

¿QUÉ ES SQLITE?

SQLite es un gestor de bases de datos relacional pero con objetivos muy diferentes a los gestores como MySQL, SQL Server, Oracle, PostgreSQL etc.

Este gestor de base de datos tiene por objetivo ser parte de la misma aplicación con la que colabora, es decir no cumple los conceptos de cliente y servidor.

Para entender sus usos podemos dar algunos ejemplos donde se utiliza el gestor SQLite:

- Firefox usa SQLite para almacenar los favoritos, el historial, las cookies etc.
- También los navegadores Opera y Chrome usan SQLite.
- La aplicación de comunicaciones Skype de Microsoft utiliza SQLite
- Los sistemas operativos Android y iOS adoptan SQLite para permitir el almacenamiento y recuperación de datos.
- Podemos conocer otras empresas famosas que hacen uso de SQLite visitando el sitio oficial de SQLite.
- SQLite se encuentra en miles de millones de dispositivos Android, iOS, Mac, Windows 10 etc.

Es decir SQLite colabora con el almacenamiento de datos cuando hacemos aplicaciones con lenguajes como Python, Java, C#, C, C++, Delphi, etc.

SQLite es Open Source y su sitio oficial es: sqlite.org

SQLite se maneja totalmente por terminal, existen varios visualizadores que pueden usarse a modo didáctico para tener control visual de nuestros ejercicios. Pero en este curso lo abordaremos totalmente por terminal para que seamos capaces de manejar una db aun en entornos donde no tengamos interfaces gráficas.

INSTALAR SQLITE

Todos los archivos necesarios para la instalación se encuentran en su página oficial. Pero si estas en una distribución de linux basada en debian solo debes poner el siguiente comando en la terminal:

```
sudo apt-get install sqlite3
```

Para asegurarnos que esté instalado y funcionando correctamente probamos el siguiente comando:

```
sqlite3
```

y deberíamos obtener algo parecido a lo siguiente:

```
$ Sqlite3
```

```
SQLite versión 3.7.15.2 01/09/2013 11:53:05
Enter ".help" para obtener instrucciones
Introduzca las sentencias SQL terminado con un ";"

sqlite>
```

SINTAXIS

SQLite se maneja casi enteramente por comandos, una sentencia esta compuesta por varios comandos, al final de cada sentencia debemos poner siempre un ";". Por convención los comandos se escriben en mayúsculas y los nombres de variables en minúsculas con snake-case.

Ejemplo de una sentencia que crea una tabla:

```
CREATE TABLE usuarios (
    nombre TEXT,
    clave TEXT
);
```

Es perfectamente factible introducir la sentencia en formato lineal:

```
CREATE TABLE usuarios (nombre TEXT, clave TEXT );
```

SQLite nos permitirá introducir saltos de línea con la tecla enter e introduce la sentencia sólo cuando detecte un ";".

CREACIÓN DE UNA DB

El mismo comando que usamos para comprobar la instalación de SQLite nos servirá para crear una DB.

```
$ Sqlite3 DatabaseName.db
```

Este comando nos permitirá acceder a una db con el nombre que le especifiquemos en el primer atributo o la creará si no existe.

Ejemplo:

Si desea crear una nueva base de datos <testDB.db>, la sentencia es la siguiente:

```
$ Sqlite3 testDB.db
SQLite versión 3.7.15.2 01/09/2013 11:53:05
```

Escriba ".help" para obtener instrucciones
Introduzca las sentencias SQL terminado con un ";"

```
sqlite>
```

El comando anterior creará un archivo "testDB.db" en el directorio actual. El documento será utilizado como la base de datos a trabajar. Si nota la siguiente línea "sqlite>" al finalizar quiere decir que la db ha sido creada/accedido correctamente.

Una vez creada la base de datos, puede utilizar el comando SQLite .databases para comprobar si está en la lista de bases de datos, de la siguiente manera:

```
sqlite> .databases
```

```
0 /home/sqlite/testDB.db
```

Nota: Para salir de SQLite usamos el comando .quit

COMANDO .dump

Puede exportar la totalidad de la base de datos en un archivo de texto, como se muestra a continuación utilizando sentencias de punto de SQLite.

```
$ Sqlite3 testDB.db .dump> testDB.sql
```

El comando anterior convierte el contenido de toda la base de datos SQLite testDB.db, y lo descarga en un archivo de texto ASCII en testDB.sql.

Para recuperar una DB de un archivo SQL podemos usar la siguiente sentencia:

```
$ Sqlite3 testDB.db <testDB.sql
```

Esto solo debe realizarse si el archivo de base de datos esta vacío, realizar esto con una db en uso puede derivar en corrupción de datos!!

TIPOS DE DATOS EN SQLite

Si vienes de otros dbms como MySQL o PostgreSQL, habrás notado que usan tipado estático. Esto quiere decir que tu declaras una columna con un tipo específico de dato, y esa columna sólo puede almacenar datos del tipo declarado.

A diferencia de estas dbms, SQLite es un sistema de tipos dinámicos. En otras palabras, el valor de un dato almacenado en una columna determina el tipo de dato, y no el tipo de dato de la columna.

Adicionalmente, no es necesario declarar un tipo de dato específico para una columna cuando creas una tabla. En caso que declares una columna como tipo de dato INTEGER, aun puedes almacenar cualquier tipo de dato como texto y BLOB, SQLite no se hace problema con esto.

SQLite provee 5 tipos de datos primitivos que son referidos como clases de almacenamiento.

Las clases de almacenamiento describen el formato que SQLite usa para almacenar datos en el disco. Una clase de almacenamiento es mas general que un tipo de dato. Ej: la clase de almacenamiento INTEGER incluye 6 tipos de integers. En muchos casos podrás usar clases de almacenamiento y tipos de datos al mismo tiempo.

La siguiente tabla muestra las 5 clases de almacenamiento en SQLite:

Clase de almacenamiento	Descripción
NULL	Los valores NULL se refieren a información faltante o desconocida.
INTEGER	Los valores INTEGER son la mayoría de los números (sean positivos o negativos). Un INTEGER puede tener tamaños variables como 1, 2, 3, 4, o 8 bytes.
REAL	Los valores REAL son los números reales con valores decimales que usan valores flotantes de 8-bytes.
TEXT	TEXT es usado para almacenar datos del tipo carácter. La máxima longitud para TEXT es ilimitada. SQLite soporta varias codificaciones de caracteres.
BLOB	BLOB está destinado para objetos binarios grandes y puede almacenar cualquier tipo de dato. El máximo tamaño para la clase BLOB es en teoría ilimitado.

TIPO DE DATO BOOLEANO

SQLite no tiene una clase de almacenamiento especial para los datos del tipo booleano. Por lo que el valor booleano se almacena como un número entero de 0 (falso) y 1 (verdadero).

TIPOS DE DATOS DE TIEMPO

SQLite tampoco tiene una clase designada para este tipo de datos, por lo que existen 3 convenciones para almacenar datos de tipo tiempo, en TEXT, REAL, INTEGER.

Nosotros almacenaremos todo en TEXT usando el siguiente formato:

YYYY-MM-DD HH:MM:SS.SSS

Para mas información sobre las otras 2 formas de almacenamiento y cómo trabajar con ellas visitar [este enlace](#).

CREACIÓN Y BORRADO DE UNA TABLA

Una base de datos almacena sus datos en tablas.

Una tabla es una estructura de datos que organiza los datos en columnas y filas; cada columna es un campo (o atributo) y cada fila, un registro. La intersección de una columna con una fila, contiene un dato específico, un solo valor.

- Cada registro contiene un dato por cada columna de la tabla.
- Cada campo (columna) debe tener un nombre. El nombre del campo hace referencia a la información que almacenará.
- Cada campo (columna) también debe definir el tipo de dato que almacenará: números enteros, números reales, caracteres etc.

usuarios:

nombre	clave
Mario Perez	Marito
Maria Garcia	Mary
Diego Rodriguez	z8080

Gráficamente acá tenemos la tabla usuarios, que contiene dos campos llamados: nombre y clave. Luego tenemos tres registros almacenados en esta tabla, el primero almacena en el campo nombre el valor "Mario Perez" y en el campo clave "Marito", y así sucesivamente con los otros dos registros.

Las tablas forman parte de una base de datos.

Toda tabla debe ser definida con un nombre que la identifique y con el cual accederemos a ella.

Creamos una tabla llamada "usuarios", escribiendo:

CREATE TABLE usuarios (

```
    nombre TEXT,  
    clave TEXT  
);
```

Para visualizar el esquema de una tabla debemos usar el comando `.scheme nombreDeLaTabla`.

Si intentamos crear una tabla con un nombre ya existente (existe otra tabla con ese nombre), mostrará un mensaje de error indicando que la acción no se realizó porque ya existe una tabla con el mismo nombre.

Cuando se crea una tabla debemos indicar su nombre y definir sus campos con su tipo de dato. En esta tabla "usuarios" definimos 2 campos:

nombre: que contendrá una cadena de caracteres, que almacenará el nombre de usuario

clave: otra cadena de caracteres que guardará la clave de cada usuario.

Cada usuario ocupará un registro de esta tabla, con su respectivo nombre y clave.

Para eliminar una tabla usamos el comando "drop table" junto al nombre de la tabla. Escribimos:

```
drop table usuarios;
```

RESTRICCIONES

Las restricciones son reglas especiales que se especifican en los atributos al momento de crear una tabla. Estos se utilizan para limitar que puede insertarse en una sección de la tabla. Esto asegura la exactitud y fiabilidad de los datos en la base de datos.

Las restricciones pueden ser a nivel de columna o tabla. La limitación de niveles de columna sólo se aplica a las columnas y las restricciones a nivel de tabla se aplican a toda la tabla.

A continuación algunas restricciones o CONSTRAINTS de uso común en SQLite:

- **NOT NULL:** Se asegura de que una columna no puede tener valores NULL.
- **DEFAULT:** Cuando no se especifica un valor de columna, se aplica el valor predeterminado de la columna proporcionada.
- **UNIQUE:** Se asegura de que todos los valores de una columna son diferentes.
- **PRIMARY KEY:** Identifica de forma única cada fila de la tabla de base de datos / registro.
- **CHECK:** Se asegura de que todos los valores de una columna satisfacen ciertas condiciones.

RESTRICCION NOT NULL

Por defecto, una columna puede contener valores nulos. Si no desea que una columna tenga un valor nulo, es necesario definir esta restricción en la columna designada.

Ej:

```
CREATE TABLE empresa(  
    id INTEGER PRIMARY KEY NOT NULL,  
    nombre TEXT NOT NULL  
);
```

RESTRICCIÓN DEFAULT

Proporciona un valor por defecto para cada dato al crear un atributo.

Ej:

```
CREATE TABLE empresa(  
    id INTEGER PRIMARY KEY NOT NULL,  
    nombre TEXT NOT NULL  
    salario REAL DEFAULT 50000.00  
);
```

RESTRICCIÓN UNIQUE

La restricción UNIQUE impide la existencia de dos registros que tengan el mismo valor en una columna en particular. En la tabla empresa, por ejemplo, es posible que desees evitar que dos o más personas tengan la misma edad.

Ej:

```
CREATE TABLE empresa(  
    id INTEGER PRIMARY KEY NOT NULL,  
    nombre TEXT NOT NULL,  
    edad INTEGER NOT NULL  
);
```

RESTRICCION PRIMARY KEY

Una llave primaria es una columna o grupo de columnas que se usa para identificar la unicidad de un registro en una tabla. Cada tabla tiene una y solo una clave primaria.

SQLite te permite definir claves primarias de 2 formas:

Cuando la clave primaria abarca solo una columna puedes usar la restricción de llave primaria para definir la clave primaria.

Ej:

```
CREATE TABLE table_name(  
    column_1 INTEGER NOT NULL PRIMARY KEY,  
    ...  
);
```

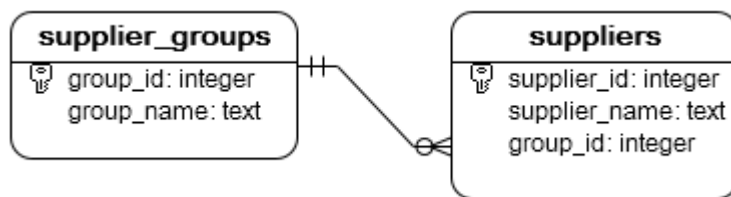
Pero si la clave primaria va a abarcar mas de una columna se debe especificar en una sentencia aparte.

Ej:

```
CREATE TABLE table_name(  
    column_1 INTEGER NOT NULL,  
    column_2 INTEGER NOT NULL,  
    ...  
    PRIMARY KEY(column_1,column_2,...)  
);
```

CLAVES FORÁNEAS

Supongamos la existencia de estas 2 tablas:



Por el tipo de relación, la entidad **suppliers** absorbe la clave primaria de **supplier_groups** y la incorpora como clave foránea. La sentencia para esto sería la siguiente:

```
CREATE TABLE suppliers (  
    supplier_id INTEGER PRIMARY KEY,  
    supplier_name TEXT NOT NULL,  
    group_id INTEGER NOT NULL,  
    FOREIGN KEY (group_id)  
        REFERENCES supplier_groups (group_id)  
);
```

RESTRICCIÓN CHECK

La restricción "check" especifica los valores que acepta un campo, evitando que se ingresen valores inapropiados cuando ejecutamos un comando insert o update.

Definimos la restricción "check" cuando creamos la tabla:

```
CREATE TABLE libros(  
    codigo INTEGER PRIMARY KEY,  
    titulo TEXT,  
    autor TEXT,  
    editorial TEXT,
```



```
    precio REAL CHECK(precio >= 0)
);
```

Definimos la restricción para el campo precio indicando que el valor a almacenar no puede ser negativo:

```
    precio REAL CHECK(precio >= 0)
```

Luego si efectuamos un insert y tratamos de insertar un valor negativo para el campo precio:

```
INSERT INTO libros (titulo, autor, editorial, precio)
VALUES('El aleph', 'Borges', 'Emece', -100);
```

CHECK constraint failed: libros:

```
INSERT INTO libros (titulo, autor, editorial, precio)
VALUES('El aleph', 'Borges', 'Emece', -100);
```

La inserción no se efectúa ya que no se cumple la restricción para el campo precio.

CARGA DE REGISTROS A UNA TABLA

Un registro es una fila de la tabla que contiene los datos propiamente dichos. Cada registro tiene un dato por cada columna.

Recordemos cómo creamos la tabla "usuarios":

```
CREATE TABLE usuarios (
    nombre TEXT,
    clave TEXT
);
```

Al ingresar los datos de cada registro debe tenerse en cuenta la cantidad y el orden de los campos.

Ahora vamos a agregar un registro a la tabla:

```
INSERT INTO usuarios (nombre, clave) VALUES ('Mario Perez', 'Marito');
```

No es necesario especificar en la consulta los nombres de las columnas, pero hay que asegurarse de que los valores que insertamos en la tabla están en el mismo orden, si ingresamos los datos en otro orden, los datos se guardan de modo incorrecto.

Por lo general se especifican las columnas cuando agregamos un registro y queremos omitir un atributo. Pero es perfectamente posible hacer lo siguiente:

```
INSERT INTO usuarios VALUES ('Mario Perez', 'Marito')
```

Note que los datos ingresados, como corresponden a campos de cadenas de caracteres se colocan entre comillas simples. Las comillas simples son OBLIGATORIAS.

CONSULTAS

RECUPERAR CAMPOS

Ahora que sabemos cómo crear tablas y agregarles contenido, viene una de las parte mas importantes, como consultamos estas tablas?

Empecemos por lo básico, como vemos una tabla por completo? Con la siguiente sentencia:

```
SELECT * FROM nombre_tabla;
```

El * indica que queremos consultar todas los atributos o columnas de la tabla.

Pero si queremos consultar solo ciertos atributos usamos las siguientes sintaxis:

```
SELECT columna_1, columna_2, ..., columna_3 FROM nombre_tabla;
```

Y nos aparecerá una lista de registros en el orden que fueron agregados.

NOTA: Para tener una vista que nos brinde mas datos usar los siguientes comandos:

.header on

.mode column

Ejemplo:

Tenemos la siguiente tabla en nuestra db, con nombre empleados:

nombre	documento	sexo	domicilio	sueldobasico
Juan Perez	22333444	m	Sarmiento 123	500.25
Ana Acosta	24555666	f	Colon 134	650
Bartolome Barrios	27888999	m	Urquiza 479	800

Para visualizarla por completo usamos el comando:

```
SELECT * FROM empleados;
```

Y para recuperar sólo algunos atributos por ejemplo nombre y documento, usamos la siguiente sentencia:

```
SELECT nombre, documento FROM empleados;
```

Y obtendremos la siguiente tabla:

nombre	documento
Juan Perez	22333444
Ana Acosta	24555666
Bartolome Barrios	27888999

RECUPERAR ALGUNOS REGISTROS

Hemos aprendido a seleccionar algunos campos de una tabla pero también es posible recuperar algunos registros.

Existe una cláusula, "**where**" con la cual podemos especificar condiciones para una consulta "select". Es decir, podemos recuperar algunos registros, sólo los que cumplan con ciertas condiciones indicadas con la cláusula "where". Por ejemplo, queremos ver el usuario cuyo nombre es "Marcelo", para ello utilizamos "where" y luego de ella, la condición:

```
SELECT nombre, clave
FROM usuarios
WHERE nombre='Marcelo';
```

La sintaxis básica y general es la siguiente:

```
SELECT nombre_campo1, ..., nombre_campoN clave
FROM nombre_tabla
WHERE condicion;
```

Para las condiciones se utilizan operadores relacionales (tema que trataremos más adelante en detalle). El signo igual (=) es un operador relacional. También podemos concatenar condiciones con operadores lógicos (AND, OR, etc), los cuales también veremos mas adelante.

Para la siguiente selección de registros especificamos una condición que solicita los usuarios cuya clave es igual a "River":

```
SELECT nombre, clave
FROM usuarios
WHERE clave='River';
```

Si ningún registro cumple la condición establecida con el "where", no aparecerá ningún registro.

Entonces, con "where" establecemos condiciones para recuperar algunos registros.

Para recuperar algunos campos de algunos registros combinamos en la consulta la lista de campos y la cláusula "where":

```
SELECT nombre
  FROM usuarios
 WHERE clave='River';
```

En la consulta anterior solicitamos el nombre de todos los usuarios cuya clave sea igual a "River", pero no visualizamos el atributo clave en la consulta.

OPERADORES RELACIONALES

Los operadores relacionales (o de comparación) nos permiten comparar dos expresiones, que pueden ser variables, valores de campos, etc.

Hemos aprendido a especificar condiciones de igualdad para seleccionar registros de una tabla; por ejemplo:

```
SELECT * FROM libros
 WHERE autor='Borges';
```

Utilizamos el operador relacional de igualdad.

Los operadores relacionales vinculan un campo con un valor para que SQLite compare cada registro (el campo especificado) con el valor dado.

Los operadores relacionales son los siguientes:

- = igual
- <> distinto
- > mayor
- < menor
- >= mayor o igual
- <= menor o igual

Podemos seleccionar los registros cuyo autor sea diferente de 'Borges', para ello usamos la condición:

```
SELECT * FROM libros
 WHERE autor<>'Borges';
```

Podemos comparar valores numéricos. Por ejemplo, queremos mostrar los títulos y precios de los libros cuyo precio sea mayor a 20 pesos:

```
SELECT titulo, precio
  FROM libros
 WHERE precio>20;
```

Los operadores relacionales comparan valores del mismo tipo. Se emplean para comprobar si un campo cumple con una condición.

OPERADORES LÓGICOS

Hasta el momento, hemos aprendido a establecer una condición con "where" utilizando operadores relacionales. Podemos establecer más de una condición con la cláusula "where", para ello aprenderemos los operadores lógicos.

Son los siguientes:

- and, significa "y",
- or, significa "y/o",
- not, significa "no", invierte el resultado
- (), paréntesis

Los operadores lógicos se usan para combinar condiciones.

Queremos recuperar todos los registros cuyo autor sea igual a "Borges" y cuyo precio no supere los 20 pesos, para ello necesitamos 2 condiciones:

```
SELECT * FROM libros
WHERE autor<>'Borges' and
      precio<=20;
```

Los registros recuperados en una sentencia que une 2 condiciones con el operador "and", cumplen con las 2 condiciones.

Queremos ver los libros cuyo autor sea "Borges" y/o cuya editorial sea "Planeta":

```
SELECT * FROM libros
WHERE autor<>'Borges' or
      editorial='planeta';
```

En la sentencia anterior usamos el operador "or", indicamos que recupere los libros en los cuales el valor del campo "autor" sea "Borges" y/o el valor del campo "editorial" sea "Planeta", es decir, seleccionará los registros que cumplan con la primera condición, con la segunda condición o con ambas condiciones.

Los registros recuperados con una sentencia que une 2 condiciones con el operador "or", cumplen 1 de las condiciones o ambas.

Queremos recuperar los libros que no cumplan la condición dada, por ejemplo, aquellos cuya editorial NO sea "Planeta":

```
SELECT * FROM libros
WHERE not editorial='planeta';
```

El operador "not" invierte el resultado de la condición a la cual antecede.

Los registros recuperados en una sentencia en la cual aparece el operador "not", no cumplen con la condición a la cual afecta el "NO".

Los paréntesis se usan para encerrar condiciones, para que se evalúen como una sola expresión.

Cuando explicitamos varias condiciones con diferentes operadores lógicos (combinamos "and", "or") permite establecer el orden de prioridad de la evaluación; además permite diferenciar las expresiones más claramente.

Por ejemplo, las siguientes expresiones devuelven un resultado diferente:

```
SELECT * FROM libros
WHERE (autor='Borges') or
      (editorial='Paidós' and precio<20);
```

```
SELECT * FROM libros
WHERE (autor='Borges' or editorial='Paidós') and
      (precio<20);
```

Si bien los paréntesis no son obligatorios en todos los casos, se recomienda utilizarlos para evitar confusiones.

El orden de prioridad de los operadores lógicos es el siguiente: "not" se aplica antes que "and" y "and" antes que "or", si no se especifica un orden de evaluación mediante el uso de paréntesis.

CONTAR REGISTROS

Existen en SQLite funciones que nos permiten contar registros, calcular sumas, promedios, obtener valores máximos y mínimos. Estas funciones se denominan funciones de agregado y operan sobre un conjunto de valores (registros), no con datos individuales y devuelven un único valor.

Imaginemos que nuestra tabla "libros" contiene muchos registros. Para averiguar la cantidad sin necesidad de contarlos manualmente usamos la función "count()":

```
SELECT count(*)
FROM libros;
```

La función "count()" cuenta la cantidad de registros de una tabla, incluyendo los que tienen valor nulo.

También podemos utilizar esta función junto con la cláusula "where" para una consulta más específica. Queremos saber la cantidad de libros de la editorial "Planeta":

```
SELECT count(*)
```

```
FROM libros
WHERE editorial='Planeta';
```

Para contar los registros que tienen precio (sin tener en cuenta los que tienen valor nulo), usamos la función "count()" y en los paréntesis colocamos el nombre del campo que necesitamos contar:

```
SELECT count(precio)
FROM libros;
```

Note que "count(*)" retorna la cantidad de registros de una tabla (incluyendo los que tienen valor "null") mientras que "count(precio)" retorna la cantidad de registros en los cuales el campo "precio" no es nulo. No es lo mismo. "count(*)" cuenta registros, si en lugar de un asterisco colocamos como argumento el nombre de un campo, se contabilizan los registros cuyo valor en ese campo NO es nulo.

OTRAS FUNCIONES DE AGRUPAMIENTO

Hemos visto que SQLite tiene una función que nos permite contar registros.

Existen además de la función count las funciones sum, min, max y avg. Todas estas funciones retornan "null" si ningún registro cumple con la condición del "where", excepto "count" que en tal caso retorna cero.

El tipo de dato del campo determina las funciones que se pueden emplear con ellas.

Las relaciones entre las funciones de agrupamiento y los tipos de datos es la siguiente:

- count: se puede emplear con cualquier tipo de dato.
- min y max: con cualquier tipo de dato.
- sum y avg: sólo en campos de tipo numérico.

La función "sum()" retorna la suma de los valores que contiene el campo especificado. Si queremos saber la cantidad total de libros que tenemos disponibles para la venta, debemos sumar todos los valores del campo "cantidad":

```
SELECT sum(cantidad)
FROM libros;
```

Para averiguar el valor máximo o mínimo de un campo usamos las funciones "max()" y "min()" respectivamente.

Queremos saber cuál es el mayor precio de todos los libros:

```
SELECT max(precio)
FROM libros;
```

Entonces, dentro del paréntesis de la función colocamos el nombre del campo del cuál queremos el máximo valor.

La función "avg()" retorna el valor promedio de los valores del campo especificado. Queremos saber el promedio del precio de los libros referentes a "PHP":

```
SELECT avg(precio)
FROM libros
WHERE titulo LIKE '%PHP%';
```

Ahora podemos entender porque estas funciones se denominan "funciones de agrupamiento", porque operan sobre conjuntos de registros, no con datos individuales.

Tratamiento de los valores nulos:

Si realiza una consulta con la función "count" de un campo que contiene 18 registros, 2 de los cuales contienen valor nulo, el resultado devuelve un total de 16 filas porque no considera aquellos con valor nulo.

Todas las funciones de agregado, excepto "count(*)", excluye los valores nulos de los campos. "count(*)" cuenta todos los registros, incluidos los que contienen "null".

ORDENANDO RESULTADOS

Muchas veces queremos que los datos consultados esten en un determinado orden en base a alguno de sus atributos, la cláusula ORDER BY nos ayuda con esto.

```
SELECT columna_1, columna_2,..., columna_n
FROM nombre_tabla
[WHERE condición]
[ORDER BY columna_1, columna_2, ..., column_n] [ASC | DESC];
```

El orden en que coloquemos los atributos a evaluar a continuación de ORDER BY representarán la jerarquía de ordenamiento, es decir primero comparara por el valor de la columna_1, si es igual pasa a comparar columna_2 y así sucesivamente.

También podemos especificar si el ordenamiento será de orden ascendente o descendente.

Ej:

```
id nombre edad direccion salario
-----
1 Pablo 32 California 20000.0
2 David 25 Tejas 15000.0
3 Teddy 23 Noruega 20000.0
4 Marcos 25 Rich Mond 65000.0
5 David 27 Texas 85000.0
6 Kim 22 Sur-Hall 45.000.0
7 James 24 Houston 10000.0
```



```
sqlite> SELECT * FROM empleados ORDER BY nombre, salario ASC;
```

```
id nombre edad direccion salario
-----
2 David 25 Texas 15000.0
5 David 27 Texas 85000.0
7 James 24 Houston 10000.0
6 Kim 22 Sur-Hall 45.000.0
4 Marcos 25 Rich Mond 65000.0
1 Pablo 32 California 20000.0
3 Teddy 23 Noruega 20000.0
```

AGRUPAR REGISTROS

Hemos aprendido que las funciones de agregado permiten realizar varios cálculos operando con conjuntos de registros.

Las funciones de agregado solas producen un valor de resumen para todos los registros de un campo. Podemos generar valores de resumen para un solo campo, combinando las funciones de agregado con la cláusula "group by", que agrupa registros para consultas detalladas.

Queremos saber la cantidad de libros de cada editorial, podemos tipear la siguiente sentencia:

```
SELECT count(*) FROM libros
WHERE editorial='Planeta';
```

y repetirla con cada valor de "editorial":

```
SELECT count(*) FROM libros
WHERE editorial='Emece';
SELECT count(*) FROM libros
WHERE editorial='Paidos';
...
```

Pero hay otra manera, utilizando la cláusula "group by":

```
SELECT editorial, count(*)
FROM libros
GROUP BY editorial;
```

La instrucción anterior solicita que muestre el nombre de la editorial y cuente la cantidad agrupando los registros por el campo "editorial". Como resultado aparecen los nombres de las editoriales y la cantidad de registros para cada valor del campo.

Los valores nulos se procesan como otro grupo.

Entonces, para saber la cantidad de libros que tenemos de cada editorial, utilizamos la función "count()", agregamos "group by" (que agrupa registros) y el campo por el que deseamos que se realice el agrupamiento, también colocamos el nombre del campo a recuperar; la sintaxis básica es la siguiente:

```
SELECT campo, funcion_de_agregado
FROM nombre_tabla
GROUP BY campo;
```

También se puede agrupar por más de un campo, en tal caso, luego del "group by" se listan los campos, separados por comas. Todos los campos que se especifican en la cláusula "group by" deben estar en la lista de selección.

```
SELECT campo_1, campo_2, funcion_de_agregado
FROM nombre_tabla
GROUP BY campo_1, campo_2;
```

Para obtener la cantidad libros con precio no nulo, de cada editorial utilizamos la función "count()" enviándole como argumento el campo "precio", agregamos "group by" y el campo por el que deseamos que se realice el agrupamiento (editorial):

```
SELECT editorial, count(precio)
FROM libros
GROUP BY editorial;
```

Como resultado aparecen los nombres de las editoriales y la cantidad de registros de cada una, sin contar los que tienen precio nulo.

Recuerde la diferencia de los valores que retorna la función "count()" cuando enviamos como argumento un asterisco o el nombre de un campo: en el primer caso cuenta todos los registros incluyendo los que tienen valor nulo, en el segundo, los registros en los cuales el campo especificado es no nulo.

Para conocer el total en dinero de los libros agrupados por editorial:

```
SELECT editorial, sum(precio)
FROM libros
GROUP BY editorial;
```

Para saber el máximo y mínimo valor de los libros agrupados por editorial:

```
SELECT editorial,
MAX(precio) as mayor,
MIN(precio) as menor
FROM libros
GROUP BY editorial;
```

Para calcular el promedio del valor de los libros agrupados por editorial:

```
SELECT editorial, avg(precio)
FROM libros
GROUP BY editorial;
```

Es posible limitar la consulta con "where".

Si incluye una cláusula "where", sólo se agrupan los registros que cumplen las condiciones.

Vamos a contar y agrupar por editorial considerando solamente los libros cuyo precio sea menor a 30 pesos:

```
SELECT editorial, count(*)
FROM libros
WHERE precio<30
GROUP BY editorial;
```

SELECCIONAR GRUPOS

Así como la cláusula "where" permite seleccionar (o rechazar) registros individuales; la cláusula "having" permite seleccionar (o rechazar) un grupo de registros.

Si queremos saber la cantidad de libros agrupados por editorial usamos la siguiente instrucción ya aprendida:

```
SELECT editorial, count(*)
FROM libros
GROUP by editorial;
```

Si queremos saber la cantidad de libros agrupados por editorial pero considerando sólo algunos grupos, por ejemplo, los que devuelvan un valor mayor a 2, usamos la siguiente instrucción:

```
SELECT editorial, count(*) FROM libros
GROUP BY editorial
HAVING count(*)>2;
```

Se utiliza "having", seguido de la condición de búsqueda, para seleccionar ciertas filas retornadas por la cláusula "group by".

Veamos otros ejemplos. Queremos el promedio de los precios de los libros agrupados por editorial, pero solamente de aquellos grupos cuyo promedio supere los 25 pesos:

```
SELECT editorial, avg(precio) FROM libros
GROUP BY editorial
HAVING avg(precio)>25;
```

En algunos casos es posible confundir las cláusulas "where" y "having". Queremos contar los registros agrupados por editorial sin tener en cuenta a la editorial "Planeta".

Analicemos las siguientes sentencias:

```
SELECT editorial, count(*) FROM libros
WHERE editorial<>'Planeta'
GROUP BY editorial;
```

```
SELECT editorial, count(*) FROM libros
GROUP BY editorial
HAVING editorial<>'Planeta';
```

Ambas devuelven el mismo resultado, pero son diferentes. La primera, selecciona todos los registros rechazando los de editorial "Planeta" y luego los agrupa para contarlos. La segunda, selecciona todos los registros, los agrupa para contarlos y finalmente rechaza fila con la cuenta correspondiente a la editorial "Planeta".

No debemos confundir la cláusula "where" con la cláusula "having"; la primera establece condiciones para la selección de registros de un "select"; la segunda establece condiciones para la selección de registros de una salida "group by".

Veamos otros ejemplos combinando "where" y "having". Queremos la cantidad de libros, sin considerar los que tienen precio nulo, agrupados por editorial, sin considerar la editorial "Planeta":

```
SELECT editorial, count(*) FROM libros
WHERE precio is not null
GROUP BY editorial
HAVING editorial<>'Planeta';
```

Aquí, selecciona los registros rechazando los que no cumplan con la condición dada en "where", luego los agrupa por "editorial" y finalmente rechaza los grupos que no cumplan con la condición dada en el "having".

Se emplea la cláusula "having" con funciones de agrupamiento, esto no puede hacerlo la cláusula "where". Por ejemplo queremos el promedio de los precios agrupados por editorial, de aquellas editoriales que tienen más de 2 libros:

```
SELECT editorial, avg(precio) FROM libros
GROUP BY editorial
HAVING count(*) > 2;
```

Podemos encontrar el mayor valor de los libros agrupados y ordenados por editorial y seleccionar las filas que tengan un valor menor a 100 y mayor a 30:

```
SELECT editorial, max(precio) as mayor
FROM libros
GROUP BY editorial
HAVING min(precio)<100 AND
min(precio)>30
ORDER BY editorial;
```

Entonces, usamos la cláusula "having" para restringir las filas que devuelve una salida "group by". Va siempre después de la cláusula "group by" y antes de la cláusula "order by" si la hubiere.

REGISTROS DUPLICADOS

Con la cláusula "distinct" se especifica que los registros con ciertos datos duplicados sean obviados en el resultado. Por ejemplo, queremos conocer todos los autores de los cuales tenemos libros, si utilizamos esta sentencia:

```
SELECT autor FROM libros;
```

Aparecen repetidos. Para obtener la lista de autores sin repetición usamos:

```
SELECT distinct autor FROM libros;
```

También podemos tipear:

```
SELECT autor FROM libros  
GROUP BY autor;
```

Note que en los tres casos anteriores aparece "null" como un valor para "autor". Si sólo queremos la lista de autores conocidos, es decir, no queremos incluir "null" en la lista, podemos utilizar la sentencia siguiente:

```
SELECT DISTINCT autor FROM libros  
WHERE autor IS NOT NULL;
```

Para contar los distintos autores, sin considerar el valor "null" usamos:

```
SELECT count(DISTINCT autor)  
FROM libros;
```

Note que si contamos los autores sin "distinct", no incluirá los valores "null" pero si los repetidos:

```
SELECT count(autor)  
FROM libros;
```

Esta sentencia cuenta los registros que tienen autor.

Podemos combinarla con "where". Por ejemplo, queremos conocer los distintos autores de la editorial "Planeta":

```
SELECT DISTINCT autor FROM libros  
WHERE editorial='Planeta';
```

También puede utilizarse con "group by" para contar los diferentes autores por editorial:

```
SELECT editorial, count(DISTINCT autor)
FROM libros
GROUP BY editorial;
```

La cláusula "distinct" afecta a todos los campos presentados. Para mostrar los títulos y editoriales de los libros sin repetir títulos ni editoriales, usamos:

```
SELECT DISTINCT titulo,editorial
FROM libros
ORDER BY titulo;
```

Note que los registros no están duplicados, aparecen títulos iguales pero con editorial diferente, cada registro es diferente.

Entonces, "distinct" elimina registros duplicados.

ACTUALIZAR REGISTROS

Decimos que actualizamos un registro cuando modificamos alguno de sus valores.

Para modificar uno o varios datos de uno o varios registros utilizamos el comando **"update"** (actualizar).

Por ejemplo, en nuestra tabla "usuarios", queremos cambiar los valores de todas las claves por la cadena "RealMadrid":

```
UPDATE usuarios SET clave='RealMadrid';
```

Utilizamos "update" junto al nombre de la tabla y "set" junto con el campo a modificar y su nuevo valor.

El cambio afectará a todos los registros.

Podemos modificar sólo algunos registros, para ello debemos establecer condiciones de selección con la cláusula **"where"**.

Por ejemplo, queremos cambiar el valor correspondiente a la clave de nuestro usuario llamado "Federico Lopez", queremos como nueva clave "Boca", necesitamos una condición "where" que afecte solamente a este registro:

```
UPDATE usuarios SET clave='Boca'
WHERE nombre='Federico Lopez';
```

Si SQLite no encuentra registros que cumplan con la condición del "where", no se modifica ninguno.

Las condiciones no son obligatorias, pero si omitimos la cláusula "where", la actualización afectará a todos los registros.

También podemos actualizar varios campos en una sola instrucción:

```
UPDATE usuarios SET nombre='Marcelo Duarte', clave='Marce'
WHERE nombre='Marcelo';
```

Para ello colocamos "update", el nombre de la tabla, "set" junto al nombre del campo y el nuevo valor y separado por coma, el otro nombre del campo con su nuevo valor.

BORRAR REGISTROS

Para eliminar los registros de una tabla usamos el comando "delete":

```
DELETE FROM usuarios;
```

Si no queremos eliminar todos los registros, sino solamente algunos, debemos indicar cuál o cuáles, para ello utilizamos el comando "delete" junto con la cláusula "where" con la cual establecemos la condición que deben cumplir los registros a borrar.

```
DELETE FROM usuarios WHERE nombre='Marcelo';
```

Si solicitamos el borrado de un registro que no existe, es decir, ningún registro cumple con la condición especificada, ningún registro será eliminado.

Tenga en cuenta que si no colocamos una condición, se eliminan todos los registros de la tabla nombrada.

BÚSQUEDA DE PATRONES

Existe un operador que se usa para realizar comparaciones exclusivamente de cadenas, "like" y "not like".

Hemos realizado consultas utilizando operadores relacionales para comparar cadenas. Por ejemplo, sabemos recuperar los libros cuyo autor sea igual a la cadena "Borges":

```
SELECT * FROM libros
WHERE autor='Borges';
```

El operador igual ("=") nos permite comparar cadenas de caracteres, pero al realizar la comparación, busca coincidencias de cadenas completas, realiza una búsqueda exacta.

Imaginemos que tenemos registrados estos 2 libros:

```
"El Aleph", "Borges";
"Antología poética", "J.L. Borges";
```

Si queremos recuperar todos los libros de "Borges" y especificamos la siguiente condición previa, sólo aparecerá el primer registro, ya que la cadena "Borges" no es igual a la cadena "J.L. Borges".

Esto sucede porque el operador "=" (igual), también el operador "<>" (distinto) comparan cadenas de caracteres completas. Para comparar porciones de cadenas utilizamos los operadores "like" y "not like".

Entonces, podemos comparar trozos de cadenas de caracteres para realizar consultas. Para recuperar todos los registros cuyo autor contenga la cadena "Borges" debemos codificar en SQLite:

```
SELECT * FROM libros
WHERE autor LIKE "%Borges%";
```

El símbolo "%" (porcentaje) reemplaza cualquier cantidad de caracteres (incluyendo ningún carácter). Es un carácter comodín. "like" y "not like" son operadores de comparación que señalan igualdad o diferencia.

Para seleccionar todos los libros que comiencen con "M":

```
SELECT * FROM libros
WHERE titulo NOT LIKE 'M%';
```

Así como "%" reemplaza cualquier cantidad de caracteres, el guión bajo "_" reemplaza un carácter, es otro carácter comodín. Por ejemplo, queremos ver los libros de "Lewis Carroll" pero no recordamos si se escribe "Carroll" o "Carrolt", entonces tipeamos esta condición:

```
SELECT * FROM libros
WHERE autor LIKE "%Carrol_";
```

"like" se emplea solo con tipos de datos TEXT.

ÍNDICE DE UNA TABLA

Para facilitar la obtención de información de una tabla se utilizan índices.

El índice de una tabla desempeña la misma función que el índice de un libro: permite encontrar datos rápidamente; en el caso de las tablas, localiza registros.

Una tabla se indexa por un campo (o varios).

El índice es un tipo de archivo con 2 entradas: un dato (un valor de algún campo de la tabla) y un puntero.

Un índice posibilita el acceso directo y rápido haciendo más eficiente las búsquedas. Sin índice, se debe recorrer secuencialmente toda la tabla para encontrar un registro.

El objetivo de un índice es acelerar la recuperación de información, es decir tiene sentido en tablas con muchos registros.

La desventaja es que consume espacio en el disco y las inserciones y borrados de registros son más lentas.

La indexación es una técnica que optimiza el acceso a los datos, mejora el rendimiento acelerando las consultas y otras operaciones. Es útil cuando la tabla contiene miles de registros.

Los índices se usan para varias operaciones:

- para buscar registros rápidamente.
- para recuperar registros de otras tablas empleando "join".

Es importante identificar el o los campos por los que sería útil crear un índice, aquellos campos por los cuales se realizan operaciones de búsqueda con frecuencia.

Hay distintos tipos de índices, a saber:

1) "primary key": es el que definimos como clave primaria. Los valores indexados deben ser únicos. Una tabla solamente puede tener una clave primaria.

2) "index": crea un índice común, los valores no necesariamente son únicos y aceptan valores "null". "key" es sinónimo de "index". Puede haber varios por tabla.

3) "unique": crea un índice para los cuales los valores deben ser únicos y diferentes, aparece un mensaje de error si intentamos agregar un registro con un valor ya existente. Permite valores nulos y pueden definirse varios por tabla.

Todos los índices pueden ser multicolumnas, es decir, pueden estar formados por más de 1 campo.

En las siguientes lecciones aprenderemos sobre cada uno de ellos.

Los nombres de índices aceptan todos los caracteres.

Una tabla puede ser indexada por campos de tipo numérico o de tipo carácter. También se puede indexar por un campo que contenga valores NULL, excepto los PRIMARY.

ÍNDICE COMÚN

Dijimos que hay 3 tipos de índices. Hasta ahora solamente conocemos la clave primaria que definimos al momento de crear una tabla.

El índice llamado primary se crea automáticamente cuando establecemos un campo como clave primaria.

Vamos a otro tipo de índice común. Un índice común se crea con "create index", los valores no necesariamente son únicos y aceptan valores "null". Puede haber varios por tabla. Un índice puede estar formado por una o más columnas.

Ej:

Creemos la tabla con la siguiente estructura:

```
CREATE TABLE libros(  
    codigo INTEGER PRIMARY KEY,  
    titulo TEXT,  
    autor TEXT,  
    editorial TEXT,  
    precio REAL  
);  
  
CREATE INDEX I_libros_editorial ON libros(editorial);
```

Empleamos el comando create index para la creación, indicando seguidamente el nombre del índice, luego la palabra clave on, el nombre de la tabla y entre paréntesis el o los campos por el que se crea el índice.

ÍNDICE ÚNICO

Veamos el otro tipo de índice, único. Un índice único se crea con "create unique index", los valores deben ser únicos y diferentes, aparece un mensaje de error si intentamos agregar un registro con un valor ya existente. Permite valores nulos y pueden definirse varios por tabla. Debemos darle un nombre.

Ej: Vamos a trabajar con nuestra tabla "libros".

```
CREATE TABLE libros(  
    codigo INTEGER PRIMARY KEY,  
    titulo TEXT,  
    autor TEXT,  
    editorial TEXT,  
    precio REAL  
);
```

Creemos un índice único sobre los campos título y editorial:

```
CREATE UNIQUE INDEX I_libros_tituloeditorial ON libros(titulo,editorial);
```

Empleamos el comando create unique index para la creación, indicando seguidamente el nombre del índice, luego la palabra clave on, el nombre de la tabla y entre paréntesis el o los campos por el que se crea el índice.

Con este tipo de índice no podemos insertar dos filas que contengan en los campos título y editorial el mismo valor.

TRABAJAR CON VARIAS TABLAS

Hasta el momento hemos trabajado con una sola tabla, pero generalmente, se trabaja con más de una.

Para evitar la repetición de datos y ocupar menos espacio, se separa la información en varias tablas. Cada tabla almacena parte de la información que necesitamos registrar.

Por ejemplo, los datos de nuestra tabla "libros" podrían separarse en 2 tablas, una llamada "libros" y otra "editoriales" que guardará la información de las editoriales.

En nuestra tabla "libros" haremos referencia a la editorial colocando un código que la identifique.

Veamos:

```
CREATE TABLE libros(  
    codigo INTEGER PRIMARY KEY,  
    titulo TEXT,  
    autor TEXT,  
    precio REAL,  
    codigoe_ditorial INTEGER  
);
```

```
CREATE TABLE editoriales(  
    codigo INTEGER PRIMARY KEY,  
    nombre TEXT  
);
```

De esta manera, evitamos almacenar tantas veces los nombres de las editoriales en la tabla "libros" y guardamos el nombre en la tabla "editoriales"; para indicar la editorial de cada libro agregamos un campo que hace referencia al código de la editorial en la tabla "libros" y en "editoriales".

Al recuperar los datos de los libros con la siguiente instrucción:

```
SELECT * FROM libros;
```

vemos que en el campo "codigoe_ditorial" aparece el código, pero no sabemos el nombre de la editorial.

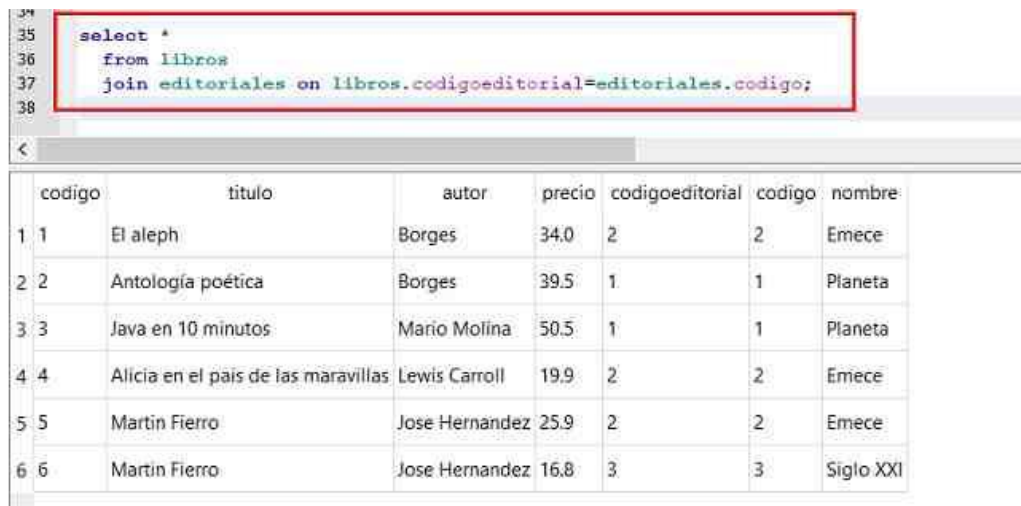
Para obtener los datos de cada libro, incluyendo el nombre de la editorial, necesitamos consultar ambas tablas, traer información de las dos.

Cuando obtenemos información de más de una tabla decimos que hacemos un "join" (combinación).

Veamos un ejemplo:

```
CREATE TABLE libros(  
    codigo INTEGER PRIMARY KEY,  
    titulo TEXT,  
    autor TEXT,  
    precio REAL,  
    codigoe_ditorial INTEGER  
);  
  
CREATE TABLE editoriales(  
    codigo INTEGER PRIMARY KEY,  
    nombre TEXT  
);  
  
INSERT INTO editoriales(nombre) VALUES('Planeta');  
INSERT INTO editoriales(nombre) VALUES('Emece');  
INSERT INTO editoriales(nombre) VALUES('Siglo XXI');  
  
INSERT INTO libros (titulo, autor, codigo_editorial, precio)  
VALUES('El aleph', 'Borges', 2, 34);  
INSERT INTO libros (titulo, autor, codigo_editorial, precio)  
VALUES('Antología poética', 'Borges', 1, 39.50);  
INSERT INTO libros (titulo, autor, codigo_editorial, precio)  
VALUES('Java en 10 minutos', 'Mario Molina', 1, 50.50);  
INSERT INTO libros (titulo, autor, codigo_editorial, precio)  
VALUES('Alicia en el país de las maravillas', 'Lewis Carroll', 2, 19.90);  
INSERT INTO libros (titulo, autor, codigo_editorial, precio)  
VALUES('Martin Fierro', 'Jose Hernandez', 2, 25.90);  
INSERT INTO libros (titulo, autor, codigo_editorial, precio)  
VALUES('Martin Fierro', 'Jose Hernandez', 3, 16.80);  
  
SELECT *  
  
FROM libros  
  
JOIN editoriales ON libros.codigoeditorial=editoriales.codigo;
```

El resultado de esta consulta es:



The screenshot shows a database query interface. At the top, a text box contains the SQL query: `select * from libros join editoriales on libros.codigoeditorial=editoriales.codigo;`. Below the text box is a table with 7 columns: `codigo`, `titulo`, `autor`, `precio`, `codigoeditorial`, `codigo`, and `nombre`. The table contains 6 rows of data, representing the join of the `libros` and `editoriales` tables.

	codigo	titulo	autor	precio	codigoeditorial	codigo	nombre
1	1	El aleph	Borges	34.0	2	2	Emece
2	2	Antología poética	Borges	39.5	1	1	Planeta
3	3	Java en 10 minutos	Mario Molina	50.5	1	1	Planeta
4	4	Alicia en el país de las maravillas	Lewis Carroll	19.9	2	2	Emece
5	5	Martin Fierro	Jose Hernandez	25.9	2	2	Emece
6	6	Martin Fierro	Jose Hernandez	16.8	3	3	Siglo XXI

Resumiendo: si distribuimos la información en varias tablas evitamos la redundancia de datos y ocupamos menos espacio físico en el disco. Un join es una operación que relaciona dos o más tablas para obtener un resultado que incluya datos (campos y registros) de ambas; las tablas participantes se combinan según los campos comunes a ambas tablas.

Podemos hacer un "join" con más de dos tablas.

Cada join combina 2 tablas. Se pueden emplear varios join para enlazar varias tablas. Cada resultado de un join es una tabla que puede combinarse con otro join.