Apuntes de SQL

migueluisV

Realizadas: Marzo 2023

Índice

Índice de Figuras

Índice de Tablas

Este documento se hizo con **Overleaf** y los ejemplos fueron desarrollados y probados dentro del área de código (Code Playground) de **Sololearn**.

1 Las bases de datos

SQL es la abreviación de **Structured Query Language**, suele utilizarse para el manejo, manipulación y administración de bases de datos. Una **base de datos** es una colección de datos ordenada de tal manera que permite acceder a los datos de manera fácil y eficiente para la administración y actualización.

1.1 Composición

Una base de datos está compuesta de *tablas*, las cuales tienen *columnas* y *filas*, las *celdas* es un dato seleccionado en n posición de columna y tabla. Las tablas suelen tener una columna o *atributo* propio e irrepetible que las diferencia de otras, este suele ser llamado *Id*, la *Tabla 1* es un simple ejemplo de la apariencia de una tabla SQL:

Table 1: Ejemplo de una tabla sencilla en SQL: Tabla "Personas"

id	**nombre**	**apellidos**	**ciudad**	**edad**
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
4	Emily	Adams	Houston	29
5	James	Roberts	Philadelphia	31

Como vemos, cada columna representa un atributo o característica de un objeto, evento o cosa del mundo real, ese es el objetivo de las tablas en las bases de datos, almacenar datos del mundo real.

Se dice que los datos almacenados dentro de una base de datos son *registros*, podemos insertar, actualizar, modificar, ordenar, consultar, filtrar registros y otras más operaciones.

El lenguaje SQL para manipulación de base de datos es una cosa, si instalamos este lenguaje en nuestra máquina no podremos interactuar con él, porque no viene con una interfaz gráfica o sistema, es aquí donde entran los **Sistemas Gestores de Bases de Datos** (abreviado *SGBD*), que son los programas o interfaces donde podemos trabajar con SQL. Uno de los SGBD más populares es MySQL, además, algunas SGBD suelen incorporar extensiones o plugins del lenguaje propios, puede que un código de Oracle no funcione con MySQL, pero casi todos las sentencias de SQL se pueden utilizar en todos los gestores.

2 Seleccionando datos

El comando **SELECT** es utilizada para recoger datos de una tabla, el símbolo **asterisco** (*) es utilizado globalmente para recoger todo el contenido de una tabla, mientras que el comando **FROM** le dice a SQL de qué tabla se tiene que recoger la información, por lo que la sentencia queda de la siguiente manera:

```
SELECT * FROM [tabla];
```

Donde [tabla] es el nombre de la tabla que queremos consultar, para los siguientes ejemplos, estaremos trabajando con la tabla "Personas" presentada en la sección anterior. En caso de querer recoger los datos de una sola columna, sustituimos el asterisco por el nombre de la columna o atributo a consultar:

```
SELECT ciudad
FROM Personas;
```

El resultado aparece en la *Tabla 2*:

Table 2: Consulta de una columna en tabla "Personas"

ciudad
New York
Los Angeles
Chicago
Houston
Philadelphia

Si queremos recoger los datos de más de una columna o atributo, simplemente los vamos añadiendo a la sentencia y separamos nombres de columnas por una *coma* (,), el resultado lo encontramos en la *Tabla 3*:

```
SELECT id, nombre
FROM Personas;
```

Table 3: Consulta de varias columnas en tabla "Personas"

id	**nombre**
1	John
2	David
3	Chloe
4	Emily
5	James

Vimos en los ejemplos anteriores que las sentencias las terminamos con un *punto y coma* (;), hacer esto no es necesario, sin embargo, si queremos ejecutar varias sentencias al mismo tiempo, si debemos agregar este punto y coma a cada sentencia:

```
SELECT id, nombre FROM Personas;
SELECT nombre, apellidos FROM Personas;
```

Nota: podemos hacer saltos de línea o espacios dentro de las sentencias para mayor legibilidad, esto debido a que SQL ignora estos saltos y espacios.

SQL no distingue entre mayúsculas y minúsculas, por lo que el siguiente ejemplo regresará el mismo resultado:

```
select id from Personas;
SELECT ID FROM Personas;
SElEct iD frOM Personas;
```

Se suelen escribir las sentencias en mayúsculas, a excepción de los nombres de tablas y columnas claro.

3 Ordenando datos y cantidad de elementos recogidos

3.1 El comando ORDER BY

El comando **ORDER BY** es utilizado para ordenar los datos recogidos del comando *SELECT*. Por defecto, el comando *ORDER BY* ordena los datos ascendentemente, y este comando va al final de la sentencia *SELECT*, como se ve en la *Tabla 4*:

```
SELECT * FROM Personas
ORDER BY nombre
```

Table 4: Ordenamiento de los elementos de una sentencia

id	nombre	apellidos	ciudad	edad
3	Chloe	Anderson	Chicago	65
2	David	Williams	Los Angeles	42
4	Emily	Adams	Houston	29
5	James	Roberts	Philadelphia	31
1	John	Smith	New York	24

Por otra parte, si buscamos que se ordenen de forma descendente los datos recogidos, utilizamos el comando *DESC*, si queremos que se ordenen de forma ascendente, utilizamos *ASC*:

```
SELECT * FROM Personas
ORDER BY nombre DESC
```

El ordenamiento de datos se puede extender a más de una columna, simplemente separamos la forma en la que queremos ordenar por medio de una coma:

```
SELECT * FROM Personas
ORDER BY ciudad ASC, apellidos DESC
```

3.2 El comando LIMIT

A este comando simplemente le pasamos el número entero de elementos que queremos que recoja una consulta SELECT. El comando *LIMIT* va al final de la sentencia SELECT, podemos ver el resultado en la *Tabla 5*:

```
SELECT id, nombre
FROM Personas
LIMIT 2
```

Table 5: Mostrar un límite de elementos recogidos de SELECT

id	nombre
1	John
2	David

Nota: se pueden combinar los comandos LIMIT, ORDER BY, ASC y DESC, solamente tome en cuenta que LIMIT va después de ORDER BY.

El comando *OFFSET* saltará n registros desde el primero, el resultado lo vemos en la *Tabla 6*:

```
SELECT * FROM Personas
OFFSET 2
```

Table 6: Saltado de los elementos de una sentencia

id	nombre	apellidos	ciudad	edad
3	Chloe	Anderson	Chicago	65
4	Emily	Adams	Houston	29
5	James	Roberts	Philadelphia	31

4 DISTINCT

En ocasiones, una columna o atributo puede contener valores iguales o duplicados, agregaremos tres registros más a nuestra tabla de ejemplo (*Tabla 7*):

Table 7: Tabla "Personas" extendida

id	nombre	apellidos	ciudad	edad
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
4	Emily	Adams	Houston	29
5	James	Roberts	Philadelphia	31
6	Andrew	Thomas	New York	21
7	Daniel	Harris	Los Angeles	67
8	Charlotte	Walker	Chicago	45

Vemos que los nuevos registros tienen los valores "New York", "Los Angeles" y "Chicago" repetidos, el comando *DISTINCT* ayuda a remover los resultados duplicados de atributos en una sentencia, este comando va después del comando *SELECT* y antes del atributo al cual aplicar DISTINCT, como se ve en la *Tabla 8*:

```
SELECT DISTINCT ciudad
FROM Personas
```

Table 8: Eliminado de duplicados en una sentencia

ciudad
New York
Los Angeles
Chicago
Houston
Philadelphia

Esta sentencia se puede combinar con las últimas tres mencionadas.

5 Filtrado de datos

Anteriormente, seleccionábamos **todos** los datos de toda una tabla o columna. El comando **WHERE** permite seleccionar únicamente uno o varios registros que cumplan con una condición. Los operadores pueden ser consultados en la *Tabla 9*:

Table 9: Operadores relacionales en SQL

Operador	Descripción
=	Igual a
>	Mayor a
<	Menor a
>=	Mayor o igual a
<=	Menor o igual a
<>	Distinto a
!=	Distinto a (en algunas versiones de SQL)

Se pueden utilizar cadenas dentro del filtrado, se especifica el valor de la misma con comillas simples ('), en caso de que el registro contenga una comilla simple (por ejemplo: 'let's'), agregue otra comilla ('let''s'). Las siguientes sentencias son ejemplos de filtrado de datos con operadores lógicos, copie y pegue en su SGBD para poder apreciar los resultados:

```
SELECT * FROM Personas WHERE edad = 60;
SELECT * FROM Personas WHERE edad $<>$ 60;
SELECT * FROM Personas WHERE nombre = 'David';
SELECT * FROM Personas WHERE edad < 30
```

Los comandos **BETWEEN** y **AND** sirven para seleccionar un rango de registros resultantes de una sentencia con el comando WHERE, un ejemplo se ve en la *Tabla 10*:

```
SELECT * FROM Personas
WHERE edad BETWEEN 30 AND 60
```

Table 10: Seleccionando un rango de registros con WHERE, BETWEEN y AND

id	nombre	apellidos	ciudad	edad
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
5	James	Roberts	Philadelphia	31
7	Daniel	Harris	Los Angeles	67
8	Charlotte	Walker	Chicago	45

Nota: los valores límite (30 y 60 para el ejemplo anterior) son incluidos en los registros resultantes.

6 Condiciones lógicas

Puede combinar operadores relacionales con operadores lógicos, estos pueden ser consultados en la *Tabla 11*:

Table 11: Operadores lógicos en SQL

Operador	**Descripción**
ALL	Regresa TRUE si todos los valores de una sub-sentencia coinciden con la condición
AND	Regresa TRUE si las dos condiciones separadas por el operador AND son ciertas
ANY	Regresa TRUE si alguno de los valores de una sub-sentencia coincide con la condición
BETWEEN	Regresa TRUE si el operador está dentro de un rango de comparación
EXIST	Regresa TRUE si la sub-sentencia regresa uno o varios registros
IN	Regresa TRUE si el operador es igual a un item dentro de una lista de expresiones
LIKE	Regresa TRUE si el operador coincide con el patrón
NOT	Muestra el contrario o negación de una condición verdadera
OR	Regresa TRUE si alguna de las dos condiciones separadas por el operador OR son ciertas
SOME	Regresa TRUE si alguno de los valores de la sub-sentencia coincide con la condición

Ya se utilizaron los operadores **BETWEEN** y **AND** en el tema anterior para conseguir el rango de valores aceptados en la condición. El operador AND permite juntar dos condiciones, si ambas son ciertas, se regresa **TRUE**. El operador **OR** sigue la misma lógica, solamente que si alguna de ambas condiciones es cierta, se regresa TRUE. A continuación, ejemplos:

```
SELECT * FROM Personas
WHERE ciudad = 'Los Angeles' AND edad = 42;
SELECT * FROM Personas
WHERE ciudad = 'Los Angeles' OR ciudad = 'Chicago'
```

El operador **IN** funciona para evitar utilizar varias veces el operador **OR**, como se ve en el siguiente ejemplo:

```
SELECT * FROM Personas
WHERE ciudad
IN ('Los Angeles', 'Chicago')
```

Esta consulta regresa todos los registros que tengan el atributo "ciudad" con los valores "Los Angeles" y "Chicago". Es obligatorio que se utilicen los paréntesis, las comillas simples y separación por comas para todos los valores a utilizar.

Caso contrario, el operador **NOT** regresará todos los registros con el atributo "ciudad" distinto a "Los Angeles" y "Chicago", es decir, todos los registros contrarios a **TRUE** o los registros contrarios a los escritos.

7 Filtros de texto

Vimos anteriormente que se pueden seleccionar registros en base a la cadena de un atributo, a esta característica se podemos sumar que se pueden recoger registros en base a un patrón. Para lograr este cometido, utilizamos el operador **LIKE**, en conjunto con los caracteres comodines (Wildcard Characters) y las comillas simples para encerrar el patrón, uno de los más populares es el comodín %, el cual representa uno o varios caracteres que, para este caso, son ignorados y solamente se toma en cuenta la cadena o caracteres a buscar como patrón, puede consultar más información sobre los comodines en este enlace. Veamos un ejemplo para que se entienda un poco más como se usan estos comodines:

'%sos' ignora los primeros caracteres menos 'sos'.
'be%' ignora los caracteres siguientes a 'be'.
'% san %' ignora los caracteres antes y después de 'san'.

Entonces, con este comodín podemos ignorar n cantidad de caracteres y tomar solamente los que nos interesan, siendo esto un patrón que utiliza el operador **LIKE** para recoger registros con una secuencia de caracteres específicos en una cadena. Veamos varios ejemplos con la tabla de ejemplo que utilizamos y los resultados aparecen en las siguientes tablas:

```
SELECT * FROM Personas
WHERE apellidos
LIKE '\%s'
```

Table 12: Seleccionando registros con una "s" al final

id	nombre	apellidos	ciudad	edad
2	David	Williams	Los Angeles	42
4	Emily	Adams	Houston	29
5	James	Roberts	Philadelphia	31
6	Andrew	Thomas	New York	21
7	Daniel	Harris	Los Angeles	67

```
SELECT * FROM Personas
WHERE apellidos
LIKE '\%a\%'
```

Table 13: Seleccionando registros con una "a" en medio

id	nombre	apellidos	ciudad	edad
2	David	Williams	Los Angeles	42
4	Emily	Adams	Houston	29
6	Andrew	Thomas	New York	21
7	Daniel	Harris	Los Angeles	67
8	Charlotte	Walker	Chicago	45

El primer ejemplo selecciona todos los registros donde el apellido de una persona tenga una "s" al final, mientras que el segundo ejemplo selecciona todos los registros donde el apellido de la persona tenga una "a" en medio.

Este comodín es muy poderoso, podemos usar el comodín _ para seleccionar únicamente un carácter dentro del patrón, como se ve en la *Tabla 14*:

```
SELECT * FROM Personas
WHERE apellidos
LIKE '\_ork'
```

Table 14: Seleccionando registros con "ork" al final

id	nombre	apellidos	ciudad	edad
1	John	Smith	New York	24
6	Andrew	Thomas	New York	21

8 NULL

La palabra reservada **NULL** representa la ausencia de un valor en una tabla. Cuando hacemos un registro dejamos vacío un atributo de la tabla, SQL internamente le asigna NULL a ese valor vacío, NULL es diferente de un espacio en blanco y cero. Podemos comprobar si un registro tiene o no un atributo NULL con la combinación de comandos:

```
SELECT * FROM Personas
WHERE apellido IS NULL;
SELECT * FROM Personas
WHERE apellido IS NOT NULL;
```

9 Funciones

SQL posee algunas funciones integradas que nos permite conocer algunos detalles o características de las tablas.

9.1 COUNT

La función **COUNT()** regresa un número entero que representa el total de registros de una tabla, se le puede pasar el carácter asterisco o el nombre de alguna columna para que cuenten los registros:

```
SELECT COUNT(*) FROM Personas;
SELECT COUNT(apellidos) FROM PERSONAS;

// Ambas sentencias regresan 8.
```

Nota: los valores NULL son ignorados.

9.2 SUM

La función **SUM()** suma todos los valores numéricos (entero o decimal) de una columna, es necesario especificar el nombre de la columna a sumar dentro de los paréntesis de la función:

```
SELECT SUM(edad) FROM Personas

// Regresa 324.
```

Si se intenta utilizar esta función con textos, el valor de retorno es 0. La suma de valores NULL es NULL. Los valores NULL son ignorados.

Las funciones pueden ser combinadas con condicionales, como por ejemplo:

```
SELECT SUM(ciudad)
FROM Personas
WHERE ciudad = 'New York'

// Regresa 45.
```

La sentencia anterior selecciona los dos registros con el valor "New York" y suma los valores de sus atributos "edad".

9.3 AVG

Similar a la función anterior, **AVG()** regresa el promedio de una columna numérica:

```
SELECT AVG(edad) FROM Personas

// Regresa 40.5.
```

Nota: los valores NULL son ignorados. Si tiene 10 registros de los cuales 5 son NULL, solamente se saca el promedio de los 5 valores que no son NULL.

9.4 MIN & MAX

Las funciones **MIN()** y **MAX()** regresan el valor numérico mínimo y máximo de una columna respectivamente. Esta función puede ser utilizada para realizar operaciones aritméticas:

```
SELECT MIN(edad) FROM Personas;
SELECT MAX(edad) FROM Personas;

// Regresan 21 y 67 respectivamente.
```

9.5 UPPER

Vimos que las funciones anteriores simplemente regresan un valor, sin embargo, otras funciones pueden regresar una columna nueva de una tabla origen, un ejemplo de estas funciones es **UPER()**, la cual genera una columna con el nombre de la función como nombre y los caracteres de una columna original en mayúsculas, como se ve en la *Tabla 15*:

```
SELECT UPPER(nombre) FROM Personas
```

Table 15: Creando una columna personalizada con UPPER

upper
JOHN
DAVID
CHLOE
EMILY
JAMES
ANDREW
DANIEL
CHARLOTTE

Si buscamos generar esta columna con otro nombre que no sea el nombre de la función, podemos utilizar el comando AS, seguido del nuevo nombre, como se ve en la *Tabla 16*:

```
SELECT UPPER(nombre) AS nombre_mayus
FROM Personas
```

Table 16: Creando una columna personalizada con nuevo nombre con UPPER

nombre_mayus
JOHN
DAVID
CHLOE
EMILY
JAMES
ANDREW
DANIEL
CHARLOTTE

En el caso anterior, el nombre no tiene espacios en blanco, en caso de querer utilizarlos, el nombre de la columna debe estar encerrado en comillas simples. Si desea crear más de una columna personalizada en una sola sentencia, sepárelas con comas:

```
SELECT UPPER(nombre) AS 'Nombre mayus', edad + 1 AS Edad_Mas_Uno
FROM Personas
```

La sentencia anterior crea dos columnas personalizadas, utilizando una operación aritmética en la segunda para crear una columna con la edad de las personas más 1.

10 Sub-secuencias

Podemos utilizar sub-secuencias dentro de otra para obtener registros, estas sub-secuencias deben estar siempre encerradas entre paréntesis, como vemos en la *Tabla 17*:

```
SELECT * FROM Personas
WHERE edad >
    (SELECT AVG(edad) FROM Personas)
```

Table 17: Seleccionando edades mayor al promedio con una sub-secuencia

id	nombre	apellidos	ciudad	edad
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
7	Daniel	Harris	Los Angeles	67
8	Charlotte	Walker	Chicago	45

11 Agrupaciones

El comando **GROUP BY** permite agrupar valores de columnas o atributos duplicados en distintos registros en uno solo, de tal forma que el resultado final de la sentencia muestra solo una vez el valor del atributo duplicado. Veamos el siguiente ejemplo en la *Tabla 18* para que podamos visualizarlo de mejor manera:

```
SELECT ciudad, COUNT(*) AS c
FROM Personas
GROUP BY ciudad
ORDER BY c ASC
```

Table 18: Agrupando valores de atributos con GROUP BY

ciudad	**c**
Houston	1
Philadelphia	1
Los Angeles	2
Chicago	2
New York	2

Sabemos que tenemos tres usuarios que tienen el valor "New York", "Los Angeles" y "Chicago" respectivamente, por lo que estos valores están duplicados, en el ejemplo anterior seleccionamos la columna "ciudad" y, en vez de desplegar todos los valores de esta columna (incluso los duplicados), utilizamos el comando **GROUP BY** para agrupar los valores duplicados en uno, y añadimos el conteo de valores duplicados en una nueva columna personalizada nombrada "c".

Cuando se utiliza este comando, se seleccionan únicamente las columnas a agrupar, cada función que utilicemos en nuestra sentencia será aplicada al grupo, es decir:

```
SELECT [columna a agrupar 1], ..., [columna a agrupar n] [FUNCIÓN]
...
GROUP BY [columna a agrupar 1], ..., [columna a agrupar n]
```

Si queremos agregar otra columna a trabajar en el comando **GROUP BY** tendremos un error, solamente se utilizan las columnas seleccionadas tanto en el *SELECT* com en *GROUP BY*.

El comando **HAVING** es como el comando *WHERE*, pero aplicado a los grupos, retomemos el último ejemplo y veamos el nuevo resultado en la *Tabla 19*:

```
SELECT ciudad, COUNT(*) AS c
FROM Personas
GROUP BY ciudad
HAVING COUNT(*) > 1
ORDER BY c ASC
```

Table 19: Agrupando valores de atributos con GROUP BY y HAVING

ciudad	**c**
Los Angeles	2
Chicago	2
New York	2

Agrega la condición de que se desplieguen solamente los grupos con más de una persona habitando en las respectivas ciudades. Aclarando: *WHERE filtra los registros de una sentencia previo a agruparlos, HAVING filtra los grupos.*

12 Manejo de tablas

12.1 Creación de una tabla

Hasta ahora, siempre manejábamos una sola tabla propuesta por nosotros, sin embargo, el mundo real requiere que el desarrollador cree y manipule sus propias tablas, recodemos nuevamente que las tablas están constituidas por columnas (atributos) y filas (registros). Las columnas de la tabla "Personas" contiene solamente dos tipos de datos, cadena y número entero, la *Tabla 20* contiene el resto de tipos de datos disponibles para la creación de columnas en tablas:

Table 20: Tipos de datos disponibles en SQL

Nombre	**Descripción**
INT	Número entero positivo o negativo
FLOAT	Número decimal positivo o negativo
DOUBLE	Como *FLOAT*, pero con mayor rango de valores decimales disponibles
DATE	Una fecha en formato YYYY-MM-DD
DATETIME	Una fecha y hora que está en formato YYYY-MM-DD HH:MM:SS
TIMESTAMP	Una marca de tiempo, calculada desde la medianoche del primero de enero de 1970
TIME	Una hora en formato en HH:MM:SS
VARCHAR(largo)	Una cadena variable con largo definido dentro de los paréntesis
TEXT	Una cadena muy larga

A la hora de crear la tabla, debes especificar el tipo de dato que almacenará cada columna, por lo que debes pensar qué tipo de datos almacenarás para que corresponda con el tipo de dato asignado. Veamos la sintaxis para crear una tabla:

```
CREATE TABLE Clientes (
    id INT,
    firstname VARCHAR(128),
    lastname VARCHAR(128),
    salary INT,
    city VARCHAR(128)
);
```

Como se puede ver, las columnas que conformarán la tabla van encerradas entre paréntesis, separados por una coma y el tipo de dato se separa del nombre de la columna por un espacio, además, el tipo *VARCHAR* recibe el largo del dato a almacenar dentro de paréntesis también.

Si requerimos que, al momento de crear una tabla, se le asigne un valor predeterminado a una de las tablas, se puede utilizar el comando **DEFAULT**:

```
CREATE TABLE Clientes (
    id INT,
    firstname VARCHAR(128),
```

```
    lastname VARCHAR(128),
    salary INT DEFAULT 0,
    city VARCHAR(128)
);
```

Este comando va enseguida del tipo de dato y el valor va enseguida del comando, al ejecutar esta sentencia, se creará la tabla y todo registro nuevo en la misma tendrá el valor predeterminado de 0 (en caso de que no se le agregue un valor a la columna "salary"). Puede crear una tabla con múltiples valores predeterminados.

Si requerimos que todo registro, al momento de hacer un registro en una tabla, se llenen todos los campos de la misma, podemos utilizar el comando **NULL** y **NOT NULL**:

```
CREATE TABLE Clientes (
    id INT NOT NULL,
    firstname VARCHAR(128) NOT NULL,
    lastname VARCHAR(128),
    salary INT NOT NULL,
    city VARCHAR(128)
);
```

Ahora la tabla "Clientes" requiere obligatoriamente que los campos "id", "firstname" y "salary" sean llenados para el correcto registro.

12.2 Actualización de una tabla

El comando **ALTER TABLE** sirve para agregar, eliminar y modificar una columna en una tabla existe. Añadiremos las columnas "ex1" y "ex2" a la tabla "Clientes" con el comando **ADD**:

```
ALTER TABLE Clientes
ADD ex1 TEXT,
ADD ex2 TEXT;
```

Todos los registros existentes recibirán un valor predeterminado en estas nuevas columnas, el cual es **NULL**. Note que enseguida del comando *ADD* va el nombre de la columna y su tipo de dato, separados por un espacio. Agregar múltiples columnas en una sentencia indica que cada columna debe separarse por comas. Ahora eliminaremos la columna "ex2" con los comandos **DROP COLUMN**:

```
ALTER TABLE Clientes
DROP COLUMN ex2
```

Se borrará la columna y todos los datos almacenados en ella. También podemos cambiar el nombre de una columna existente o el nombre de la tabla con los comandos **RENAME** y **TO**:

```
ALTER TABLE Clientes
RENAME ex1 TO example1;

ALTER TABLE Clientes
RENAME TO Clients;
```

El aspecto final de la tabla "Clientes" es el siguiente (*Tabla 21*) después de las actualizaciones:

Table 21: Aspecto de la tabla "Clientes" después de varios cambios

id	firstname	lastname	salary	city

12.3 Eliminación de una tabla

El comando **DROP TABLE** permite borrar toda una tabla y su contenido, eliminaremos una tabla imaginaria "People" simplemente para mostrar la sintaxis de la sentencia:

```
DROP TABLE People
```

13 Manejo de datos

13.1 Inserción de registros

El comando **INSERT** crea un registro dentro de una tabla, vacía o con registros previos:

```
INSERT INTO Personas
VALUES (9,'Mike','Towers','Houston',39)
```

Este comando va acompañado del comando **INTO**, los valores que se vayan a registrar dentro de la tabla van encerrados dentro de paréntesis y son separados por comas, los valores que sean tipo cadena son contenidos dentro de comillas simples ('), los valores numéricos no requieren estas comillas. Asegúrese de que los valores a ingresar estén en el mismo orden de las columnas de la tabla.

El ejemplo anterior es la versión corta de la sentencia de inserción, pero podemos especificar a que columna va a insertarse un registro:

```
INSERT INTO Personas (id,nombre,apellidos,ciudad,edad)
VALUES (10,'Steve','Jobs','Philadelphia',41)
```

De esta forma, podemos escribir el código evitando escribir los valores a registrar en una columna que no le corresponde. Con esta sintaxis, podemos insertar valores a ciertas columnas únicamente, para que los valores predeterminados se inserten en el registro:

```
INSERT INTO Clientes (id,firstname,lastname,city)
VALUES (1,'Mario','España','CDMX')
```

Para este ejemplo, retomamos el último aspecto de la tabla "Clientes" que estuvimos creando y modificando en la sección anterior, aquella que tiene el valor 0 como predeterminado para la columna "salary", es por ello que la omitimos en este ejemplo de inserción, quedando el resultado en la *Tabla 22*:

Table 22: Inserción de registros en una tabla

id	firstname	lastname	salary	city
1	Mario	España	0	CDMX

Nota: si no escribe un valor para una tabla que es de rellenado obligatorio y no tiene un valor predeterminado, la sentencia *INSERT INTO* lanzará un error.

Si deseamos insertar varios registros en una misma sentencia, simplemente separe cada registro por comas y vea el resultado en la *Tabla 23*:

```
INSERT INTO Clientes (id,firstname,lastname,city)
VALUES
(2,'Arnulfo','Rodriguez',500.99,'BCS'),
(3,'Christopher','Robin',1999.87,'EDOMEX');
```

Table 23: Múltiple inserción de registros en una tabla

id	firstname	lastname	salary	city
1	Mario	España	0	CDMX
2	Arnulfo	Rodriguez	500.99	BCS
3	Christopher	Robin	1999.87	EDOMEX

13.2 Actualización de registros

El comando **UPDATE** actualiza los datos de un registro dentro de una tabla:

```
UPDATE Clientes
SET salary = 2010.56
WHERE id = 2
```

La actualización de registros dependen de una condición a la cual aplicarle las modificaciones, en el caso anterior se le aplica al registro que tiene un "id" de 2, pero se puede aplicar una condición que afecte a más de un registro (**si omite la condicional, todos los registros sufrirán la modificación**).

Como con la inserción, podemos actualizar múltiples datos separándolos por coma:

```
UPDATE Clientes
SET
salary = 2210.56,
city = 'BC'
WHERE id = 2
```

Quedando la tabla "Clientes" de la siguiente manera (*Tabla 24*) después de las inserciones y actualizaciones:

Table 24: Actualización de un registro en una tabla

id	firstname	lastname	salary	city
1	Mario	España	0	CDMX
2	Arnulfo	Rodriguez	2210.56	BC
3	Christopher	Robin	1999.87	EDOMEX

13.3 Eliminación de registros

El comando **DELETE** elimina los registros dentro de una tabla:

```
DELETE FROM Personas
WHERE id = 10
```

Tenga cuidado con la condicional, el caso anterior borra el registro que tiene el atributo "id" igual a 10, si pone una condicional que afecte a más de un registro, todos ellos serán eliminados. La tabla "Personas" queda de la siguiente manera (25) después de las inserciones y eliminación:

Table 25: Aspecto de la tabla "Personas" después de varios cambios

id	**nombre**	**apellidos**	**ciudad**	**edad**
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
4	Emily	Adams	Houston	29
5	James	Roberts	Philadelphia	31
6	Andrew	Thomas	New York	21
7	Daniel	Harris	Los Angeles	67
8	Charlotte	Walker	Chicago	45
9	Mike	Towers	Houston	39

A este conjunto de operaciones de creación, lectura, actualización y eliminación de registros se le conoce como **CRUD** (Create, Read, Update & Delete).

14 Más funciones

Para continuar con los siguientes temas, el nuevo aspecto de la tabla "Personas" aparece en la *Tabla 26*:

Table 26: Tabla "Personas" re acondicionada

id	**nombre**	**apellidos**	**ciudad**	**edad**
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
4	Emily	Adams	Houston	*NULL*
5	James	Roberts	Philadelphia	31
6	Andrew	Thomas	New York	21
7	Daniel	Harris	New York	67
8	Charlotte	Walker	Chicago	*NULL*
9	Samuel	Clark	San Diego	*NULL*
10	Anthony	Young	Los Angeles	52

14.1 De cadenas

Vimos que SQL posee algunas funciones especiales que permiten realizar conteos u operaciones a las columnas y su contenido, la *Tabla 27* contiene más funciones, enfocadas a las cadenas de texto, puede consultar el resto de funciones en este enlace:

Table 27: Funciones para cadenas

Función	**Descripción**
CONCAT(c1, c2, ..., cn)	Une dos o más cadenas o datos de distintas columnas
LOWER(columna)	Convierte todas las cadenas de una columna a minúsculas
SUBSTRING(columna, comienzo, fin)	Extrae una subcadena de una cadena, es decir, de todos los datos de una columna
REPLACE(columna, a-remplazar, remplazo)	Remplaza una cadena por otra en todos los datos de una columna
LENGTH(cadena)	Regresa el total de caracteres de una cadena
FORMAT(numero, decimales)	Regresa un número decimal con cierta cantidad de decimales

Puede estas funciones con las tablas mostradas en este documento para ver los resultados. Además, podemos utilizar funciones en conjunto:

```
SELECT CONCAT(
    SUBSTRING(nombre, 1, 1),
    '. ',
    UPPER(apellidos)) AS name
```

```
FROM Personas
```

La sentencia anterior toma la primer letra de cada registro de la columna "nombre" y las combina con cada registro de la columna "apellidos", todo en mayúsculas, quedando el resultado de la siguiente manera (*Tabla 28*):

Table 28: Uso de varias funciones en una sola sentencia

name
J. SMITH
D. WILLIAMS
C. ANDERSON
E. ADAMS
J. ROBERTS
A. THOMAS
D. HARRIS
C. WALKER
S. CLARK
A. YOUNG

14.2 Agregación y matemáticas

Ya hay punto en ese documento referente a las funciones matemáticas, la *Tabla 29* contiene los operadores aritméticos permitidos en SQL, puede consultar más en este enlace:

Table 29: Operadores aritméticos en SQL

Función	**Descripción**
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

Estos operadores pueden ser aplicados en sentencias para renombrar una columna de una cierta manera:

```
SELECT nombre, apellidos, edad / 2 AS menos
FROM Personas
```

Esta sentencia creará una columna con el nombre, apellidos y la mitad de edad de cada persona registrada. Otro ejemplo sería el siguiente:

```
SELECT nombre, apellidos, peso / (altura * altura) AS imc
FROM Personas
```

Suponiendo que en nuestra tabla "Personas" existen columnas llamadas "peso" y "altura" creamos una columna llamada "imc" con la división del peso entre el cuadrado de la altura, es decir, varias operaciones aritméticas en una sola sentencia.

15 CASE

Si recordamos de los lenguajes de programación, existen las estructuras condicional, como el *if* y *case*, SQL puede utilizar la estructura *case* mediante el comando **CASE** para asignar valores a registros según un criterio o condición. Veamos el siguiente ejemplo y el resultado en la *Tabla 30*:

```
SELECT nombre, apellidos,
CASE
    WHEN edad >= 65 THEN 'Senior'
    WHEN edad >= 25 AND edad < 65 THEN 'Adulto'
    ELSE 'Joven'
END AS categoria
FROM Personas
```

Table 30: Uso de CASE para asignación de valores

nombre	**apellidos**	**categoria**
John	Smith	Joven
David	Williams	Adulto
Chloe	Anderson	Senior
Emily	Adams	Joven
James	Roberts	Adulto
Andrew	Thomas	Joven
Daniel	Harris	Senior
Charlotte	Walker	Joven
Samuel	Clark	Joven
Anthony	Young	Adulto

El ejemplo anterior crear una columna personalizada llamada "categoria" con ayuda del comando *CASE*: el comando **WHEN** ayuda a poner la condición a seguir para asignar un valor, esta primera sub-sentencia asigna el valor "Senior" con el comando **THEN** a la columna si la edad es mayor igual a 65; la segunda sub-sentencia asigna el valor "Adulto" a la columna si la edad está entre 25 y 64, el comando **ELSE** asigna el valor "Joven" en caso de que las dos sub-sentencias anteriores no se cumplan (algo como el *if-else*); no olvide agregar el comando **END** al final de su CASE para indicarle a SQL que ese es el final de la condicional. Podemos utilizar cuantos comando *WHEN* deseemos y omitir el uso del comando *ELSE*.

16 Identidad

Todo este tiempo hemos trabajado con la tabla "Personas", esta posee la columna "id", esta columna es un identificador para cada registro en nuestra tabla, un identificador que diferencia un registro de otro. Esta columna suele ser del tipo entero (INT), no permite valores NULL y se llama "id", "Id" o "ID", el comando **AUTO_INCREMENT** permite que con cada nuevo registro insertado, el valor de la columna "id" auto incremente: si tenemos tres registro e insertamos uno nuevo, el id del nuevo registro será cuatro automáticamente.

Esta columna es muy importante en todas las tablas, porque el el índice que nos permite seleccionar registros de forma rápida sin tener que utilizar el resto de columnas (nombre, apellidos, ciudad y edad en este caso). Un ejemplo de como configurar esta columna "id" es el siguiente:

```
CREATE TABLE Pilotos(
    id INT NOT NULL AUTO_INCREMENT,
    nombres varchar(50),
    apellidos varchar(50)
);
```

De esta manera, cada nuevo registro tendrá un id automático auto incrementable y no deberemos especificarlo en la inserción:

```
INSERT INTO Pilotos(nombre, apellidos)
VALUES
    ('pancho', 'villa'),
    ('emiliano', 'zapata'),
    ('porfirio', 'diaz');
```

Esta tabla "Pilotos" queda como en la *Tabla 31*:

Table 31: Registros con id auto incrementable

id	nombres	apellidos
1	pancho	villa
2	emiliano	zapata
3	porfirio	diaz

El comando *AUTO_INCREMENT* tiene como valor de inicio el 1, se puede cambiar al momento de la creación de la tabla o ya creada:

```
-- Creando la tabla.
CREATE TABLE demo(
    id INT NOT NULL AUTO_INCREMENT = 100, ...
)

-- Modificando la tabla "Pilotos".
ALTER TABLE Pilotos
    AUTO_INCREMENT=200
```

Si la tabla fue creada previamente y se modifica el valor inicial de AUTO_INCREMENT, los siguientes registros a esta modificación ya serán con el nuevo valor inicial, los registros anteriores se mantendrán con el valor con el que fueron insertados.

17 Llaves Primarias y Secundarias

La **llave primaria** (**PRIMARY KEY**) de una tabla es una columna que permite la relación con otras tablas, por ejemplo: una sola persona puede tener múltiples números de celular, sería problemático tener varios registros de una persona con distinto número de celular, por lo que sería recomendable crear una tabla aparte con únicamente números de celular de cada persona; debemos hacer que tengan una relación las personas (tabla "Personas") con los números que poseen (nueva tabla "NumCel"), por lo que debemos establecer una llave primaria en alguna de las tablas para iniciar esa relación, ahí entra la columna "id", la cual generalmente es la llave primaria gracias a las siguientes características o reglas:

- Debe contener valores **únicos**.
- No debe tener valores **NULL**.
- Una tabla debe tener **solamente una** llave primaria.

Repetiremos la creación de la tabla "Personas" pero con la asignación de la llave primaria a la columna "id" y crearemos la tabla "NumCel" para ejemplificar esta lección:

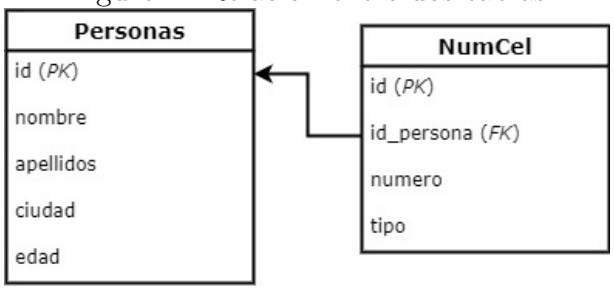
```
CREATE TABLE Personas(
    id INT NOT NULL AUTO_INCREMENT,
    nombre VARCHAR(255),
    apellidos VARCHAR(255),
    ciudad (VARCHAR255)
    edad INT,
    PRIMARY KEY (id)
);

CREATE TABLE PhoneNumbers (
    id INT NOT NULL AUTO_INCREMENT,
    id_persona INT NOT NULL,
    numero VARCHAR(55),
    tipo VARCHAR(55),
    PRIMARY KEY (id),
    FOREIGN KEY (customer_id) REFERENCES Personas(id)
);
```

El comando **PRIMARY KEY ()** permite asignar la llave primaria de una tabla a una columna. Ahora la tabla "Personas" tiene como llave primaria a la columna "id", mientras que la tabla "NumCel" tiene a "id" como llave primaria. Hasta este punto, todavía no hay relación entre ambas tablas, debemos mencionar lo que es una llave foránea.

Una **llave foránea** (**FOREIGN KEY**) es una columna que crea la relación entre las tablas, en una tabla Y se crea una columna con los valores de una tabla X para cada registro de la tabla Y que tengan relación. Veamos el ejemplo anterior de una manera más visual en la *Figura 1*:

Figure 1: Relación entre dos tablas



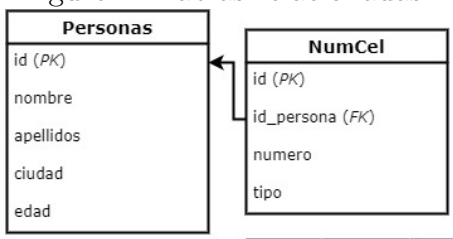
Ambas tablas tienen su llave primaria junto con sus registros, pero la tabla "NumCel" tiene una columna llamada "id_persona" que contiene los ids de las personas de la columna "id" de la tabla "Personas", por lo que esta columna es la que realmente hace la relación entre ambas columnas; la llave foránea de una tabla es la que conecta con la llave primaria de la otra tabla.

El comando **FOREIGN KEY ()** permite asignar la llave foránea de una tabla a una columna, analicemos el siguiente comando:

```
FOREIGN KEY (customer_id) REFERENCES Personas(id)
```

FOREIGN KEY (customer_id) asigna la llave foránea a la columna "customer_id" de la tabla "NumCel", y *REFERENCES Personas(id)* es la llave primaria de la tabla a relacionar, es aquí donde se crea la relación (en código) de ambas tablas. Insertamos algunos registros en la tabla "NumCel" y veamos visualmente como se ve el resultado en la *Figura 2*:

Figure 2: Tablas relacionadas



id	nombre	apellidos	ciudad	edad
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65

id	id_persona	numero	tipo
1	1	555123456	móvil
2	1	943554545	casa
3	2	331111111	móvil
4	2	88112233	trabajo
5	2	999001133	emergencia

Nota: una tabla puede tener múltiples llaves foráneas.

18 Llaves Únicas

Las **llaves únicas** (**UNIQUE**) son columnas que poseen valores únicos (no duplicados) pero no son el identificador ni llave primaria de la tabla, una tabla puede tener varias llaves únicas, pero una sola llave primaria. Haremos que la columna "apellidos" de la tabla "Personas" sea una llave única:

```
ALTER TABLE Personas
ADD UNIQUE apellidos
```

Ahora, cada apellido de cada registro no podrá estar duplicado, por lo que si intentamos insertar un registro con el apellido "Anderson" u alguno otro de esta tabla tendremos un error. Los valores *NULL* son ignorados en una llave única, por lo que podemos tener múltiples valores NULL en estas columnas.

19 Múltiples tablas

En el mundo real, existen miles de registros en una tabla, y un registro puede estar conectado con otra tabla, como en el ejemplo anterior de las personas y los números de celular.

El comando *SELECT, FROM y WHERE* permiten consultar información de dos tablas en una misma sentencia (*Tabla 32*):

```
SELECT nombre, apellidos, ciudad, numero, tipo
FROM Personas, NumCel
WHERE Personas.id = NumCel.id_personas
```

Table 32: Sentencia usando dos tablas

nombre	**apellidos**	**ciudad**	**numero**	**tipo**
John	Smith	New York	555123456	móvil
John	Smith	New York	943554545	casa
David	Williams	Los Angeles	331111111	móvil
David	Williams	Los Angeles	88112233	trabajo
David	Williams	Los Angeles	999001133	emergencia

Utilizar [nombre de tabla].[llave primaria] y [nombre de tabla].[llave foránea] es recomendable para hacer sentencias con más de una tabla, esto hace que las sentencias sean más fáciles de leer, este mismo formato aplica para consultas de una sola tabla:

SELECT PERSONAS.edad FROM Personas

El siguiente ejemplo es igual a un *SELECT * FROM Personas, NumCel* y muestra simplemente el uso del formato citado anteriormente:

```
SELECT Personas.nombre, Personas.apellidos, Personas.ciudad, Personas.edad, NumCel.numero,
     NumCel.tipo
FROM Personas, NumCel
WHERE Personas.id = NumCel.id_personas
```

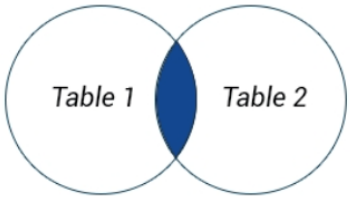
20 JOIN

Otra forma de combinar dos tablas en una consulta es por medio del comando **JOIN**, este comando funciona en base en condiciones, utilizaremos el ejemplo del tema anterior pero con *JOIN*:

```
SELECT nombre, apellidos, ciudad, numero, tipo
FROM Personas JOIN NumCel
ON Personas.id = NumCel.id_personas
```

FROM Personas JOIN NumCel hace la unión entre ambas tablas y *ON* establece la condición. Este comando tiene el aspecto de la *Figura 3*:

Figure 3: Concepto de JOIN



Los registros relacionados por la condición de *ON* en ambas tablas aparecen en la zona coloreada.

20.1 Alias

En el ejemplo donde recogemos todos los registros de todas las columnas, utilizamos el formato [nombre de tabla].[columna] el cual puede terminar siendo muy largo, es entonces que podemos abreviar el nombre de las columnas a una o pocos caracteres de la siguiente manera:

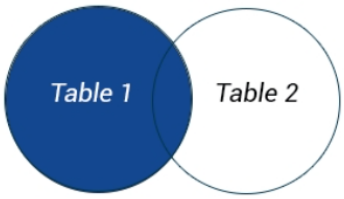
```
SELECT P.nombre, P.apellidos, P.ciudad, NC.numero, NC.tipo
FROM Personas AS P JOIN NumCel AS NC
ON C.id = NC.id_personas
```

Abreviamos el nombre de las tablas en el *SELECT* y definimos esta abreviación con *AS* en el *FROM*, a esto se le llama **alias de tablas**.

20.2 Tipos de JOIN

JOIN selecciona los registros que coinciden en ambas tablas, **LEFT JOIN** selecciona los registros que coinciden en ambas tablas y los de la primer tabla (tabla izquierda), el aspecto de *LEFT JOIN* está en la *Figura 4*:

Figure 4: Concepto de LEFT JOIN



RIGHT JOIN selecciona los registros que coinciden en ambas tablas y los de la segunda tabla (tabla derecha).

21 UNION

Este comando permite combinar los registros de ambas tablas; si en estas tablas hay dos o más registros iguales entre si (duplicados), solo se agrega uno de ellos y no sus duplicados.

Para combinar correctamente los registros, en la sentencia debe haber el mismo número de columnas, tipo de dato y que el orden de las columnas sea el mismo. Para ejemplificar este comando, partiremos la tabla "Personas" en dos y duplicaremos algunos registros, quedando como resultado las siguientes *Tablas*:

Table 33: Tabla "Personas" a la mitad

id	**nombre**	**apellidos**	**ciudad**	**edad**
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
4	Emily	Adams	Houston	*NULL*
5	James	Roberts	Philadelphia	31
6	John	Smith	New York	24
7	David	Williams	Los Angeles	42

Table 34: Tabla "Contactos"

id	**nombre**	**apellidos**	**ciudad**	**edad**
1	Andrew	Thomas	New York	21
2	Daniel	Harris	New York	67
3	Charlotte	Walker	Chicago	*NULL*
4	Samuel	Clark	San Diego	*NULL*
5	Anthony	Young	Los Angeles	52
6	Daniel	Harris	New York	67
7	Samuel	Clark	San Diego	*NULL*

El siguiente ejemplo con *UNION* dará como resultado la *Tabla 35*:

```
SELECT id, nombre, apellidos, ciudad, edad FROM Personas
UNION
SELECT id, nombre, apellidos, ciudad, edad FROM Contactos
```

Table 35: Unión de las tablas "Personas" y "Contactos"

id	**nombre**	**apellidos**	**ciudad**	**edad**
1	John	Smith	New York	24
2	David	Williams	Los Angeles	42
3	Chloe	Anderson	Chicago	65
4	Emily	Adams	Houston	*NULL*
5	James	Roberts	Philadelphia	31
6	Andrew	Thomas	New York	21
7	Daniel	Harris	New York	67
8	Charlotte	Walker	Chicago	*NULL*
9	Samuel	Clark	San Diego	*NULL*
10	Anthony	Young	Los Angeles	52

Este ejemplo utiliza dos tablas con la misma cantidad de columnas del mismo tipo, muestra todos los registros de ambas tablas menos los registros duplicados. El comando **UNION ALL** si incluye registros duplicados. En caso de que queramos unir dos tablas, pero alguna de ellas tiene columnas extras que queramos unir, podemos agregar dicha columna en alguna de las sentencias *SELECT*, en el siguiente ejemplo ponemos una columna llamada "trabajo" que solamente existe en la tabla "Contactos" (*Tabla 36*):

```
SELECT id, nombre, apellidos, ciudad, edad, trabajo FROM Contactos
UNION
SELECT id, nombre, apellidos, ciudad, edad, NULL FROM Personas
```

Table 36: Union de las tablas "Personas" y "Contactos"

id	nombre	apellidos	ciudad	edad	trabajo
1	John	Smith	New York	24	ingeniero
2	David	Williams	Los Angeles	42	*NULL*
3	Chloe	Anderson	Chicago	65	licenciada
4	Emily	Adams	Houston	*NULL*	*NULL*
5	James	Roberts	Philadelphia	31	*NULL*
6	Andrew	Thomas	New York	21	*NULL*
7	Daniel	Harris	New York	67	ingeniero
8	Charlotte	Walker	Chicago	*NULL*	*NULL*
9	Samuel	Clark	San Diego	*NULL*	*NULL*
10	Anthony	Young	Los Angeles	52	licenciado

Al tener la primer sentencia *SELECT* una columna extra, la segunda debe tener una columna extra para que sean el mismo número de columnas con el mismo tipo de dato, en este caso, asignamos una columna con solamente valores *NULL*.

Podemos utilizar condiciones dentro de las uniones:

```
SELECT id, nombre, apellidos, ciudad, edad FROM Personas
WHERE edad > 30
UNION
SELECT id, nombre, apellidos, ciudad, edad FROM Contactos
WHERE edad < 25
```