la tabla que queramos**. Así pues, podremos realizar entre otras las siguientes operaciones:

- 1 Añadir restricciones.
- 2 Modificar restricciones.
- 3 Eliminar restricciones.

4

Añadir columnas.

5

Eliminar columnas.

6

Modificar columnas (por ejemplo, su tipo: SET DATA TYPE nuevo\_tipo).

Realicemos las modificaciones sobre nuestra tabla 'cliente\_madrid':

```
ALTER TABLE cliente_madrid ALTER COLUMN nombre SET NOT NULL;  
ALTER TABLE cliente_madrid ALTER COLUMN apellido1 SET NOT NULL;
```

```
ALTER TABLE cliente_madrid ALTER COLUMN apellido2 SET NOT NULL;
```

```
ALTER TABLE cliente_madrid ALTER COLUMN direccion SET NOT NULL;
```

```
ALTER TABLE cliente_madrid ALTER COLUMN cp SET NOT NULL;
```

```
ALTER TABLE cliente_madrid ALTER COLUMN poblacion SET NOT NULL;
```

```
ALTER TABLE cliente_madrid ALTER COLUMN provincia SET NOT NULL;
```

```
ALTER TABLE cliente_madrid ADD CONSTRAINT pk_cliente_madrid PRIMARY  
KEY (id);
```

---

Pues ya estaría, fíjate que para añadir la restricción 'not-null', la sintaxis cambia un poco, pero se entiende bien qué pasos se han seguido, ¿verdad?

## DROP TABLE

Cuando terminemos de jugar con nuestra tabla de prueba 'cliente\_madrid', llega el **momento de eliminarla**, para ello, haremos uso del comando que nos toca ahora comentar: **DROP TABLE**. La sintaxis es muy sencilla:

```
DROP TABLE cliente_madrid
```



Hasta aquí la parte de definición de datos, pasaremos ahora a la manipulación de estos.

# DML (data manipulation language)

X Edix Educación

## INSERT

El comando **INSERT** nos permite **añadir nuevas filas a una tabla**. Podremos insertar una fila o más a través de expresiones de valores, o también podremos insertar cero o más filas a partir de una query.

Para hacer las pruebas, creemos **una tabla copia de la tabla producto** y llamémosla 'producto\_copia' (CREATE TABLE producto\_copia AS select \* from producto).

Ahora que ya tenemos creada nuestra propia tabla de pruebas, observemos las **diferentes opciones** que tenemos para agregar nuevos registros a nuestra tabla. Podemos realizarlo a través de expresiones de valores:

Sintaxis:

```
INSERT INTO nombre_tabla  
VALUES  
(valor_columna1, valor_columna2, valor_columna3, ...)
```

En nuestro caso:

```
INSERT INTO producto_copia  
VALUES  
(999, 'Producto prueba', 5.5, 7, false, 1)
```

Nota: Recuerda sustituir las comillas

Si ejecutamos el código, se insertará en nuestra tabla 'producto\_copia' un nuevo producto.

¡Lamentablemente, algo hemos hecho mal! Si recuerdas al realizar una copia de la tabla, **no nos ha generado las restricciones** y, en concreto, esta tabla tenía una **restricción de clave foránea**. ¿Qué ha pasado? Al no especificar clave foránea a la tabla de 'estructura\_producto', hemos insertado un valor 1, pero si consultamos dicha tabla vemos que no existe ningún caso con ese identificador. En definitiva, **se trata de un problema importante de integridad referencial**.

No te preocupes, hemos realizado este ejemplo para que veas lo importante que es **mantener la integridad en los datos**, en caso contrario, se producirían inconsistencias que pueden provocar a futuro muchos problemas.

También podemos agregar varios valores en la misma query:

Sintaxis:

```
INSERT INTO nombre_tabla  
VALUES  
(valor_columna1, valor_columna2, valor_columna3 , ...),  
(valor_columna1, valor_columna2, valor_columna3 , ...),  
(valor_columna1, valor_columna2, valor_columna3 , ...)
```

```
INSERT INTO producto_copia  
VALUES  
(1000, 'Producto prueba 2', 10.5,6,true,1),  
(1001, 'Producto prueba 3', 0.5,16,true,1),  
(1002, 'Producto prueba 4', 4.6,21,false,1),  
(1003, 'Producto prueba 5', 13.5,4,true,1)
```

Además, igual que cuando estábamos creando tablas, también podremos agregar elementos basándonos en una query select.

Insertaremos en ‘producto\_copia’ los productos de la tabla producto cuya columna ‘precio\_peso’ tenga un valor ‘FALSE’.

Sintaxis:

```
INSERT INTO tabla_destino  
SELECT listado_columnas FROM tabla_origen
```

```
INSERT INTO producto_copia  
SELECT * from producto where precio_peso is FALSE
```

## UPDATE

El comando UPDATE nos permitirá **actualizar valores de columnas** en una tabla a partir de una condición booleana (que se puede evaluar como verdadero o falso).

La sintaxis es la siguiente:

```
UPDATE nombre_tabla  
SET  
    nombre_columna1 = valor_columna1,  
    nombre_columna2 = valor_columna2,  
    nombre_columnaN = valor_columnaN  
WHERE condicion_booleana
```

---

**Hay que aclarar que podremos actualizar 1 a 'n' columnas (siendo 'n' el número máximo de columnas de la tabla). Igualmente, podremos actualizar la tabla sobre la totalidad de las filas, sin necesidad de incluir la condición (podemos prescindir de la parte 'WHERE condición booleana').**

Continuemos con nuestra **tabla de pruebas** y actualicemos sus datos a partir de una condición. Por ejemplo, actualicemos todos los productos de la tabla, incrementando su PVP en 1€.

```
UPDATE producto_copia  
SET pvp = pvp + 1
```

---

Si hubiésemos querido actualizar el precio sobre un subconjunto de los productos, podríamos haber puesto algún tipo de condición.

---

Por ejemplo, actualicemos los productos de la tabla cuyo PVP sea inferior a 5€, incrementando su precio en un 20% su valor.

```
UPDATE producto_copia  
SET pvp = 1.2 * pvp  
  
WHERE pvp < 5
```

## **DELETE**

De la misma manera, podremos **eliminar filas de una tabla** atendiendo a una condición (**¡ten cuidado!**, si olvidas poner la condición borrarás la tabla entera).

La sintaxis es la siguiente:

```
DELETE FROM nombre_tabla  
WHERE condicion_booleana
```

Qué mejor forma de usar este comando borrando todas las filas de nuestra tabla-copia (y recuerda: estamos realizando pruebas, lo normal no es borrar todas las filas de una tabla, ¡cuidado!).

```
DELETE FROM producto_copia
```

Tenemos, actualmente, la tabla ‘producto\_copia’ vacía, por lo que pasemos también a eliminar la tabla por completo.

```
DROP TABLE producto_copia
```

## SELECT

Usamos la sentencia SELECT para realizar consultas de datos sobre tabla/s de nuestra base de datos. Su estructura general es la siguiente:

- 1 `SELECT <columna1> [,<columna2>, ..., <columna N>]`
- 2 `FROM <tabla1>[,<tabla2>], ..., [<tabla N>]`
- 3 `[WHERE <condicion >]`
- 4 `[GROUP BY <columna1>, <columna2>, ..., <columnaN>]`
- 5 `[HAVING <condicion>]`
- 6 `[ORDER BY <columna1>, <columna2>, ..., <columnaN>]`
- 7 `[LIMIT N]`

# Posibilidades que ofrece SELECT

X Edix Educación

En este apartado, analizaremos las **distintas posibilidades** que nos ofrece la sentencia SELECT, y dejaremos para fastbooks sucesivos todo lo referente a From, Group By y Having.

1

SELECT

Indicaremos la **columna o las columnas de las que queremos extraer información**. Sobre estas columnas se pueden realizar todo tipos de operaciones e incluso combinaciones entre ellas.

Podremos indicar **columnas específicas** presentes en la tabla:

```
SELECT nombre, apellido1, apellido2  
FROM empleado
```

U obtener todas las columnas disponibles, mediante el uso del operador **'\*'**:

```
SELECT *  
FROM empleado
```

También podremos **aplicar numerosas funciones a nuestras columnas disponibles**, como la realización de operaciones aritméticas, concatenación de textos, o el tratamiento de fechas:

```
SELECT id_empleado, (salario + bonus) / 12  
FROM empleado
```

2

WHERE

El comando WHERE nos va a permitir establecer la condición que deben cumplir las columnas de nuestra tabla para que las filas de ésta sean devueltas a la hora de ejecutar nuestra consulta.

La condición se evaluará a TRUE o FALSE (verdadero o falso) y puede ser una condición o un conjunto de condiciones enlazadas a través de operadores lógicos (AND, OR, NOT).

Aquí, muestro un **listado de todos los empleados** cuyos nombres son Pedro:

```
SELECT id_empleado,nombre, apellido1,apellido2  
FROM empleado  
  
WHERE nombre = "Pedro"
```

Dentro de las **condiciones usadas como filtro**, destacan cinco comandos que nos serán muy útiles en nuestro día a día:

- BETWEEN.
- LIKE.
- IN.
- ORDER BY.
- LIMIT.

## BETWEEN

Usaremos el **comando BETWEEN** para expresar que una columna se encuentra situada entre dos valores (dentro de un umbral).

Así podremos filtrar información entre dos fechas, entre dos números, dos cadenas de texto, etc. Por ejemplo:

```
SELECT id_empleado, nombre, apellido1, apellido2
```

```
FROM empleado
```

```
WHERE salario BETWEEN 25000 AND 50000
```

## LIKE

El comando **LIKE** es una **herramienta muy potente** que nos permite buscar patrones de texto dentro de columnas de este tipo.

Existen **dos elementos muy útiles** para la búsqueda de patrones de texto:

#### El carácter '%'

Permite actuar como comodín y representa a 0,1 o 'n' caracteres (letras o números) cualesquiera.

#### El carácter '\_'

Permite actuar como comodín y representa a 1 carácter cualquiera (letra o número).

Antes de terminar la explicación de este comando, solo nos falta incidir en dos puntos interesantes:

- Podemos añadir el **comando NOT** justo antes del LIKE para buscar elementos que no cumplan con un patrón.
- El comando **LIKE** es CASE SENSITIVE, es decir, es sensible (diferencia) entre mayúsculas y minúsculas. Si queremos que no sea sensible a este cambio, usaremos el comando ILIKE (de INSENSITIVE LIKE).

```
SELECT id_empleado, nombre, apellido1, apellido2
```

```
FROM empleado
```

```
WHERE
```

```
apellido1 LIKE '%ez%' OR apellido2 LIKE '%ez%'
```

## IN

El comando IN es una potente herramienta que nos va a permitir que una determinada columna presente un valor dentro de un listado de elementos.

Por ejemplo, pensemos que queremos buscar dentro de nuestra tabla de empleados a los empleados cuyos nombres sean Carlos, Juan, Luis o Pedro. Como somos principiantes, podríamos construir la condición de la siguiente manera:

WHERE

nombre = “Carlos” OR

nombre = “Juan” OR

nombre = “Luis” OR

nombre = “Pedro”

---

Sin embargo, haciendo uso del comando IN, simplificaremos mucho el código escrito, mejorando la legibilidad.

---

WHERE

nombre in (“Carlos”, “Juan”, “Luis”, “Pedro”)

## ORDER BY

Usaremos el comando cuando queramos ordenar los resultados de nuestra consulta.

Añadiremos el comando seguido de tantas columnas por las que queramos ordenar nuestros resultados.

En el caso de añadir más de una columna, el orden será jerárquico, es decir, primará en el orden la **primera columna utilizada**, en el caso de que haya casos con el mismo valor, se usará la segunda columna para decidir el orden.

Por defecto y de manera implícita, las columnas se ordenan de manera ascendente. Si quisieramos **ordenar de manera descendente**, debemos añadir la palabra reservada DESC después del nombre de la columna.

De la misma manera, si queremos expresar de manera explícita el **orden ascendente**, añadiremos la palabra reservada ASC.

## LIMIT

El comando LIMIT nos permitirá limitar los resultados de nuestra consulta. Esto puede resultar muy útil cuando queramos ver un número determinado de casos o simplemente ver unas filas para ver el contenido de las columnas, o cuando no sepamos siquiera las **columnas que existen y queremos conocerlas por primera vez**. También es útil en situaciones en las que el número de filas de la tabla es enorme y no queremos sobrecargar el sistema solicitando una gran cantidad de información.

Aquí, disponemos de un **ejemplo sobre listar usuarios**, limitándolo a los 100 primeros casos:

```
SELECT id_empleado, nombre, apellido1, apellido2
```

```
FROM empleado
```

```
LIMIT 100
```

# Conclusiones

X Edix Educación

---

Como pequeño resumen de lo comentado en este fastbook, voy a **destacar los aprendizajes clave** de este tema:

- Hemos dado un **primer vistazo a la base de datos** con la que vamos a trabajar en estos fastbooks, viendo las tablas y los campos que las componen.
- **Hemos conocido los principales comandos del sublenguaje de definición de datos** (DDL) y hemos empezado a ver el **sublenguaje de manipulación de datos** (DML).
- En lo referente al DDL, hemos estudiado los comandos necesarios para crear, modificar y eliminar tanto las bases de datos como sus tablas. También, hemos aprendido que para **modificar o borrar una base de datos** no debe tener sesiones activas.
- En lo referente al DML, hemos **descubierto las acciones de inserción, modificación y borrado**. Además, hemos puesto el foco en la sentencia 'Select', por ser una de las más importantes, y la hemos analizado en detalle.

¡Enhorabuena! Fastbook superado

edix

Creamos Digital Workers