



DECODED

ACADEMY

FULL STACK DEVELOPER BOOTCAMP

Desarrollamos
{ talento }

SQL

Base de datos SQL

Structured Query Language (SQL), son bases de datos que nos permiten tener acceso a un sistema de gestión de base de datos relacionales.

Sistemas de Gestión de Bases de datos SQL

- DB2
- Firebird
- HSQL
- Informix
- Interbase
- MariaDB
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- PervasiveSQL
- SQLite
- Sybase ASE

MySQL

Instalación

Vamos a instalar de un paquete preconfigurado si tenéis Windows

Ampps



En caso de linux

```
sudo apt-get update  
sudo apt-get install mysql-server
```

Conceptos 1/2

- **Dato:** El dato es un trozo de información concreta sobre algún concepto o suceso
- **Tipo de dato:** El tipo de dato indica la naturaleza del campo.
- **Campo:** Un campo es un identificador para toda una familia de datos.
- **Registro:** Es una recolección de datos referente a un mismo concepto o suceso.
- **Campo Clave:** Es un campo especial que identifica de forma única a cada registro.
- **Tabla:** Es un conjunto de registros bajo un mismo nombre que representa el conunto de todos ellos.
- **Consulta:** Es una instrucción para hacer peticiones a bases de datos.

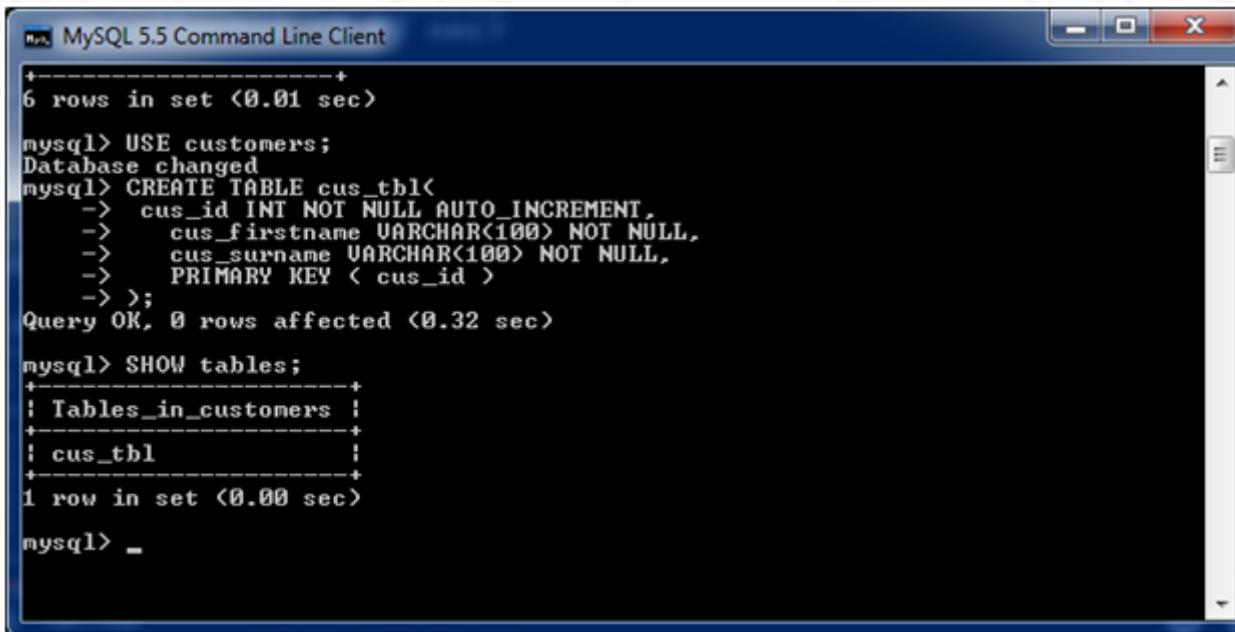
Conceptos 2/2

- **Índice:** Es una estructura que almacena los campos clave de una tabla, organizándolos para hacer más fácil encontrar y ordenar los registros de esa tabla.
- **Vista:** Es una transformación que se hace a una o más tablas para obtener una nueva tabla. Es una tabla virtual
- **Informe:** Es un listado ordenado de los campos y registros seleccionados en un formato fácil de leer
- **Guiones:** o Scripts. Son un conjunto de instrucciones que ejecutadas de forma ordenada, realizan operaciones avanzadas de mantenimiento de los datos almacenados en la base de datos
- **Procedimientos:** Son un tipo especial de script que está almacenada en la base de datos y que forma parte de su esquema

Estructura de la base de datos.

Una base de datos almacena los datos a través de un esquema. El esquema es la definición de la estructura donde se almacenan los datos, contiene todo lo necesario para organizar la información mediante tablas, registros y campos.

También contiene otros objetos necesarios para el tratamiento de los datos (Procedimientos, vistas, índices)



The screenshot shows a window titled "MySQL 5.5 Command Line Client". The terminal output is as follows:

```
+-----+  
6 rows in set <0.01 sec>  
  
mysql> USE customers;  
Database changed  
mysql> CREATE TABLE cus_tbl<  
    -> cus_id INT NOT NULL AUTO_INCREMENT,  
    -> cus_firstname VARCHAR(100) NOT NULL,  
    -> cus_surname VARCHAR(100) NOT NULL,  
    -> PRIMARY KEY ( cus_id )  
    -> >;  
Query OK, 0 rows affected (0.32 sec)  
  
mysql> SHOW tables;  
+-----+  
| Tables_in_customers |  
+-----+  
| cus_tbl             |  
+-----+  
1 row in set (0.00 sec)  
  
mysql> _
```

Herramienta que necesitamos

Podemos descargar el mismo en caso de Windows y Mac desde la página de [MySQL](#)

O en el caso de Linux desde el repositorio oficial

```
sudo apt-get install mysql-workbench
```

Tipos de datos en MySQL



GEEKSHUBS



Academy_

Podemos dividir en 3 grandes grupos estos datos:

- Numéricos
- Fecha
- String

Tipos de dato numéricos

- TINYINT: Ocupación de 1 bytes con valores entre -128 y 127 o entre 0 y 255.
- SMALLINT: Ocupación de 2 bytes con valores entre -32768 y 32767 o entre 0 y 65535.
- MEDIUMINT: Ocupación de 3 bytes con valores entre -8388608 y 8388607 o entre 0 y 16777215.
- INT (INTEGER): Ocupación de 4 bytes con valores entre -2147483648 y 2147483647 o entre 0 y 4294967295.
- BIGINT: Ocupación de 8 bytes con valores entre -8388608 y 8388607 o entre 0 y 16777215.
- DECIMAL (NUMERIC): Almacena los números de coma flotante como cadenas o string.
- FLOAT (m,d): Almacena números de coma flotante, donde 'm' es el número de dígitos de la parte entera y 'd' el número de decimales.
- DOUBLE (REAL): Almacena número de coma flotante con precisión doble. Igual que FLOAT, la diferencia es el rango de valores posibles.
- BIT (BOOL, BOOLEAN): Número entero con valor 0 o 1.

Tipos de dato fecha

- DATE: Válido para almacenar una fecha con año, mes y día, su rango oscila entre '1000-01-01' y '9999-12-31'.
- DATETIME: Almacena una fecha (año-mes-día) y una hora (horas-minutos-segundos), su rango oscila entre '1000-01-01 00:00:00' y '9999-12-31 23:59:59'.
- TIME: Válido para almacenar una hora (horas-minutos-segundos). Su rango de horas oscila entre -838-59-59 y 838-59-59. El formato almacenado es 'HH:MM:SS'.
- TIMESTAMP: Almacena una fecha y hora UTC. El rango de valores oscila entre '1970-01-01 00:00:01' y '2038-01-19 03:14:07'.
- YEAR: Almacena un año dado con 2 o 4 dígitos de longitud, por defecto son 4. El rango de valores oscila entre 1901 y 2155 con 4 dígitos. Mientras que con 2 dígitos el rango es desde 1970 a 2069 (70-69).

- CHAR: Ocupación fija cuya longitud comprende de 1 a 255 caracteres.
- VARCHAR: Ocupación variable cuya longitud comprende de 1 a 255 caracteres.
- TINYBLOB: Una longitud máxima de 255 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúsculas y mayúsculas.
- BLOB: Una longitud máxima de 65.535 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúsculas y mayúsculas.
- MEDIUMBLOB: Una longitud máxima de 16.777.215 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúsculas y mayúsculas.
- LONGBLOB: Una longitud máxima de 4.294.967.298 caracteres. Válido para objetos binarios como son un fichero de texto, imágenes, ficheros de audio o vídeo. No distingue entre minúsculas y mayúsculas.
- SET: Almacena 0, uno o varios valores una lista con un máximo de 64 posibles valores.
- ENUM: Igual que SET pero solo puede almacenar un valor.
- TINYTEXT: Una longitud máxima de 255 caracteres. Sirve para almacenar texto plano sin formato. Distingue entre minúsculas y mayúsculas.
- TEXT: Una longitud máxima de 65.535 caracteres. Sirve para almacenar texto plano sin formato. Distingue entre minúsculas y mayúsculas.
- MEDIUMTEXT: Una longitud máxima de 16.777.215 caracteres. Sirve para almacenar texto plano sin formato. Distingue entre minúsculas y mayúsculas.
- LONGTEXT: Una longitud máxima de 4.294.967.298 caracteres. Sirve para almacenar texto plano sin formato. Distingue entre minúsculas y mayúsculas.

LENGUAJE DDL

(LENGUAJE DE DEFINICIÓN DE DATOS)

Es un sub lenguaje de SQL, que permite la definición de datos, sus funciones son:

- Crear tablas, índices y otros objetos de la base de datos(como vistas, sinónimos, etc.)
- Definir la estructuras físicas donde se almacenarán los objetos de las bases de datos(espacios de tabla, ficheros de datos, etc.)

```
CREATE TABLE Actores (
    Codigo INTEGER PRIMARY KEY,
    Nombre VARCHAR(40),
    Fecha DATE,
    Nacionalidad VARCHAR(20),
    CodigoPersonaje INTEGER
);
```

El lenguaje tiene tres instrucciones básicas

- **CREATE tipo_objeto Nombre Definición** Crea un objeto de un determinado tipo(DATABASE, TABLE, INDEX, etc.)
- **DROP tipo_objeto Nombre** Elimina un tipo de objeto especificado mediante un nombre.
- **ALTER tipo_objeto Nombre Modificación** Modifica la definición de un objeto.

Creación de una base de datos.

```
CREATE {DATABASE | SCHEMA} [IF NOT EXIST] nombre_db  
[especificacion_create [, especificacion_create ] ...]  
especificacion_create:  
[DEFAULT] CHARACTER SET juego_caracteres  
[DEFAULT] COLLATE nombre_colación.
```

Creación de una base de datos

```
CREATE DATABASE alumnos CHARACTER SET Latin1 COLLATE latin1_spanish_ci;
```

Algunas instrucciones para tener en cuenta.

Para mostrar la relación de todas las base de datos.

```
SHOW DATABASES;
```

Para acceder a una de ellas.

```
USE [NOMBRE_BASEDEDATOS];
```

Modificación de una Base de datos

```
ALTER DATABASE alumnos COLLATE latin1_spanish_ci;
```

Borrando de la base de datos

```
DROP DATABASE alumnos;
```

Creación de tablas

```
CREATE [TEMPORARY] TABLE [esquema.] nombre_tabla
  [(definicion_create,...)]
  [opciones_tabla]
definicion_create:
  [definicion_columna]
  | [CONSTRAINT [símbolo] PRIMARY KEY (nombre_columna,...)
  | [CONSTRAINT [símbolo] FORGEIN KEY (nombre_columna,...)
  [definicion_referencia]
definicion_referencia:
  REFERENCES nombre_tabla [(nombre_columna, ...)]
    [ON DELETE {CASCADE | SET NULL | NO ACTION}]
    [ON UPDATE {CASCADE | SET NULL | NO ACTION}]
```

Vamos a crear una tabla con una clave primaria

```
create table pelicula(  
    isbn varchar(7) primary key,  
    titulo varchar(50),  
    director varchar(50),  
    precio numeric(4,2)  
) ;
```

Vamos a crear una tabla con una clave foránea

```
create table mascotas(
    codigo integer PRIMARY KEY,
    nombre varchar(50),
    raza varchar(50),
    cliente varchar(9),
    FOREIGN KEY (cliente) references clientes(dni)
);
```

Otras características de la creación de tablas

```
create table if not exists Bultos(  
    codigo int auto_increment primary key,  
    fecha datetime,  
    estado enum('Pendiente','Entregado','Rechazado')  
)  
comment = 'tabla de bultos'  
auto_increment = 100  
max_rows=100000  
checksum=1  
engine=innodb;
```

Otras instrucciones a tener en cuenta con las tablas

Consulta de las tablas de una base de datos

```
SHOW tables;
```

Consulta de una estructura de una tabla de la base de datos

```
describe [esquema.]nombre_tabla
```

Modificación de una tabla

```
ALTER TABLE nombre_tabla
    especificación_alter [, especificación_alter]...
especificación_alter:
    ADD definición_columna [FIRST | AFTER nombre_columna]
    | ADD (definicion_columna,...)
    | ADD [CONSTRAINT [símbolo]]
        PRIMARY KEY(nombre_columna, ...)
    | ADD [CONSTRAINT [símbolo]]
        UNIQUE (nombre_columna, ...)
    | ADD [CONSTRAINT [símbolo]]
        FOREIGN KEY (nombre_columna, ...)
        [definicion_referencia]
    | CHANGE [COLUMN] anterior_nombre_columna definición_columna
        [FIRST|AFTER nombre_columna]
    | RENAME COLUMN anterior_nombre_columna TO nuevo_nombre_columna
    | MODIFY definicion_columna [FIRST | AFTER nombre_columna]
    | DROP COLUMN nombre_columna
    | DROP PRIMARY KEY
    | DROP FOREIGN KEY fk_simbolo
    | opciones_tabla
```

Modificación de las tablas ejemplo

```
ALTER TABLE Mascotas ADD Especie VARCHAR(10) AFTER raza;
```

```
ALTER TABLE peliculas DROP PRIMARY KEY;
```

```
ALTER TABLE peliculas DROP ISBN;
```

```
ALTER TABLE peliculas ADD COLUMN codigo VARCHAR(10) PRIMARY KEY FIRST;
```

Borrado de Tablas

```
DROP TABLE Bultos CASCADE
```

-- Donde CASCADE no hace nada, pero se permite para facilitar migrar a mysql desde otros motores de bases de datos.

```
-- DROP TABLE Bultos CASCADE CONSTRAINT; //BORRA LAS CLAVES FORÁNEAS DE OTRAS TABLAS.
```

Renombrado de tablas

```
RENAME TABLE peliculas TO films;
```

LENGUAJE DML

Las sentencias DML utilizados en el lenguaje SQL las podemos establecer en:

- La sentencia SELECT, se utiliza para extraer información de la base de datos de una o varias tablas.
- La sentencia INSERT, cuyo objetivo es insertar una o varios registros en una tabla
- La sentencia DELETE, que borra registros.
- La sentencia UPDATE, que modifica registros en una tabla.

La sentencia SELECT

Es la sentencia más versatil y utilizada en todo SQL, el formato básico es:

```
select columna from tabla;
```

```
SELECT [DISTINCT] select_expr,[select_ expre]...FROM TABLA
select_expr:
nombe_columna[AS alias]
| *
| expression
```

TODOS LOS CAMPOS

Como obtener todos los registros, con todos los campos

```
select * from comunidades;
```

LA SENTENCIA SELECT

Podemos seleccionar los campos que queremos obtener, indicando el nombre de la columna.

```
SELECT comunidad FROM comunidades;
```

LA SENTENCIA SELECT

El parametro opcional **DISTINCT** fuerza que solo se muestren los registros con valores distintos.

```
SELECT DISTINCT nombrecolumna FROM comunidades;
```

LA SENTENCIA SELECT

Podemos ejecutar expresiones algebraicas

```
SELECT 5*10;
```

Además se puede incluir estas expresiones en cualquier consulta

```
SELECT comunidad, 5*10 FROM comunidades;
```

LA SENTENCIA SELECT

Podemos incluso ejectar tareas con las propias columnas y concatenando además podemos modificarlo a un nuevo nombre de columna

```
SELECT concat(comunidad, capital_id) as Comunidades from comunidades;
```

FILTROS EN LA SENTENCIA SELECT

Los filtros son condiciones que cualquier gestor de base de datos que interpreta para seleccionar registros y mostrarlos en la consulta.

Es uno de los elementos más importantes en las consultas SQL.

FILTROS EN LA SENTENCIA SELECT

La sintaxis de la cláusula SELECT

```
SELECT [DISTINCT] select_expr [, select_expr]...
      [FROM tabla] [WHERE filtro]
```

El filtro es una expresión que indica la condición o condiciones que deben satisfacer los registros para ser seleccionados.

FILTROS EN LA SENTENCIA SELECT

Expresiones para filtros podemos ejecutar diferentes tipos

```
// Expresión aritméticas.  
SELECT (10*5)+67;  
//Expresión de comparación.  
SELECT 4>6;
```

FILTROS EN LA SENTENCIA SELECT

Los elementos que pueden formar parte de estas expresiones son:

- Operandos
- Operadores aritméticos
- Operadores relacionales
- Operadores lógicos.
- Paréntesis.
- Funciones.

FILTROS EN LA SENTENCIA SELECT

Los operandos pueden ser constantes, variables, cadenas de caracteres, números, y también otras expresiones.

FILTROS EN LA SENTENCIA SELECT

Los operadores aritméticos puede ser:

- +,- -> Se utilizar para sumar y restar operandos.
- *,/ -> Se utilizan multiplicar y dividir dos operandos.
- % -> devuelve el resto de una división entre dos operandos.

FILTROS EN LA SENTENCIA SELECT

Los operadores relaciones, sirven para comparar dos operandos.

- <,> -> Se utiliza para comparar mayor y menor que.
- <> -> Se utiliza para comprobar si son diferentes los operandos que se comparan
- >= -> Se comprueba si los dos operandos es mayor o igual que.
- <= -> Se comprueba si los dos operandos es menor o igual que.
- = -> Nos indica si son iguales.

FILTROS EN LA SENTENCIA SELECT

Los operadores lógicos, se comportan como establece la siguiente tabla.

Op1	Op2	Op1 AND Op2	Op1 OR Op2	NOT Op1
falso	falso	falso	falso	verdadero
falso	verdadero	falso	verdadero	verdadero
verdadero	falso	falso	verdadero	falso
verdadero	verdadero	verdadero	verdadero	falso

FILTROS EN LA SENTENCIA SELECT

Los paréntesis (), hace que los operadores tengan una prioridad, por ejemplo

```
SELECT (4+5)*12;  
SELECT 4+5*12;
```

FILTROS EN LA SENTENCIA SELECT

Las funciones son específicas de cada SGBD, como por ejemplo:
concat, left, right, date_add ...

FILTROS EN LA SENTENCIA SELECT

Ejemplo de la utilización de algún operando.

```
SELECT * FROM customer WHERE (active=1 AND address_id < 10);
```

FILTROS EN LA SENTENCIA SELECT

También tenemos los **Filtros con operador de conjuntos** este operador permite comprobar si una columna tiene un valor igual que cualquier de los que están incluidos dentro del paréntesis

nombre_columna IN (Value1, Value2, ...)

```
SELECT * FROM customer WHERE active=1 AND address_id IN (1, 5, 6, 10);
```

FILTROS EN LA SENTENCIA SELECT

También tenemos los **Filtros con operador de rango BETWEEN** permite seleccionar los registros que están incluidos en un rango
nombre_columna BETWEEN Value1 and Value2

```
SELECT * FROM sakila.address WHERE postal_code BETWEEN 35200 AND 36000;
```

FILTROS EN LA SENTENCIA SELECT

Tenemos **Filtros con valor nulo**

Los operadores **IS** y **IS NOT** permiten verificar si un campo es o no es nulo

```
SELECT * FROM address where address2 IS null;
```

FILTROS EN LA SENTENCIA SELECT

Tenemos **Filtros con test de patrón**

Son filtros con un patrón para ello se usa los caracteres % y _

--Seleccionamos un patron

```
SELECT * FROM address where address like '962%';
```

--Seleccionamos con un caracter.

```
SELECT * FROM address where district like 'Ad_n%';
```

ORDENACIÓN EN SQL

Para ordenar los elementos utilizamos la cláusula **ORDER BY**

```
SELECT [DISTINCT] select_expr [,select_expr] ...
[FROM tabla]
[WHERE filtro]
[ORDER BY {nombre_columna | expr | posicion} [ASC | DESC] , ...]
```

ORDENACIÓN EN SQL

La cláusula nos permite ordenar el conjunto de datos de forma ascendente o descendente

```
-- ORDENAR DE FORMA ASCENDENTE.  
SELECT * FROM address ORDER BY postal_code ASC;  
-- ORDENAR DE FORMA DESCENDENTE.  
SELECT * FROM address ORDER BY postal_code DESC;
```

EXPRESIONES EN CONSULTAS SQL

Podemos ejecutar diferentes expresiones para calcular con funciones predefinidas:

- **SUM(Expresión)** -> Suma los valores indicados en el argumento
- **AVG(Expresión)** -> Calcula la media de los valores
- **MIN(Expresión)** -> Calcula el mínimo de los valores
- **COUNT(Columna)** -> Cuenta el número de valores de una columna excepto los nulos
- **MAX(Expresión)** -> Calcula el máximo.

EXPRESIONES EN CONSULTAS SQL

-- Saber la cantidad dinero de los alquileres todos.

```
SELECT SUM(amount) as total from payment;
```

-- Saber cual es el alquiler más caro.

```
SELECT MAX(amount) as ALQUILER_MAXIMO from payment;
```

-- Saber cual es el alquiler más barato.

```
SELECT MIN(amount) as ALQUILER_MINIMO from payment;
```

-- Saber la media de los alquileres.

```
SELECT AVG(amount) as ALQUILER_MEDIO from payment;
```

-- Conocer el número total de alquileres.

```
SELECT COUNT(*) as TOTAL_ALQUILERES from payment;
```

EXPRESIONES EN CONSULTAS SQL CON GROUP BY

Podemos agrupar las consultas utilizando **GROUP BY**

```
SELECT [DISTINCT] select_expr [, select_expr] ...
[FROM tabla]
[WHERE FILTRO]
[GROUP BY expr]
[ORDER BY {NOMBRE_COLUMN} ASC, DESC]
```

Ejemplo:

```
SELECT rental_duration, COUNT(*) as TOTAL  FROM film WHERE
rental_duration<5 GROUP BY rental_duration;
```

SUBCONSULTAS.

Las subconsultas me permiten realizar filtros con datos de otra consulta.

```
SELECT * FROM address WHERE city_id in(SELECT city_id FROM city WHERE city_id<10) ORDER BY city_id;
```

Podemos comprobar mediante el parámetro **EXIST** si la subconsulta devuelve algún tipo de dato

```
-- Que tenga relación  
SELECT * FROM address WHERE EXISTS(SELECT city_id from city WHERE  
city_id=1);  
-- Que no tenga relación  
SELECT * FROM address WHERE NOT EXISTS(SELECT city_id from city WHERE  
city_id=1);
```

Las subconsultas podemos incluir expresiones cuantificativas ALL o ANY

```
-- Mientras sea mayor que todos los resultados de la consulta  
SELECT * FROM rental where staff_id > ALL (select rental_rate from film  
where rating = "PG");  
-- Mientras que sea mayor que algún resultado de la consulta.  
SELECT * FROM rental where staff_id > ANY (select rental_rate from film  
where rating = "PG");
```

Podemos crear subconsultas anidadas

-- Encadenamos en diferentes subconsultas en diferentes tablas, con la condición que la subconsulta

```
SELECT * FROM address WHERE city_id = (SELECT city_id FROM city WHERE country_id LIKE (SELECT MAX(country_id) FROM country));
```

Consultas multitable

Una consulta multitable es aquella en la que se puede consultar información de más de una tabla. Se aprovechan los campos relacionados de las tablas para realizar la unión

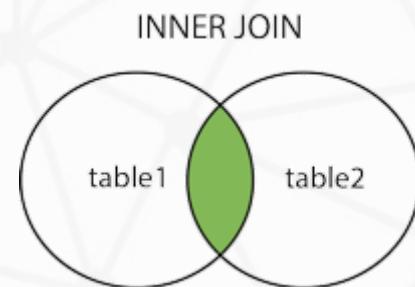
```
SELECT [DISTINCT] select_expr[,select_expr]...
[FROM referencias_tablas]
[WHERE filtro]
[GROUP BY expre [,expre]...]
[HAVING BY filtro_grupos]
[ORDER BY {nombre_columnas | expre | posicion} [ASC| DESC]]
```

Podemos establecer diferentes tipo de consultas multitable

- De equivalencia **INNER JOIN**
- Natural **NATURAL JOIN**
- **CROSS JOIN**
- Derecha **RIGHT OUTER JOIN**
- Izquierda **LEFT OUTER JOIN**
- Completa **FULL OUTER JOIN**

Composiciones internas con **INNER JOIN** se calcula el producto cartesiano de todos los registros, después, cada registro en la primera tabla es combinado con cada registros de la segunda tabla, y sólo se seleccionan aquellos registros que satisfacen las condiciones que se especifiquen. Hay que tener en cuenta que los valores NULOS no se combinan.

Podemos definir el inner join



Un ejemplo podemos comprobar las siguientes condiciones

El resultado será la concatenación de las filas en las que los valores comparados den verdadero.

```
SELECT * FROM city INNER JOIN country ON city .country_id =  
country.country_id;
```

Composiciones naturales con **NATURAL JOIN** es una especialización de las INNER JOIN, automáticamente recoge todas las columnas que tenga el mismo nombre en las dos columnas.

```
SELECT * FROM rental NATURAL JOIN customer;
```

Composición producto cartesiano **CROSS JOIN** nos establece una combinación cruzada.

El resultado de la consulta es la multiplicación de la cantidad de filas de ambas tablas, si se utiliza un where el resultado va a ser el mismo que el de un INNER JOIN.

```
-- Resultado de las tablas cruzadas.  
SELECT * FROM rental CROSS JOIN customer;  
-- funciona como un INNER JOIN  
SELECT * FROM rental CROSS JOIN customer WHERE rental.customer_id =  
customer.customer_id;
```

Composición externa **OUTER JOIN**

en este tipo de operación, las tablas no requieren que haya una equivalencia. El registro es seleccionado para ser mostrado aunque no haya otro registro que le corresponda.

Aparecen todos los registros de la tabla de la izquierda aunque no tengan correspondencia con los de la derecha, en caso de no haber correspondencia, los de la derecha aparecen como null.

```
SELECT * FROM rental LEFT OUTER JOIN payment ON (rental.customer_id = payment.customer_id AND rental.rental_id = payment.rental_id);
```

Aparecen todos los registros de la tabla de la derecha aunque no tengan correspondencia con los de la izquierda, en caso de no haber correspondencia, los de la izquierda aparecen como null.

```
SELECT * FROM rental RIGHT OUTER JOIN payment ON (rental.customer_id = payment.customer_id AND rental.rental_id = payment.rental_id);
```

Lenguaje DCL

Un Lenguaje de Control de Datos (DCL por sus siglas en inglés: Data Control Language) es un lenguaje proporcionado por el Sistema de Gestión de Base de Datos que incluye una serie de comandos SQL que permiten al administrador controlar el acceso a los datos contenidos en la Base de Datos.

INSERT

La sentencia **INSERT** permite insertar una fila en una tabla, es decir, añadir un registro de información a una tabla.

```
INSERT [INTO] nombre_tabla [(nombre_columna, ...)]  
VALUES ({expr| DEFAULT}, ...)
```

INSERT

Podemos utilizar el valor **DEFAULT** para asignar el valor por defecto a las columnas.

```
INSERT INTO actor VALUES(DEFAULT, 'SANTIAGO', 'SEGURA', DEFAULT);
```

INSERT EXTENDIDA

Los puntos suspensivos indican que podemos añadir más de una lista de valores.

```
INSERT [INTO] nombre_tabla [(nombre_columna, ...)]  
VALUES ({expr | DEFAULT}, ...), (...)
```

INSERT y SELECT

Nos permite seleccionar una serie de datos para posteriormente insertarla en la tabla.

```
INSERT [INTO] nombre_tabla [(nombre_columna, ...)]  
SELECT ... FROM ...
```

INSERT Y SELECT

Por ejemplo podemos seleccionar todas las filas de una tabla para realizar una copia

```
INSERT INTO BackupActores SELECT * FROM actor;
```

A la vez nos permite realizar consultas y aplicar filtros utilizando la potencia de la clausula **where**

UPDATE

La sentencia **UPDATE** permite modificar el contendio de cualquier columna y de cualquier fila de una tabla.

```
UPDATE nombre_tabla  
SET nombre_columna=expresión, [nombre_columna_n = expresión_n],  
[WHERE filtro]
```

Update

Actualización individual

```
UPDATE actor SET last_name = 'TORRENTE' WHERE first_name = 'SANTIAGO';
```

Tenemos que desactivar las actualizaciones seguras de tablas desde
Edit => SQL Editor => Other => Safe Updates

Update

Actualización completa, podemos actualizar todos los registros de una tabla obviando el filtro **WHERE**

```
UPDATE film SET rental_duration = 4;
```

Delete

La sentencia **DELETE** nos sirve para eliminar las filas de una tabla

```
DELETE FROM nombre_tabla  
[WHERE filtro]
```

Delete

Muy importante la clausula **WHERE** comprobar que se enuentra ya que nos puede generar un borrado masivo de filas

```
DELETE FROM actor WHERE first_name='SANTIAGO';
```

Sentencias con subconsultas

Es posible actualizar o borrar registros de una tabla filtrando a través de una subconsulta

```
DELETE FROM actor where actor_id Not in(select actor_id from film_actor);
```

Sentencias con relaciones

Cuando disponemos relaciones entre tablas las sentencias de update, o delete nos devolverá un error, para ello cuando creamos una tabla estableciamos la opción REFERENCES

```
REFERENCES nombre_tabla[ (nombre_columna, . . . ) ]  
    [ON DELETE opcion_referencia]  
    [ON UPDATE opcion_referencia]  
opcion_referencia:  
    CASCADE | SET NULL | NOT ACTION
```

En función de como se haya definido en el proceso de creación de la tabla el comportamiento será uno u otro

ORM - SEQUELIZE

¿Qué es un ORM?

Un ORM es una técnica de programación para convertir datos entre el sistema de tipos utilizados en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia.

Otros ORM

Tenemos distintas implementaciones para ORM en diferentes lenguajes

- PHP-> Doctrine
- Java -> Hibernate
- C# -> n-Hibernate
- Node.js -> Sequalize

¿Qué es Sequelize?

Sequelize es un ORM basado en promesas para Node.js, tiene ciertas características interesnates como es la sincronización, asociación, validación, etc. También tiene soporte para PostgreSQL, MySQL, MariaDB, SQLite y MSSQL.

Instalación de dependencias.

Instalación de sequelize.

```
$ npm install --save sequelize
```

Instalación del conector a BD

```
$ npm install --save mysql2
```

Preparando la conexión a nuestra BD

Preparando la conexión a nuestra BD

```
const Sequelize = require('sequelize');
const sequelize = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: 'mysql',
  operatorAliases: false,
  pool: {
    max: 5,
    min: 0,
    acquire: 30000,
    idle: 10000,
  },
});
// O BIÉN.
const sequelize = new
Sequelize('mysql://user:pass@example.com:3306/dbname');
```

Comprobando la conexión

```
sequelize
  .authenticate()
  .then(() => {
    console.log('Connection has been established successfully.');
  })
  .catch(err => {
    console.error('Unable to connect to the database:', err);
  });
}
```

Los modelos

El modelo es la representación de nuestro objeto para su persistencia en la base de datos, para ello definimos la estructura del mismo.

Introducción a ORM

Nuestro primer modelo

```
//Modelo
const User = sequelize.define('user', {
  firstName: {
    type: Sequelize.STRING,
  },
  lastName: {
    type: Sequelize.STRING,
  },
});

// force: true will drop the table if it already exists
User.sync({ force: true }).then(() => {
  // Table created
  return User.create({
    firstName: 'John',
    lastName: 'Hancock',
  });
});
```

Introducción a ORM

Tal como se puede comprobar nos introduce dos campos de forma natural `createAt` y `updateAt`

- `Sequelize.STRING`
- `Sequelize.TEXT`
- `Sequelize.INTEGER`
- `Sequelize.FLOAT`
- `Sequelize.DECIMAL`
- `Sequelize.BOOLEAN`
- `Sequelize.{...}`

Podemos ver los diferentes tipos de datos existentes de Sequelize en el siguiente enlace: [Definición de modelos y tipos](#)

Pirmera consulta con Sequelize.

Podemos utilizar un método predefinido de Sequelize para realizar consultas comunes como puede ser un findAll()

```
User.findAll().then(users => {
  res.send(users);
});
```

Query SQL con Sequelize

Nos permite también ejecutar raw SQL queries

```
sequelize.query('SELECT * FROM users WHERE id=1').then(user => {  
  res.send(user);  
}) ;
```

Buscando por ID

Cuando buscamos con `findById` nos devuelve el objeto concreto que coincide con el ID, en cambio `findAll()` nos devuelve un array independientemente de la cantidad de registros encontrados.

```
//Obtención de todos los registros.  
Usuario.findAll().then(users => {  
  res.send(users);  
});  
//Obtención de un registro por su id.  
Usuario.findById(1).then(user => {  
  res.send(user);  
});
```

WHERE

Filtros con las consultas.

```
Usuario.findAll({  
    where: { firstName: 'John' },  
}).then(user => {  
    res.send(user);  
});
```

Operadores

Tenemos una cantidad importante de [operadores](#)

- Op.and => &
- Op.or => |
- Op.gt => >
- Op.gte => >=
- Op.like => like

AND

```
const { Op } = Sequelize;
Users.findAll({
  where: {
    firstName: {
      [Op.and]: [{ [Op.like]: '_o%' }, 'John'],
    },
  },
}) .then(user => {
  res.send(user);
});
```

OR

```
const { Op } = Sequelize;
Users.findAll({
  where: {
    firstName: {
      [Op.or]: [{ [Op.like]: '_o%' }, 'Nombre Falso'],
    },
  },
}) .then(user => {
  res.send(user);
});
```

LIKE

Aplicando operadores a los filtros

```
const Op = Sequelize.Op;
Usuario.findAll({
  where: {
    firstName: {
      [Op.like]: '_o%',
    },
  },
}).then(user => {
  res.send(user);
});
```

FIND OR CREATE

Además de las consultas ya vistas, podemos ampliar las mismas con otros métodos que nos provee sequelize.

Con `findOrCreate` creará el registro en caso de no encontrarlo en nuestra base de datos.

```
Users.findOrCreate({
  where: { firstName: 'Pepe' },
  defaults: {
    lastName: 'Grillo',
  },
}) .spread((user, created) => {
  res.send(
    user.get({
      plain: true,
    })
  );
});
```

FIND AND COUNT ALL

Podemos realizar consultas con devolución del número total de registros.

```
// Únicamente los registros con nombre John hasta un máximo de 5
User.findAndCountAll({ where: { firstName: 'John' }, limit: 5
}).then(user => {
  res.send(user);
});

// Total de los usuarios y todos sus registros
User.findAndCountAll().then(user => {
  res.send(user);
});
```

MODIFICANDO EL RESULTADO



Sequelize nos permite también modificar el dataset de salida de la consulta

```
// Esto nos devolverá sólo 3 filas
Users.findAll({ limit: 3 }).then(user => {
  res.send(user);
});

// Esto nos devuelve los siguientes registros a partir del 10
Users.findAll({ offset: 10 }).then(user => {
  res.send(user);
});

//Nos lo ordena de forma Ascendente o Descendente.
Users.findAll({ order: [['id', 'DESC']] }).then(user => {
  res.send(user);
});

//Para agrupar las consultas.
Users.findAll({ group: ['id', 'firstName'] }).then(user => {
  res.send(user);
});
```

Consultas con los modelos

Después disponemos de elementos para extraer directamente columnas calculadas.

```
// Nos devuelve el valor máximo.  
Users.max('id').then(max => {  
  res.send({ max });  
});
```

```
// Nos devuelve el valor mínimo.  
Users.min('id').then(min => {  
  console.log(min);  
});
```

```
// Nos devuelve el campo calculado.  
Users.sum('id').then(sum => {  
  console.log(sum);  
});
```

Create

Podemos añadir nuevos registros con el método **Create**. Debemos pasarle los elementos que queremos crear y nos devuelve el nuevo registro de nuestra base de datos.

```
Users.create({  
  firstName: 'Antonio',  
  lastName: 'Miguel',  
}).then(user => {  
  res.send(user);  
});
```

BulkCreate

Creación de varios registros a la vez, mediante sequelize

```
Users.bulkCreate([
  {
    firstName: 'Pepe',
    lastName: 'Soler',
  },
  {
    firstName: 'Miguel',
    lastName: 'Soriano',
  },
]).then(() => {
  Users.findAll().then(users => {
    res.send(users);
  });
});
```

Update

Para actualizar cualquier registro para ello primero debemos localizarlo y posteriormente actualizar el mismo

```
Users.findOne({  
  where: { firstName: 'John' } ,  
}).then(user => {  
  user.updateAttributes({  
    lastName: 'Ruiz',  
  });  
  res.send(user);  
});
```

Update

Tenemos otro método de actualización mas directo

```
Users.update(  
  { firstName: 'Ivan' },  
  {  
    where: { firstName: 'John' }  
  }  
) .then(n => res.send(n));
```

Delete

Eliminar registros para ello utilizamos el método隐式的 **destroy**

```
Users.destroy({  
  where: { firstName: 'Miguel' },  
}).then(borrado => {  
  res.send('borrado');  
});
```

Sequelize devuelve 0 o 1 para indicar si se ha borrado con éxito

Relaciones

Podemos establecer relaciones también desde Sequelize.

- Relaciones 1: 1
 - belongsTo() -> la clave ajena se añade a la tabla origen
 - hasOne() -> la clave ajena se añade a la tabla destino
- Relaciones 1:N
 - hasMany() -> la clave ajena se añade a la tabla de destino
- Relaciones M:N
 - belongsToMany -> Es bidireccional añade un nuevo modelo que relaciona a ambos.

1:1

Relaciones 1:1 entre artistas y sus bandas.

```
var Artist = sequelize.define('artist', { name: Sequelize.STRING });
var Band = sequelize.define('band', {
  name: Sequelize.STRING,
  //artist_id: Sequelize.INTEGER
}) ;

Artist.sync({ force: true });
Band.sync({ force: true });
sequelize.query('SET GLOBAL FOREIGN_KEY_CHECKS = 0');
```

1:n

Relacion 1 a N entre ciudades y paises.

```
const City = sequelize.define('city', { countryCode: Sequelize.STRING });
const Country = sequelize.define('country', { isoCode: Sequelize.STRING });

// Here we can connect countries and cities base on country code
CountryhasMany(City, { foreignKey: 'countryCode', sourceKey: 'isoCode' });
City.belongsTo(Country, { foreignKey: 'countryCode', targetKey: 'isoCode' });
```