

> Structured query language

Comunicándonos con la base de datos

SEMANA 8



INTRODUCCIÓN

¿Qué es sql?

Structured Query Language no es más que un lenguaje estándar de **comunicación con bases de datos**. Hablamos por tanto de un **lenguaje normalizado** que nos permite trabajar con cualquier tipo de lenguaje en **combinación con cualquier tipo de base de datos** (SQL Server, MySQL, ...).

El hecho de que sea estándar no quiere decir que sea idéntico para cada base de datos. En efecto, determinadas bases de datos implementan funciones específicas que no tienen necesariamente que funcionar en otras. Aparte de esta universalidad, el SQL posee otras dos características muy apreciadas. Por una parte, presenta una potencia y versatilidad notables que contrasta, por otra, con su accesibilidad de aprendizaje.

Tipos de datos de campos

Como hemos visto en páginas anteriores, una base de datos está compuesta de tablas donde almacenamos registros catalogados en función de distintos campos (características).

Un aspecto previo a considerar, es la naturaleza de los valores que introducimos en esos campos.

Dado que una base de datos trabaja con todo tipo de datos e información, es importante especificar qué tipo de valor le estamos introduciendo de manera que, por un lado, facilitemos la búsqueda posterior y por otro, optimizar los recursos de memoria.

Cada base de datos introduce tipos de valores de campo que no necesariamente están presentes en otras.

Sin embargo, existe un conjunto de tipos que están representados en la totalidad de estas bases.

Estos son los tipos más comunes:

(ver siguiente página)

| Tipo | Palabra reservada | Longitud | Descripción |
|-------------------------------|--------------------|--|--|
| Caracteres y cadenas de texto | CHAR | Hasta un máximo de 255 bytes | Cadenas de texto de longitud fija. |
| | VARCHAR | Entre 1 y 255 bytes | Cadenas de texto de longitud variable. |
| | TEXT | Varía según el SGBD (o DBMS, se mide en bytes) | Cadenas de texto de longitud variable más largas. |
| Enteros | INTEGER O INT | 4 bytes (32 bits) | Números enteros de tamaño normal. Puede almacenar valores enteros en el rango de -2,147,483,648 a 2,147,483,647. |
| | SMALLINT | 2 bytes (16 bits) | Números enteros de tamaño reducido. Puede almacenar valores enteros en el rango de -32,768 a 32,767. |
| | BIGINT | 8 bytes (64 bits) | Números enteros de tamaño grande. Puede almacenar valores enteros en el rango de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807. |
| Decimales | FLOAT | Varía según la precisión especificada (medida en byte) | Números de punto flotante con precisión simple. |
| | DOUBLE | 8 bytes (64 bits) | Números de punto flotante con precisión doble. |
| | DECIMAL | Varía según la precisión y escala especificada (medida en byte) | Números decimales de precisión fija. Se especifica la precisión total y la escala. Cuando hablamos de valores monetarios se puede usar NUMERIC, aunque usar DECIMAL, es totalmente válido también. |
| Booleanos | BOOL o BOOLEAN | 1 byte (8 bits) | Valores de verdadero o falso. |
| Fechas y horas | DATE | No tiene una longitud específica. | Fecha sin incluir la hora. |
| | TIME | No tiene una longitud específica. | Hora sin incluir la fecha. |
| | DATETIME/TIMESTAMP | Varía según el SGBD (o DBMS, se mide en bytes) | Fecha y hora combinadas. |
| Datos binarios | VARBINARY | La longitud máxima se define al declararlo (Por ejemplo: VARBINARY(50), almacena hasta 50 bytes) | Datos binarios de longitud variable. |
| | BLOB | Varía según el SGBD (o DBMS, se mide en bytes) | (Binary Large Object) Datos binarios de longitud variable o de gran tamaño. Son campos de imágenes, archivos o documentos. |

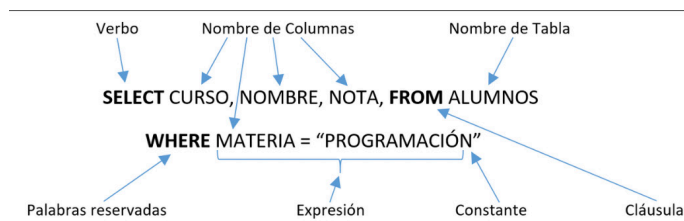
Tipos de sentencias SQL y sus componentes sintácticos

En SQL tenemos bastantes sentencias que se pueden utilizar para realizar diversas tareas.

| SENTENCIA | DESCRIPCIÓN |
|--|--|
| DML - Lenguaje de Manipulación de Datos | |
| SELECT | →Recupera datos de la base de datos (consulta). |
| INSERT | →Añade nuevas filas de datos a la base de datos. |
| DELETE | →Suprime filas de datos de la base de datos. |
| UPDATE | →Modifica datos existentes en la base de datos. |
| DDL - Lenguaje de Definición de Datos | |
| CREATE TABLE | →Añade una nueva tabla a la base de datos. |
| DROP TABLE | →Suprime una tabla de la base de datos. |
| ALTER TABLE | →Modifica la estructura de una tabla existente. |
| CREATE VIEW | →Añade una nueva vista a la base de datos. |
| DROP VIEW | →Suprime una vista de la base de datos. |
| CREATE INDEX | →Construye un índice para una columna. |
| DROP INDEX | →Suprime el índice para una columna. |
| DCL - Lenguaje de Control de Datos | |
| GRANT | →Concede privilegios de acceso a usuarios. |
| REVOKE | →Suprime privilegios de acceso a usuarios. |
| TCL - Lenguaje de Control Transaccional | |
| COMMIT | →Finaliza la transacción actual. |
| ROLLBACK | →Aborta la transacción actual. |

Componentes sintácticos

La mayoría de sentencias SQL tienen la misma estructura. Todas comienzan por un verbo (create, select, insert, update, delete), a continuación, le siguen una o más cláusulas que nos dicen los datos con los que vamos a operar (from, where), algunas de estas son opcionales y otras obligatorias como es el caso del from.



INSERCIÓN Y MODIFICACIÓN DE DATOS

Creación de tablas

Te explicaremos la manera de crear tablas a partir de sentencias SQL:

Definimos los tipos de campos principales y la forma de especificar los índices.

En general, la mayoría de las bases de datos poseen potentes editores de bases que permiten la creación rápida y sencilla de cualquier tipo de tabla con cualquier tipo de formato.

Sin embargo, una vez que la base de datos está alojada en el servidor, puede darse el caso de que necesitemos introducir una nueva tabla ya sea con carácter temporal (para gestionar un carrito de compra, por ejemplo) o bien permanente por necesidades concretas de nuestra aplicación.

En estos casos, podemos, a partir de una sentencia SQL, crear la tabla con el formato que deseemos lo cual nos puede ahorrar más de un dolor de cabeza.

Este tipo de sentencias son especialmente útiles para bases de datos como MySQL (que es la que usaremos), las cuales trabajan directamente con comandos SQL y no por medio de editores.

Para crear una tabla debemos especificar diversos datos: El nombre que le queremos asignar, los nombres de los campos y sus características. Además, será necesario especificar cuáles de estos campos van a ser índices o key y de qué tipo van a serlo.

La sintaxis de creación puede variar ligeramente de una base de datos a otra ya que los tipos de campo aceptados no están completamente estandarizados.

Sintaxis

```
create table nombre_tabla
(
    nombre_campo1 tipo (tamaño) propiedad,
    nombre_campo2 tipo (tamaño) propiedad,
    nombre_campo3 tipo (tamaño) propiedad,
    key(nombre_campo)
);
```

En esta estructura básica para la creación de una tabla podemos observar la sintaxis específica usada. Pero para analizar la creación de la misma vamos a descomponer cada parte en el cuadro de la siguiente página.

| Nombre | Descripción |
|---------------------|--|
| nombre_tabla | Es el nombre de la tabla que se va a crear. |
| nombre_campo1, 2, 3 | Es el nombre del campo o de los campos que se van a crear en la nueva tabla. La nueva tabla debe contener, al menos, un campo. |
| tipo | Es el tipo de dato del campo en la nueva tabla. (Ver tabla de Tipos de Datos comunes) |
| tamaño | Es el tamaño del campo (veremos ejemplos más adelante). |
| propiedad | Se utiliza para definir el comportamiento de un campo. |
| key (campo) | Es una cláusula CONSTRAINT que define el tipo de índice o key a crear. Aquí podremos definir las llaves primarias o foráneas de nuestra tabla. Podés ver que se utilizan (paréntesis) y , (comas). |
| A tener en cuenta | Los paréntesis servirán para encapsular todas las declaraciones que se harán sobre la tabla y sobre los campos. Las comas servirán para finalizar las declaraciones de cada campo. Al final, cuando va a ser la última declaración que realizamos, no es necesaria la coma, pero no debemos olvidar el paréntesis de cierre. Además según el SGBD que estemos utilizando, luego del último paréntesis de cierre, puede ser necesario poner un punto y coma ; |

Recalcamos que puede haber algunas diferencias según el SGBD que usemos, pero para MySQL, que es el SGBD que utilizaremos, respetando esta sintaxis, no habría inconvenientes.

Ahora veremos un ejemplo de la creación de una tabla de pedidos:

```
create table pedidos
(
    id_pedido INT(4) NOT NULL AUTO_INCREMENT,
    id_cliente INT(4) NOT NULL,
    id_articulo INT(4) NOT NULL,
    fecha DATE,
    cantidad INT(4),
    total INT(4),
    KEY(id_pedido, id_cliente, id_articulo)
);
```


Para el caso de la tabla "pedidos" creamos los campos id los cuales son considerados de tipo entero de una longitud especificada por el número entre paréntesis (4 en este caso). Para "id_pedido" requerimos que dicho campo se incremente automáticamente (AUTO_INCREMENT, esta es una propiedad que usamos sobre este campo para definir su comportamiento) de a una unidad con cada introducción de un nuevo registro para, de esta forma, automatizar su valor.

Por otra parte, para evitar un mensaje de error, es necesario requerir que los campos que van a ser definidos como claves o key NO puedan ser nulos (NOT NULL, esta es otra propiedad que usamos sobre un campo), es decir, es necesario que este tipo de campos requiera siempre tener un valor, ya que si por algún motivo pasara que no tuviese valor, habría una inconsistencia en la base de datos ya que no podríamos hacer referencia a él, por ende, no podríamos acceder al mismo.

El campo fecha es almacenado con formato de fecha (DATE) para permitir su correcta explotación a partir de las funciones previstas a tal efecto.

Finalmente, definimos la, o las claves en este caso, escribiéndolas dentro de los paréntesis, precedidas de la palabra KEY.

Del mismo modo podríamos crear la tabla de artículos con una sentencia como ésta:

```
create table articulo  
(  
  id_articulo INT (4) NOT NULL AUTO_INCREMENT,  
  titulo VARCHAR (50),  
  autor VARCHAR (25),  
  editorial VARCHAR (25),  
  precio NUMERIC,  
  KEY(id_articulo)  
);
```

En este caso puede verse que los campos alfanuméricos (VARCHAR) son introducidos de la misma forma que los numéricos. Volvemos a recordar que en tablas que tienen campos comunes es de vital importancia definir estos campos de la misma forma para el buen funcionamiento de la base de datos. Muchas son las opciones que se ofrecen al generar tablas. No vamos a tratarlas detalladamente ya que sale de lo estrictamente práctico.

Estructuras de las tablas en SQL

Una base de datos en un sistema relacional está compuesta por un conjunto de tablas, que corresponden a las relaciones del modelo relacional.

En la terminología usada en SQL no se alude a las relaciones, del mismo modo que no se usa el término atributo, pero sí la palabra columna, y no se habla de tupla, sino de línea o fila. Así como lo venimos viendo en los ejemplos, donde creamos nuestras tablas y cada campo es una columna de la tabla y, todos los valores que ingresemos en nuestras tablas formarán las filas, a las cuales nos podemos referir como registros. La expresión que solemos usar para hacer referencia a la obtención de datos de la base de datos será: "vamos a realizar una consulta de la tabla", y nos estaremos refiriendo a uno o más registros.

Cláusulas CONSTRAINT

Una CONSTRAINT (restricción) es una regla o condición que se aplica a los datos almacenados en una base de datos para mantener su integridad y consistencia. Las constraints se utilizan para definir y aplicar reglas de validación sobre los datos, o como dijimos antes, definir el comportamiento de un campo, asegurando que cumplan ciertas condiciones predefinidas.

Existen varios tipos de constraints que se pueden aplicar a las tablas de una base de datos, entre ellos:

ØPrimary Key (Clave primaria): Es una constraint que identifica de manera única a cada fila de una tabla. Se utiliza para asegurar que no haya duplicados en el valor de una columna específica y es fundamental para establecer relaciones entre tablas. Recordá que estamos hablando de los IDs.

Te mostraremos un ejemplo (con una tabla Clientes) con otra forma de declarar la PK, diferente a los ejemplos anteriores:

```
CREATE TABLE Clientes (  
    ID INT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    Email VARCHAR(50)  
);
```

ØForeign Key (Clave foránea): Es una constraint que establece una relación entre dos tablas. La foreign key hace referencia a la primary key de otra tabla y asegura que los valores en la columna referenciada existan en la tabla relacionada. Como lo estuvimos viendo en MER y en tablas, cuando tenemos que hacer una referencia, aparte del propio ID de la tabla, a otra entidad, usamos el id propio de la entidad como clave primaria y un id propio de otra entidad pero que, al pasar a otra tabla, este id se convierte en FK.

```
CREATE TABLE Pedidos (  
    PedidoID INT PRIMARY KEY,  
    ClienteID INT,  
    FechaPedido DATE,  
    FOREIGN KEY (ClienteID) REFERENCES Clientes(ID)  
);
```

ØUnique (Único): Es una constraint que asegura que los valores en una columna sean únicos. A diferencia de la primary key, no impone la restricción de que todos los valores sean distintos de NULL.

Un ejemplo para este caso sería el DNI:

```
CREATE TABLE Empleados (  
    ID INT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    DNI VARCHAR(10) UNIQUE  
);
```

ØNot Null (No Nulo): Es una constraint que asegura que un campo o columna no tenga valores nulos. Obliga a que se ingrese un valor válido en esa columna.

Verás un ejemplo con una tabla de Productos:

```
CREATE TABLE Productos (  
    ID INT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    Precio DECIMAL(10, 2) NOT NULL  
);
```

ØCheck (Verificación): Es una constraint que permite definir una condición que los valores de una columna deben cumplir. Solo se permitirán los valores que satisfagan la condición definida.

En este ejemplo, una tabla de Estudiantes y una condición que le podríamos dar con esta constrain:

```
CREATE TABLE Estudiantes (  
    ID INT PRIMARY KEY,  
    Nombre VARCHAR(50),  
    Edad INT CHECK (Edad >= 18)  
);
```

Estas constraints ayudan a mantener la integridad y consistencia de los datos en la base de datos, evitando la inserción de datos inválidos o inconsistentes. Además, garantizan que se cumplan las reglas establecidas y las relaciones entre las tablas.

Creación de índices

Estas estructuras mejoran la eficiencia y la velocidad de las consultas al permitir un acceso más rápido a los datos. Su función principal es acelerar la recuperación de información al crear una referencia ordenada de los valores de una o varias columnas en una tabla. Sin embargo, la creación de índices no es obligatoria.

¿Cuál es la utilidad?

Ø Mejora del rendimiento de las consultas: Los índices permiten que las consultas se ejecuten más rápidamente al proporcionar un acceso directo a los registros relevantes. Al utilizar un índice, el motor de la base de datos puede encontrar los datos solicitados de manera más eficiente, evitando tener que examinar todas las filas de una tabla.

Ø Optimización de la búsqueda: Los índices agilizan las operaciones de búsqueda y recuperación de datos. Al crear índices en columnas comunes utilizadas en consultas frecuentes, se reducen los tiempos de búsqueda y se mejora la experiencia del usuario.

Ø Ordenación de los datos: Los índices ordenan los valores de las columnas en una tabla, lo que facilita la clasificación ascendente o descendente de los resultados de las consultas. Esto es especialmente útil cuando se necesita recuperar datos en un orden específico sin tener que realizar una clasificación en tiempo de ejecución.

Ø Restricciones de unicidad y clave primaria: Los índices también se utilizan para garantizar la unicidad de los valores en una columna o conjunto de columnas mediante la creación de índices únicos o de clave primaria. Esto evita la duplicación de datos y garantiza la integridad de los registros en una tabla.

Sin embargo, la otra cara de la moneda es que, aunque los índices mejoran la velocidad de las consultas, también pueden afectar negativamente el rendimiento de las operaciones de inserción, actualización y eliminación de registros, ya que los índices deben actualizarse junto con los datos. Por lo tanto, es necesario equilibrar cuidadosamente la creación de índices en función de las necesidades de consulta y las operaciones de manipulación de datos en una base de datos.

Algunos ejemplos son:

```
CREATE INDEX indx_nombre ON Empleados (nombre);
```

1. Este ejemplo crea un índice llamado "indx_nombre" en la tabla "Empleados" basado en la columna "nombre". El índice agilizará las consultas que involucren la columna "nombre" en la tabla "Empleados".

```
CREATE INDEX indx_empleados_ubicacion ON Empleados (ciudad, provincia);
```

2. Aquí se crea un índice compuesto llamado "indx_empleados_ubicacion" en la tabla "Empleados" basado en las columnas "ciudad" y "provincia". Este índice permite realizar consultas eficientes que involucren ambas columnas, como buscar clientes por ciudad y provincia simultáneamente.

Más adelante, veremos cómo realizar consultas con distintos comandos, pero te recordamos nuevamente que el mal uso de índices puede ser perjudicial para la base de datos, por lo que hay que ser cuidadosos al usarlo.

Modificar/eliminar tablas

Estos comandos son utilizados para realizar modificaciones en la estructura de una tabla o eliminar una tabla de nuestra base de datos.

1. ALTER TABLE:

El comando "ALTER TABLE" se utiliza para modificar la estructura de una tabla existente en una base de datos. Algunos ejemplos comunes de las operaciones que se pueden realizar con "ALTER TABLE" incluyen agregar, modificar o eliminar columnas, cambiar el tipo de datos de una columna, agregar restricciones como claves primarias, entre otros.

Un ejemplo del uso de ALTER TABLE para agregar una columna a una tabla son:

```
ALTER TABLE Empleados  
ADD COLUMN domicilio alternativo VARCHAR(50);
```

Este comando agrega una columna llamada "domicilio alternativo" a la tabla "Empleados" con el tipo de dato VARCHAR y una longitud máxima de 50 caracteres.

2. DROP TABLE:

El comando "DROP TABLE" se utiliza para eliminar una tabla completa de una base de datos, junto con todos sus datos y metadatos asociados. Es importante tener precaución al usar este comando, ya que los datos de la tabla se eliminarán permanentemente y no se podrán recuperar.

Se recomienda hacer una copia de seguridad de los datos importantes antes de ejecutar "DROP TABLE".

Un ejemplo del uso de DROP TABLE para eliminar una tabla es:

```
DROP TABLE Empleados;
```

Este comando elimina la tabla "Empleados" de la base de datos junto con todos sus datos y definiciones.

Añadir un nuevo registro

Para añadir un nuevo registro a una tabla en SQL, se utiliza el comando "INSERT INTO".

La sintaxis básica de cómo utilizarlo es:

```
INSERT INTO nombre_de_tabla (columna1, columna2, columna3, ...)  
VALUES (valor1, valor2, valor3, ...);
```

1. Especificamos el nombre de la tabla a la que deseamos añadir el registro después del comando "INSERT INTO".

2. Después del nombre de la tabla, enumeramos entre paréntesis los nombres de las columnas en las que deseamos insertar valores. Si queremos insertar valores en todas las columnas, podemos omitir esta parte.

3. Utilizamos la palabra clave "VALUES" y entre paréntesis los valores que deseamos insertar en las respectivas columnas. Vamos a asegurarnos de que el orden de los valores coincida con el orden de las columnas enumeradas anteriormente.

Un ejemplo de su uso es:

Supongamos que tenemos una tabla llamada "clientes" con las columnas "id_cliente", "nombre" y "email". Para añadir un nuevo cliente a esta tabla, podemos utilizar el siguiente comando:

```
INSERT INTO clientes (id_cliente, nombre, email)  
VALUES (1, 'Infor Matorio', 'informatario@chaco.gob.ar');
```

En este ejemplo, se inserta un nuevo registro con un "id_cliente" de 1, nombre "Infor Matorio" y email "informatario@chaco.gob.ar" en la tabla "clientes".

Recordamos que los valores que se inserten deben estar en el tipo de dato correcto y cumplir con las restricciones definidas en la estructura de la tabla, como claves primarias, etc.

Además, si la tabla tiene columnas con valores predeterminados o que permiten nulos, puedes omitir la lista de columnas y simplemente proporcionar los valores correspondientes en el orden correcto.

También en este ejemplo, se puede ver que los campos con valores numéricos no van con las comillas "", en el caso de número id. Sin embargo, si nuestro campo id es autoincremental (es decir, al crear el campo lo definimos así), deberíamos modificar el ejemplo anterior, de la siguiente forma:

```
INSERT INTO clientes (nombre, email)
VALUES ('Infor Matorio', 'informatario@chaco.gob.ar');
```

En el ejemplo, no incluimos la columna id, ni pusimos nada en los valores. La base de datos se encargará automáticamente de generar un valor único para el campo id. Lo importante a destacar es que la columna id debe estar configurada como AUTO_INCREMENT al crear la tabla.

Borrar un registro

Para borrar un registro de una tabla en SQL, se utiliza el comando "DELETE". Esta sentencia permite eliminar uno o varios registros que cumplan con una condición.

La sintaxis básica de cómo utilizarlo es:

```
DELETE FROM nombre_de_tabla WHERE condicion;
```

Un ejemplo de uso es:

```
DELETE FROM clientes WHERE id = 1;
```

Esta sentencia eliminará el registro de la tabla "usuarios" que tenga un ID igual a 1. Podemos observar además, un nuevo comando que nos va a servir mucho cuando tengamos que dar condiciones. El comando es WHERE.

Algo a tener en cuenta y cuidado, es que, si usamos la sentencia DELETE sin condiciones, podemos borrar todo el contenido de la tabla, a diferencia de DROP TABLE que borra la tabla, si usamos la sentencia DELETE sin condiciones, borramos todos los registros, es decir todos los valores, pero la tabla aún queda.

```
DELETE FROM clientes;
```

Actualizar un registro

Para actualizar un registro en una tabla en SQL, se utiliza el comando "UPDATE". Esta sentencia permite modificar los valores de una o varias columnas en un registro existente que cumpla con una condición específica.

La sintaxis básica es:

```
UPDATE nombre_de_tabla SET columna1 = valor1, columna2 = valor2 WHERE condicion;
```

Un ejemplo es:

```
UPDATE clientes SET nombre = 'Informatario Chaco' WHERE nombre = 'Infor Matorio';
```

Mediante esta sentencia cambiamos el nombre 'Infor Matorio' por el de 'Informatario Chaco' en todos los registros cuyo nombre sea 'Infor Matorio'. Aquí también hay que ser cuidadoso de no olvidarse de usar WHERE, de lo contrario, modificaríamos todos los registros de nuestra tabla. Es decir, en este caso, sin el uso de WHERE cambiaríamos todos los nombres sin importar nada, a 'Informatario Chaco'.

Búsqueda y selección de datos en SQL

Para comenzar con las consultas seguiremos con el ejemplo de la tabla clientes, con los siguientes campos:

| CLIENTES | |
|--------------|-------------|
| nombre | VARCHAR(30) |
| apellidos | VARCHAR(30) |
| direccion | VARCHAR(50) |
| provincia | VARCHAR(25) |
| codigopostal | INT(4) |
| email | VARCHAR(30) |
| pedidos | INT(3) |

Selección de tablas - SELECT

La selección total o parcial de una tabla se lleva a cabo mediante la instrucción **SELECT**.

En dicha selección hay que especificar:

- Los campos que queremos seleccionar
- La tabla en la que haremos la selección

Para comenzar con esta consulta o Query, utilizaremos nuestra tabla de ejemplo:

```
SELECT nombre, direccion FROM clientes;
```

Ø Se puede apreciar que mediante el comando **SELECT**, se llaman a todos los datos que haya en los campos "nombre" y "direccion", desde (**FROM**) la tabla "clientes".

Ø Pero si quisiéramos todos los registros de la tabla clientes, se podría utilizar el asterisco en lugar de nombrar las columnas de la tabla:

```
SELECT * FROM clientes;
```

Ø Si necesitáramos filtrar, por ejemplo, por un nombre de provincia. Vamos a suponer que queremos filtrar para ver los clientes que son solo de Chaco. Entonces, podemos usar **WHERE** y **LIKE**:

```
SELECT * FROM clientes WHERE provincia LIKE 'Chaco';
```

Ø Además, si necesitáramos ordenar los resultados, en función de uno o varios campos. Por ejemplo, queremos que nuestra consulta de los clientes de Chaco, esté ordenada por el nombre de cada cliente. Para ello usaremos **ORDER BY**:

```
SELECT * FROM clientes WHERE provincia LIKE 'Chaco' ORDER BY nombre;
```

Ø También se podría ordenar por apellido:

```
SELECT * FROM clientes WHERE provincia LIKE 'Chaco' ORDER BY nombre, apellido;
```

Ø Si invirtiésemos el orden « nombre, apellido » por « apellido, nombre », el resultado sería distinto. Tendríamos los clientes ordenados por apellido y aquellos que tuviesen apellidos idénticos se subclasificarían por el nombre.

Es posible también ordenar por orden inverso. Si por ejemplo quisiésemos ver nuestros clientes por orden de pedidos realizados teniendo a los mayores en primer lugar, usaríamos el comando **DESC**:

```
SELECT * FROM clientes ORDER BY pedidos DESC;
```

Ø Una opción interesante es la de efectuar selecciones sin coincidencia. Si, por ejemplo, buscásemos saber en qué provincias se encuentran nuestros clientes sin necesidad de que para ello aparezca varias veces la misma provincia usaríamos una sentencia **DISTINCT**:

```
SELECT DISTINCT provincia FROM clientes ORDER BY provincia;
```

Así evitaríamos ver repetido Chaco tantas veces como clientes tengamos en esa provincia.

Lista de operadores más comunes para realizar selecciones

Además de los operadores matemáticos, lógicos que ya hemos visto en Python, podemos usar otros operadores propios para SQL. Algunos ya se han mostrado en ejemplos anteriores. Estos operadores serán utilizados después de la cláusula WHERE y pueden ser combinados mediante paréntesis para optimizar nuestra query.

| Comando | Descripción |
|-----------------------|---|
| LIKE | Selecciona los registros cuyo valor de campo se asemeje, no teniendo en cuenta mayúsculas y minúsculas. |
| IN Y NOT IN | Da un conjunto de valores para un campo para los cuales la condición de selección es (o no) válida. |
| IS NULL Y IS NOT NULL | Selecciona aquellos registros donde el campo especificado esta (o no) vacío. |
| BETWEEN... AND | Selecciona los registros comprendidos en un intervalo. |
| DISTINCT | Selecciona los registros no coincidentes. |
| DESC | Clasifica los registros por orden inverso. |
| * | Se utiliza para incluir <u>todo</u> en los campos. |
| % | Sustituye a cualquier cosa o nada dentro de una cadena. |
| _ (guión bajo) | Sustituye un solo carácter dentro de una cadena. |

Algunos ejemplos para ver el uso de estos operadores son:

Ø En esta sentencia seleccionamos todos los clientes de Chaco cuyo nombre no es Informatorio.

Para este caso podríamos usar el operador matemático = pero vamos a usar LIKE, ya que por convención es una buena práctica usar el comando LIKE, en vez de = para obtener el resultado buscado.

```
SELECT * FROM clientes WHERE provincia LIKE 'Chaco' AND NOT nombre LIKE 'Informatorio';
```

Ø Si quisiéramos una selección de los clientes de nuestra tabla cuyo apellido comienza por A y cuyo número de pedidos está comprendido entre 20 y 40, podemos usar BETWEEN...AND:

```
SELECT * FROM clientes WHERE apellidos LIKE 'A%' AND pedidos BETWEEN 20 AND 40;
```

Ø El operador IN, es muy práctico para consultas en varias tablas. Para casos en una sola tabla es empleado del siguiente modo:

```
SELECT * FROM clientes WHERE provincia IN ('Chaco', 'Corrientes', 'Misiones');
```

De esta forma seleccionamos aquellos clientes que vivan en esas tres provincias.

Realizar selecciones sobre varias tablas

Una base de datos puede ser considerada como un conjunto de tablas. Estas tablas en muchos casos están relacionadas entre ellas y se complementan unas con otras. De nuestra tabla clientes, en la cual..

tenemos la columna "pedidos", podríamos dividirla y hacer una tabla de pedidos exclusiva, donde se podrían tener más detalles de los mismos. Esta tabla de pedidos, a su vez, puede estar relacionada con una tabla de artículos, donde pueden estar los artículos con los detalles de estos. Con esto, podríamos obtener el detalle completo, desde los datos del cliente, los datos del pedido hasta los datos del artículo. Es decir, si queremos obtener toda la información referida, lo podemos consultar a estas tres tablas.

Podríamos, por ejemplo, obtener la designación del artículo más popular en una determinada región, donde la designación del artículo sería obtenida de la tabla artículos, la popularidad (cantidad de veces que ese artículo se vendió) vendría de la tabla pedidos y la región estaría comprendida en la tabla clientes.

Este tipo de organización basada en múltiples tablas conectadas nos permite trabajar con tablas mucho más manejables a la vez que nos evita copiar el mismo campo en varios sitios ya que podemos acceder a él a partir de una simple consulta a la tabla que lo contiene.

Supongamos que queremos enviar un mail con una oferta a todos aquellos que hayan realizado un pedido ese mismo día.

–Ver consulta en imagen 1

IMAGEN 1

```
SELECT clientes.apellidos, clientes.email FROM clientes, pedidos WHERE pedidos.fecha LIKE '01/01/23' AND pedidos.id_cliente = clientes.id_cliente;
```

IMAGEN 2

```
SELECT pedidos.id_pedido, articulos.titulo FROM pedidos, articulos WHERE pedidos.id_articulo LIKE articulos.id_articulo;
```

Se puede observar que después de la cláusula FROM, introducimos el nombre de las dos tablas de donde sacamos los datos. Además, el nombre de cada campo va precedido de la tabla de origen separados ambos por un punto. En los campos que poseen un nombre que solo aparece en una de las tablas, no es necesario especificar su origen, aunque a la hora de leer la sentencia puede resultar más claro el precisarlo. En este caso, el campo fecha podría haber sido designado como "fecha" en lugar de "pedidos.fecha".

Un último ejemplo. Esta vez, deseamos ver el título del libro correspondiente a cada uno de los pedidos realizados: ver imagen 2

Podemos observar que la consulta, es básicamente, la misma que para la consulta de una única tabla.

El empleo de funciones para la explotación de los campos numéricos y otras utilidades.

Además de los criterios hasta ahora explicados para realizar las consultas en tablas, SQL permite también aplicar un conjunto de funciones predefinidas. Estas funciones, aunque básicas, pueden ayudarnos en algunos momentos a expresar nuestra selección de una manera más simple sin tener que recurrir a operaciones adicionales. Algunas de estas funciones son representadas en la tabla siguiente:

| Función | Descripción |
|------------|---|
| Sum(campo) | Calcula la suma de los registros del campo especificado. |
| Avg(Campo) | Calcula la media de los registros del campo especificado. |
| Count(*) | Nos da el número de registros que han sido contados. |
| Max(Campo) | Nos indica cual es el valor máximo del campo. |
| Min(Campo) | Nos indica cual es el valor mínimo del campo. |

Ø Dado que el campo de la función no existe en la base de datos, sino que lo estamos generando virtualmente, esto puede crear inconvenientes cuando estamos trabajando con nuestros comandos al momento de tratar su valor y nombre de campo. Es por ello que el valor de la función ha de ser recuperada a partir de un alias que nosotros especificaremos en la sentencia SQL a partir de la instrucción AS. Además, ocuparemos SUM():

```
SELECT SUM(total) AS suma_pedidos FROM pedidos;
```

A partir de esta sentencia calculamos la suma de los valores de todos los pedidos realizados y almacenamos ese valor en un campo virtual llamado "suma_pedidos" que podrá ser utilizado como cualquier otro campo por nuestras páginas dinámicas.

Por supuesto, todo lo visto hasta ahora puede ser aplicado en este tipo de funciones de modo que, por ejemplo, podríamos establecer condiciones con la cláusula WHERE construyendo sentencias como esta:

```
SELECT SUM(cantidad) AS suma_articulos FROM pedidos WHERE id_articulo=6;
```

Esto nos proporcionaría la cantidad de ejemplares de un determinado libro que han sido vendidos.

Ø Otra propiedad interesante de estas funciones es que permiten realizar operaciones con varios campos dentro de un mismo paréntesis con AVG():

```
SELECT AVG(total/cantidad) FROM pedidos;
```

Esta sentencia da como resultado el precio medio al que se están vendiendo los libros. Este resultado no tiene por qué coincidir con el del precio medio de los libros presentes en el inventario, ya que, puede ser que la gente tenga tendencia a comprar los libros caros o los baratos:

```
SELECT AVG(precio) AS precio_venta FROM artículos;
```

Ø Una cláusula interesante en el uso de funciones es GROUP BY. Esta cláusula nos permite agrupar registros a los cuales vamos a aplicar la función. Podemos, por ejemplo, calcular el dinero gastado por cada cliente:

```
SELECT id_cliente, SUM(total) AS suma_pedidos FROM pedidos GROUP BY id_cliente;
```

Ø O saber el número de pedidos que han realizado:

```
SELECT id_cliente, COUNT(*) AS numero_pedidos FROM pedidos GROUP BY id_cliente;
```

Ø O saber el mínimo o máximo con MIN, MAX:

```
SELECT MIN(gastos) AS elminimo FROM pedidos WHERE provincia = 'Chaco';
SELECT MAX(gastos) AS elmaximo FROM pedidos WHERE provincia = 'Corrientes';
```

Devolver literales

En determinadas ocasiones nos puede interesar incluir una columna con un texto fijo en una consulta de selección, por ejemplo, de tabla de empleados deseamos recuperar las tarifas semanales de los electricistas, podríamos realizar la siguiente consulta:

```
SELECT Empleados.Nombre, 'Tarifa semanal: ', Empleados.TarifaHora * 40 FROM Empleados WHERE Empleados.Cargo = 'Electricista';
```

Uso de índices de las tablas

Si deseamos que la sentencia SQL utilice un índice para mostrar los resultados se puede utilizar la palabra reservada INDEX de la siguiente forma:

```
SELECT ... FROM Tabla (INDEX=Indice) ...
```

Normalmente los motores de las bases de datos deciden que índice se debe utilizar para la consulta, para ello utilizan criterios de rendimiento y sobre todo los campos de búsqueda especificados en la cláusula WHERE. Si se desea forzar a no utilizar ningún índice utilizaremos la siguiente sintaxis:

```
SELECT ... FROM Tabla (INDEX=0) ...
```

Otros ejemplos usando Operadores Lógicos y Matemáticos

Si deseamos seleccionar rangos específicos o dar ciertas condiciones, como en los siguientes casos:

```
SELECT * FROM Empleados WHERE edad > 25 AND edad < 50;  
SELECT * FROM Empleados WHERE (edad > 25 AND edad < 50) OR sueldo = 100;  
SELECT * FROM Empleados WHERE NOT estado_civil = 'Soltero';  
SELECT * FROM Empleados WHERE (sueldo > 100 AND sueldo < 500) OR (provincia = 'Chaco' AND estado_civil = 'Casado');
```

Subconsultas

Una subconsulta es una instrucción SELECT anidada dentro de una instrucción SELECT, SELECT...INTO, INSERT...INTO, DELETE, o UPDATE o dentro de otra subconsulta.

Se puede utilizar tres formas de sintaxis para crear una subconsulta:

```
comparación [ANY | ALL | SOME] (instrucción sql)  
expresión [NOT] IN (instrucción sql)  
expresión [NOT] EXISTS (instrucción sql)
```

En donde:

| | |
|-----------------|---|
| Comparación | Es una expresión y un operador de comparación que compara la expresión con el resultado de la subconsulta. |
| Expresión | Es una expresión por la que se busca el conjunto resultante de la subconsulta. |
| Instrucción SQL | Es una instrucción SELECT, que sigue el mismo formato y reglas que cualquier otra instrucción SELECT. Debe ir entre paréntesis. |

Se puede utilizar una subconsulta en lugar de una expresión en la lista de campos de una instrucción SELECT o en una cláusula WHERE o HAVING. En una subconsulta, se utiliza una instrucción SELECT para proporcionar un conjunto de uno o más valores especificados para evaluar en la expresión de la cláusula WHERE o HAVING.

Ø Se puede utilizar el comando ANY o SOME, los cuales son sinónimos, para recuperar registros de la consulta principal, que satisfagan la comparación con cualquier otro registro recuperado en la subconsulta. El ejemplo que veremos devuelve todos los productos cuyo precio unitario es mayor que el de cualquier producto vendido con un descuento igual o mayor al 25 por ciento: **Ver imagen 1**

El comando ALL se utiliza para recuperar únicamente aquellos registros de la consulta principal que satisfacen la comparación con todos los registros recuperados en la subconsulta. Si se cambia ANY por ALL en el ejemplo anterior, la consulta devolverá únicamente aquellos productos cuyo precio unitario sea mayor que el de todos los productos vendidos con un descuento igual o mayor al 25 por ciento. Esto es mucho más restrictivo.

Ø El comando IN se emplea para recuperar únicamente aquellos registros de la consulta principal para los que algunos registros de la subconsulta contienen un valor igual.

El siguiente ejemplo devuelve todos los productos vendidos con un descuento igual o mayor al 25 por ciento: **Ver imagen 2.**

Inversamente se puede utilizar NOT IN para recuperar únicamente aquellos registros de la consulta principal para los que no hay ningún registro de la subconsulta que contenga un valor igual.

Ø El comando EXISTS (con la palabra reservada NOT opcional) se utiliza en comparaciones de verdadero/falso para determinar si la subconsulta devuelve algún registro. Supongamos que deseamos recuperar todos aquellos clientes que hayan realizado al menos un pedido: **Ver imagen 3**

IMAGEN 1

```
SELECT * FROM Productos WHERE preciounidad ANY (SELECT preciounidad FROM detallepedido WHERE descuento = 0.25);
```

IMAGEN 2

```
SELECT * FROM Productos WHERE id_producto IN (SELECT id_producto FROM detallepedido WHERE descuento = 0.25);
```

IMAGEN 3

```
SELECT Clientes.compañía, Clientes.teléfono FROM Clientes WHERE EXISTS (SELECT id_pedido FROM pedidos WHERE pedidos.id_pedido = Clientes.id_cliente);
```

Creación de la Base de Datos

Por último, vamos al comienzo de todo. Este es el comando para crear nuestra base de datos y luego aplicar todo el conocimiento que has adquirido hasta aquí.

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] nombre_base_datos;
```

Ø DATABASE y SCHEMA son sinónimos.

Ø IF NOT EXISTS crea la base de datos sólo si no existe una base de datos con el mismo nombre.

```
CREATE SCHEMA IF NOT EXISTS Empresa1;
```

Eliminar una Base de Datos

```
DROP {DATABASE | SCHEMA} [IF EXISTS] nombre_base_datos;
```

Ø DATABASE y SCHEMA son sinónimos.

Ø IF EXISTS elimina la base de datos sólo si existe.

```
DROP DATABASE Empresa1;
```