SQL y MySQL

Apuntes de lenguaje SQL y el sistema MySQL.

La información fue obtenida en las siguientes páginas, créditos a sus creadores:

- **Documentación - MySQL 8.0**
- **1keydata**
- **mysqltutorial**
- **guru99**
- **w3schools**

---

kurotom

Contenido

- SQL y MySQL
  - Características MySQL
- Lenguaje SQL
  - Data Definition Language (DDL)
  - Data Manipulate Language (DML)
  - Data Control Language (DCL)
- Características SQL
- Ventajas
- Desventajas
- Instalación
  - Relocate MySQL data
- ACID
- InnoDB
- Modelamiento de base de datos
- Definición de patrones para campos
- Estructura de datos
- Conceptos Database
- Tipos de datos
- Control de Acceso y Administración de cuentas
  - Privilegios
  - Roles
    - CREATE ROLE
    - DROP ROLE
    - SET DEFAULT ROLE
    - SET ROLE
    - CURRENT_ROLE
  - Administración de cuentas
    - CURRENT_USER
    - CREATE USER
    - ALTER USER
    - DROP USER
    - RENAME USER
    - SET PASSWORD
    - GRANT
    - REVOKE
    - SHOW GRANTS
- Backup and Recovery

  - Tipos de respaldos y recuperación
  - mysqldump
  - mysqlimport

- Básico
  - Comentarios
  - CREATE DATABASE
  - DROP DATABASE
  - USE DATABASE
  - SHOW DATABASES
  - SELECT DATABASES
  - ALTER DATABASE
  - SELECT
  - Valores Nulos y no Nulos
  - Funciones NULL
  - DISTINCT
  - WHERE
  - AND NAND OR NOR
  - IN
  - BETWEEN
  - LIKE
  - ORDER BY
  - GROUP BY
  - HAVING
  - ALIAS
  - JOIN
  - ON
  - CONCAT
  - SUBSTR
  - TRIM
    * LTRIM
    * RTRIM
  - LIMIT
  - TABLAS
    * CREATE
    * AS
    * Cargar Data en Tabla
    * Restricciones
      · AUTO_INCREMENT
      · Cambiar AUTO_INCREMENT
      · Obtener el último indice de la última fila agregada
      · Revisar las relaciones creadas
      · Habilitar o Deshabilitar comprobación FOREIGN KEY
    * CREATE VIEW

        * CREATE INDEX
            · SHOW INDEX
            · DROP INDEX
        * ALTER
        * DESCRIBE TABLE
        * DROP
        * TRUNCATE
        * INSERT
        * UPDATE
        * DELETE
        * RENAME

- TRANSACTION
    - COMMIT
    - ROLLBACK
    - Modo Autocommit
    - Habilitar o Deshabilitar AutoCommit
    - Uso de Transaction

- SQL Avanzado
    - UNION
    - UNION ALL
    - INTERSECT
    - MINUS
    - EXISTS
    - Triggers
        * Variable `NEW`
        * Variable `OLD`
        * Sintaxis TRIGGER
        * TRIGGER INSERT
        * TRIGGER UPDATE
        * TRIGGER DELETE
        * SHOW TRIGGERS
        * DROP TRIGGER
    - Stored Procedures
        * Parámetros en Stored Procedures
            · Parámetro IN
            · Parámetro OUT
            · Parámetro INOUT
        * Variables en Stored Procedures
        * DECLARE … HANDLER
            · Manejo de errores
            · Ejemplo Completo DECLARE HANDLER
        * Usar un stored procedure
        * Modificar Stored Pprocedure
        * DROP Stored Procedure
        * Listar Stored Procedure
        * SHOW CREATE PROCEDURE
    - Consultas Anidadas

  - Control de Flujo
    * IF ELSEIF ELSE
    * CASE
      · CASE NOT FOUND
    * ITERATE
    * LEAVE
    * LOOP
    * REPEAT
    * RETURN
    * WHILE
  - FUNCTION
    * DROP FUNCTION
    * Listar funciones

- Funciones MySQL
  - AVG
  - COUNT
  - FIRST
  - LAST
  - MAX
  - MIN
  - SUM
  - UCASE
  - LCASE
  - MID
  - LENGTH
  - FORMAT
  - DIV
  - CEIL
  - CEILING
  - FLOOR
  - RAND
  - ROUND
  - CONVERT
  - CURDATE
  - CURRENT_TIMESTAMP
  - YEAR
  - MONTH
  - DAY
  - MONTHNAME
  - DAYNAME
  - DATEDIFF
  - DATE_SUB
  - DATE_FORMAT
  - SUBSTRING

- DBA
  - Funciones DBA
  - Conexiones
  - Servicio MySQL
  - Optimization MySQL

        * Hardware y sistema operativo
            · RAID
            · Puede hacer
            · No puede hacer
    - Variables de ambiente MySQL
        * VARIABLES SERVIDOR
        * SHOW STATUS
        * SHOW VARIABLES
        * SET variable
        * Tipo de variables
    - Mecanismos de almacenamiento
        * SHOW ENGINES;
        * SHOW TABLE STATUS
        * MyISAM
            · Variables
            · Herramientas MyISAM
        * InnoDB
            · Variables
        * MEMORY
    - Costo de ejecución
        * EXPLAIN
        * EXPLAIN ANALYZE
    - mysqlslap
        * Uso mysqlslap

- Seguridad
- Resumen con Ejemplos

SQL y MySQL

MySQL es un sistema de gestión de bases de datos relacional desarrollado bajo licencia dual: Licencia pública general/Licencia comercial por Oracle Corporation y está considerada como la base de datos de código abierto más popular del mundo, y una de las más populares en general junto a Oracle y Microsoft SQL Server, todo para entornos de desarrollo web.

Lanzamiento inicial 1995, programado en C y C++, plataformas x86_64, IA-32, A64, Sun SPARC.

Última versión estable es la 8.0.29 (26 abril 2022).

Documentación MySQL - https://dev.mysql.com/doc/refman/8.0/en/

Básicamente MySQL se compone de:

1. Tablas
2. Views
3. Stored Procedures
4. Triggers

Características Mysql

- Servidor Robusto, multi-access, data integrity, Transaction control.
- Portabilidad, Windows/Linux, acceso usando lenguajes de programación.
- Multi-threads, facilita integración con hardware, mayor escalabilidad.
- Velocidad, sistema más rápido.
- Seguridad.
- Capacidad, hasta 65000 TB.
- Aplicabilidad, internet/desktop/corporativo.
- Logs, registra todo, recuperación, replica de servidores.

Lenguaje SQL

SQL (Structured Query Language) es un lenguaje de dominio específico, diseñado para administrar, y recuperar información de sistemas de gestión de bases de datos relacionales. Una de sus principales características es el manejo del álgebra y el cálculo relacional para efectuar consultas con el fin de recuperar, de forma sencilla, información de bases de datos, así como realizar cambios en ellas.

Apareció en 1974, extensiones comunes *.sql*.

En los 1970, laboratorios IBM definieron el lenguaje SEQUEL (Structured English Query Language).

SQL es una evolución de SEQUEL, en 1986 la ANSI estandarizó este lenguaje y lo nombró *SQL-86* o *SQL1*.

En 1987, ISO adoptó este estándar.

Data Definition Language (DDL)

Se encarga de la modificación de la estructura de los objetos de la base de datos. Ordenes modificar, borrar o definir tablas.

Operaciones:

- CREATE
- ALTER
- DROP
- TRUNCATE

Data Manipulate Language (DML)

Lenguaje proporcionado por el sistema de gestión de base de datos, permite realizar tareas de consulta o manipulación de datos, organizados por el modelo de datos adecuado.

Cada tipo de base de datos se diferencian en las funciones que tienen y en la forma que realizan los **Store Procedures**, que son colecciones pre-compiladas de sentencias SQL almacenadas dentro de la base de datos (sub-programas o sub-rutinas)

Un **Store Procedure** siempre contiene un *nombre*, *lista de parámetros*, y *sentencias SQL*, se invocan utilizando **Triggers**.

Operaciones:

- SELECT
- INSERT
- UPDATE
- DELETE

Stored Procedure

- Incrementan el rendimiento de la aplicación. Una vez que son creados, se compilan y almacenan en la base de datos.
- Reducen el tráfico entre aplicación y servidor de base de datos, debido a que la aplicación envía solamente el nombre del procedimiento almacenado y los parámetros en lugar de enviar múltiples sentencias SQL.
- Procedimientos almacenados son reutilizables y transparentes para cualquier aplicación.
- Siempre son seguros. El administrador puede garantizar permisos hacia aplicaciones que tengan acceso a procedimientos almacenados en la base de datos sin tener permisos a otras tablas de la base de datos.

Data Control Language (DCL)

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.

Algunos ejemplos de comandos incluidos en el DCL son los siguientes:

- **GRANT**: Permite dar permisos a uno o varios usuarios o roles para realizar tareas determinadas.
- **REVOKE**: Permite eliminar permisos que previamente se han concedido con GRANT.

Las tareas sobre las que se pueden conceder o denegar permisos son las siguientes:

- CONNECT
- SELECT
- INSERT
- UPDATE
- DELETE
- USAGE

Características SQL

- Lenguaje de definición de datos: El DDL de SQL proporciona comandos para la definición de esquemas de relación, borrado de relaciones y modificaciones de los esquemas de relación.
- Lenguaje interactivo de manipulación de datos: El DML de SQL incluye lenguajes de consultas basado tanto en álgebra relacional como en cálculo relacional de tuplas.
- Integridad: El DDL de SQL incluye comandos para especificar las restricciones de integridad que deben cumplir los datos almacenados en la base de datos.
- Definición de vistas: El DDL incluye comandos para definir las vistas.
- Control de transacciones: SQL tiene comandos para especificar el comienzo y el final de una transacción.
- SQL incorporado y dinámico: Esto quiere decir que se pueden incorporar instrucciones de SQL en lenguajes de programación como: C++, C, Java, PHP, COBOL, Pascal y Fortran.
- Autorización: El DDL incluye comandos para especificar los derechos de acceso a las relaciones y a las vistas.

Ventajas SQL

- Costos reducidos de aprendizaje.
- Portabilidad.
- Longevidad.
- Comunicación.
- Libertad de elección.

Desventajas SQL

- Falta de creatividad, fue creado sin vista a la posible evolución actual de la información.
- NoSQL.
- Falta de mayor estructuración.

Instalar MySQL

Utilizar versión *MySQL Community*, tanto para servidor *MySQL* como para *MySQL Workbench*.

En RHEL/CentOS/Fedora

- https://dev.mysql.com/downloads/
- https://dev.mysql.com/doc/refman/8.0/en/linux-installation-yum-repo.html
- https://dev.mysql.com/downloads/repo/yum/
- https://docs.fedoraproject.org/en-US/quick-docs/installing-mysql-mariadb/

Installing MySQL on Fedora

```
sudo dnf install community-mysql-devel community-mysql community-mysql-server

sudo dnf install mysql-workbench-community
```

Start MySQL Service and Enable at Loggin:

```
sudo systemctl start mysqld
sudo systemctl enable mysqld
```

Find Default Password, For security reasons, MySQL generates a temporary root key. Please note that MySQL has even stricter security policies than MariaDB.

```
sudo grep 'temporary password' /var/log/mysqld.log
```

Configuring MySQL before the first use

```
sudo mysql_secure_installation
```

Then, answer the security questions as you prefer. or just say yes to all of them.

Using MYSQL

```
sudo mysql -h ipHOST -P 3306 -u root -p
```

Removing MySQL

I suggest to remove in the following way, the most appropriate and safe way without removing many dependencies is:

```
sudo rpm -e --nodeps mysql-community-libs mysql-community-common mysql-community-server
```

Location files MySQL

For default, MySQL store database in **/var/lib/mysql/** on RHEL/CentOS/Fedora.

On Windows systems, **C:\Program Files\MySQL\[MySQL Server Version]\data**.

Relocate MySQL data

Por defecto MySQL escribe la información en el directorio `/var/lib/mysql/` en Linux y en Windows `C:\Program Files\MySQL\[MySQL Server Version]\data`.

Para cambiar el directorio por defecto se debe editar el fichero de configuración:

- `my.init` - Windows
- `my.conf` - Linux
- `/etc/my.cnf.d/community-mysql-server.cnf` - Linux - MySQL Community Edition.

Entorno Windows

Se debe detener el servicio MySQL, crear un directorio en la ruta deseada, copiar el directorio *Data* de la ruta por defecto en el nuevo directori.

Luego editar *my.init*, en la línea `datadir` con la nueva ruta. Establecer permisos para el nuevo directorio e iniciar el servicio.

Entorno Linux

1. Detener el servicio MySQL

```
sudo systemctl stop mysqld.service
```

2. Crear el directorio nuevo de MySQL.

```
mkdir /dir-mysql
```

3. Editar el fichero de configuración del servidor según sea su versión.

Si se está usando la versión *Community* se puede cambiar de nombre el fichero `community-mysql-server.cnf` para que no tome la configuración de este y utilice el fichero por defecto `/etc/my.cnf`.

O se comenta la línea `!includedir /etc/my.cnf.d` para que no incluya el directorio y use el fichero `my.cnf`.

```
sudo vim /etc/my.cnf.d/community-mysql-server.cnf
```

Agregamos las nuevas líneas, guardar y salir.

```
#
# This group is read both both by the client and the server
# use it for options that affect everything
#

[mysql]
socket=/dir-mysql/mysql.sock

[client-server]

[mysqld]
datadir=/dir-mysql
socket=/dir-mysql/mysql.sock

log-error=/var/log/mysql/mysqld.log
pid-file=/run/mysqld/mysqld.pid

#
# include all files from the config directory
#
#!includedir /etc/my.cnf.d
```

4. Copiar contenido del antiguo directorio al nuevo.

```
sudo cp -R /var/lib/mysql/* /dir-mysql/
```

5. Establecer propiedad y contexto SELinux.

```
sudo chown mysql:mysql -R /dir-mysql
```

```
sudo semanage fcontext -a -t mysqld_db_t "/dir-mysql(/.*)?"

sudo restorecon -RvF /dir-mysql
```

6. Iniciar el servicio.

```
sudo systemctl start mysqld.service
```

ACID

Acrónimo de **Atomicity**, **Consistency**, **Isolation** y **Durability**.

Estas propiedades son todas las deseadas en un sistema de base de datos, y están todas apegadas a la nocion de una transaction.

Características de transacciones de **InnoDB** se adhieren al principio ACID.

Transacciones son unidades *atomic* de trabajo que pueden ser *commited* o *rolled back*. Una transacción al crear múltiples cambios, estos pueden ser realizados todos cuando son exitosos o se deshacen cuando se revierten los cambios.

La base de datos mantiene un estado consistente todo el tiempo, después de cada *commit* o *rollback*, y mientras las transacciones están en progreso. Si la información relacionada se actualiza en múltiples tablas, las consultas ven los valores antiguos o los nuevos, no se mezclan datos viejos con nuevos.

Las transacciones están protegidas (asiladas) de otras mientras están en progreso, no se interfieren con operaciones que no hayan realizado *commit*. Esto se logra mediante el mecanismo de *bloqueo*. Usuarios experimentados pueden ajustar el nivel de aislamiento, cambiando la protección en favor de incrementar el rendimiento y *concurrencia*, cuando ellos garanticen que las transacciones no se interferirán.

Los resultados de las transacciones son durables, una vez que las operaciones realizadas son exitosas, los cambios están seguros de fallas de energía, fallo de sistema, condiciones *race* o cualquier otro daño potencial de cualquier otra aplicación no base de datos sea vulnerables.

Durabilidad, involucra escritura en el disco, con cierta cantidad de redundancia para proteger la información contra fallas de energía o fallas del servidor durante operaciones de escritura.

En InnoDB, **doublewrite buffer** ayudan a la durabilidad.

InnoDB MySQL

> InnoDB - doc

InnoDB es un motor de almacenamiento de propósito general que equilibra alta confiabilidad y alto rendimiento.

En MySQL es el motor de almacenamiento por defecto, a menos que se configure otro motor al momento de crear una tabla con la opción `ENGINE`.

Ventajas de InnoDB

- DML siguen el modelo ACID, con características de transacciones commit, rollback y con capacidad de recuperación de fallos para proteger la información del usuario.
- Bloqueo a nivel de filas y lecturas consistentes al estilo Oracle incrementa concurrencia y rendimiento multi-usuario.
- Tablas InnoDB arregla la información en el disco para optimizar las consultas basadas en llaves primarias. Cada tabla InnoDB tiene un índice de llave primarias llamadas *índice agrupado* que organiza la información para minimizar I/O para búsquedas de llaves primarias.
- Para mantener la integridad, InnoDB soporta restricción FOREIGN KEY. Con claves foráneas, inserciones, actualizaciones y eliminaciones son comprobadas para asegurar que no generen resultados inconsistentes a través de tablas relacionadas.

Estructura de datos

Estructura base de dato relacional - doc

Base de datos que se puede percibir como un conjunto de tablas y se puede manipular según el modelo relacional de los datos.

Cada base de datos incluye:

- conjunto de tablas de catálogo de sistema que describe la estructura lógica y física de los datos.
- archivo de configuración que contiene los valores de parámetro asignados a la base de datos.
- registro de recuperación con transacciones en curso y transacciones archivables.

Conceptos database

- Base de datos, conjunto de tablas que se pueden manipular mediante el modelo relacional de datos.
- Tabla, objeto de base de datos que contiene una colección de datos, compuesta de filas y columnas.
- Campos, son las columnas de las tablas, componente vertical, cada columna tiene un nombre y tipo de dato específico.
- Registros, son las filas de las tablas, componente horizontal, son una secuencia de valores uno por cada columna.
- Esquemas, agrupación de tablas que tengan una relación entre ellas.
- Vista, tabla lógica que se basa en datos almacenados en un conjunto subyacente de tablas. Los datos que se usan mediante sentencia *SELECT*. Alto costo de procesamiento.
- Índice, conjunto de punteros que están ordenados lógicamente según los valores de una clave, proporcionan acceso rápido a los datos y pueden imponer la exclusividad de los valores de clave para las filas en la tabla.
- Relación, enlace entre uno o más objetos que se crean utilizando sentencia JOIN.
- JOIN, operación relacional SQL que puede unir dos tablas mediante una condición de unión.
- Claves primarias, cuando asigna una clave primaria a un atributo, la clave identifica de forma exclusiva el objeto asociado a dicho atributo. El valor de la columna primaria determina qué atributos se utilizan para crear la clave primaria.
- Clave foránea, es la clave primaria de otra tabla, esta se puede relacionar entre tablas utilizando la clave única de cada tabla y mostrar los datos.
- Trigger, avisos automáticos que se ejecutan cuando ocurren modificaciones en la base de datos o en la tabla, ejecutando una función o procedimiento cuando la condición es satisfecha.

Modelamiento de base de datos

1. Análisis de los requisitos

- Entender las reglas de negocio.
- Realizar reuniones y entrevistas.
- Diseñar un modelo funcional y viable para poder trabajar.

2. Crear modelo conceptual

- Construir diagrama *Entidad-Relación*.
- Establecer la cantidad de entidades, es decir, si existen relaciones de tipo *Uno a Uno*, *Uno a Muchos* o *Muchos a Muchos*.

3. Atributos de entidades

- Construir un diagrama con las entidades y establecer las características de estas y sus relaciones entre las distintas entidades.

4. Transformar el diagrama de entidades en esquemas de tablas.

- Cada relación será una conexión entre tablas.

5. Construir la base de datos

- Podemos utilizar herramientas que ayudan a la creación de la base de datos, como Computer-Aide Software Engineering (CASE), Star UML, Astah, ERWin.
- Por línea de comandos o utilizando entorno gráfico como *MySQL WorkBench*.

Definición de patrones para campos

Al momento de la creación de una tabla, establecer campos por defecto es una buena práctica. Al momento de agregar una fila nueva y no se complete ese campo, automáticamente agregará el valor por defecto.

En los campos con fecha, se puede establecer el campo se auto-complete con la fecha del momento del ingreso de la fila.

Es recomendable crear un campo en las tablas que registre la fecha de creación de las filas nuevas para tener un control de registros, mediante el uso de `CURRENT_TIMESTAMP()`.

Tipos de datos

Algunos de los tipos de datos básicos de SQL son:

1. **Números enteros**

Uso en CREATE TABLE	Uso en query	características
TINYINT	TYNYINT(tamaño)	-128 a 127 normal. 0 a 255 sin signo. La cantidad máxima de dígitos se puede especificar entre paréntesis.
SMALLINT	SMALLINT(tamaño)	-32768 a 32767 normal. 0 a 65535 sin signo. La cantidad máxima de dígitos se puede especificar entre paréntesis.
MEDIUM	MEDIUMINT(tamaño)	-8388608 a 8388607 normal. 0 a 16777215 sin signo. La cantidad máxima de dígitos se puede especificar entre paréntesis.
INT	INT(tamaño)	-2147483648 a 2147483647 normal. 0 a 4294967295 sin signo. La cantidad máxima de dígitos se puede especificar entre paréntesis.
BIGINT	BIGINT(tamaño)	-9223372036854775808 a 9223372036854775807 normal. 0 a 18446744073709551615 sin signo. La cantidad máxima de dígitos se puede especificar entre paréntesis.

2. **Números en punto flotante**

Uso en CREATE TABLE	Uso en query	características
FLOAT	FLOAT(tamaño, d)	Un pequeño número con un punto decimal flotante. La cantidad máxima de dígitos se puede especificar en el parámetro de tamaño. El número máximo de dígitos a la derecha del punto decimal se especifica en el parámetro d.
DOBLE	DOBLE (tamaño, d)	Un número grande con un punto decimal flotante. La cantidad máxima de dígitos se puede especificar en el parámetro de tamaño. El número máximo de dígitos a la derecha del punto decimal se especifica en el parámetro d.

Uso en CREATE TABLE	Uso en query	características
DECIMAL	DECIMAL (tamaño, d)	Un DOBLE almacenado como una cadena, lo que permite un punto decimal fijo. La cantidad máxima de dígitos se puede especificar en el parámetro de tamaño. El número máximo de dígitos a la derecha del punto decimal se especifica en el parámetro d.

3. **Fechas y tiempos**

Uso en CREATE TABLE	Uso en query	características
DATE	DATE(fecha)	Una fecha. Formato: YYYY-MM-DD Nota: el rango admitido es de '1000-01-01' a '9999-12-31'
DATETIME	DATETIME (FECHA HORA)	Una combinación de fecha y hora. Formato: YYYY-MM-DD HH: MI: SS Nota: el rango admitido es de '1000-01-01 00:00:00' a '9999-12-31 23:59:59'
TIMESTAMP	TIMESTAMP (tiempo_timestap)	Una marca de tiempo. Los valores de TIMESTAMP se almacenan como el número de segundos desde la época de Unix ('1970-01-01 00:00:00' UTC). Formato: YYYY-MM-DD HH: MI: SS Nota: el rango admitido es de '1970-01-01 00:00:01' UTC a '2038-01-09 03:14:07' UTC
TIME	TIME (HH:MM:SS)	Un tiempo. Formato: HH: MM: SS Nota: el rango admitido es de '-838: 59: 59' a '838: 59: 59'
YEAR	YEAR (YYYY)	Un año en formato de dos o cuatro dígitos. Nota: Valores permitidos en formato de cuatro dígitos: de 1901 a 2155. Valores permitidos en formato de dos dígitos: 70 a 69, que representan los años de 1970 a 2069

4. **Cadena de caracteres**

Uso en CREATE TABLE	Uso en query	características
CHAR	CHAR (caracter)	Tiene una cadena de longitud fija (puede contener letras, números y caracteres especiales). El tamaño fijo se especifica entre paréntesis. Puede almacenar hasta 255 caracteres.
VARCHAR	VARCHAR (tamaño)	Tiene una cadena de longitud variable (puede contener letras, números y caracteres especiales). El tamaño máximo se especifica entre paréntesis. Puede almacenar hasta 255 caracteres. Nota: si agrega un valor mayor que 255, se convertirá en un tipo de texto.
TINYTEXT	TINYTEXT	Tiene una cadena con una longitud máxima de 255 caracteres.
TEXT	TEXTO	Tiene una cadena con una longitud máxima de 65.535 caracteres.
MEDIUMTEXT	MEDIUMTEXT	Tiene una cadena con una longitud máxima de 16,777,215 caracteres.
LONGTEXT	LONGTEXT	Tiene una cadena con una longitud máxima de 4.294.967.295 caracteres.
BLOB	BLOB	Para BLOB (Objetos grandes binarios). Almacena hasta 65.535 bytes de datos.
MEDIUMBLOB	MEDIUMBLOB	Para BLOB (Objetos grandes binarios). Tiene capacidad para 16.777.215 bytes de datos.
LONGBLOB	LONGBLOB	Para BLOB (Objetos grandes binarios). Tiene capacidad para 4.294.967.295 bytes de datos.
BINARY	BINARY(binario)	almacena valores binarios en una cadena de carácteres de valor fijo entre 0 y 255.
VARBINARY	VARBINARY(varbinary)	almacena valores binarios variables en una cadena de carácteres variable con valor variable de 0 y 255.

5. **Enum y set**

Uso en CREATE DATABASE	Uso en query	características
	Enum (x, y, z, etc.)	Permite ingresar una lista de valores posibles. Puede enumerar hasta 65535 valores en una lista ENUM. Si se inserta un valor que no está en la lista, se insertará un valor en blanco. Nota: los valores se ordenan en el orden en que los ingresas. Ingrese los valores posibles en este formato: ENUM ('X', 'Y', 'Z').
	Set	Similar a ENUM, excepto que SET puede contener hasta 64 elementos de lista y puede almacenar más de una opción.
		Para crear Base de datos o tablas.
	COLLATE	conjunto de carácteres que se va aceptar, utf-8, utf-16, etc. Para crear Base de datos o tablas.

6. **Binarios**

Uso en CREATE TABLE	Uso en query	características
BIT	BIT(bit)	Entero que puede ser 0 o 1 o NULL.

7. **Campos espaciales (GPS)**

Tipos espaciales - doc

- GEOMETRY
- LINESTRING
- POINT
- POLYGON

Control de Acceso y Administración de cuentas

MySQL permite la creación de cuentas que permiten a clientes conectarse al servidor y acceder a la información manejada por el servidor.

La función primaria del del sistema de privilegios de MySQL es autenticar a un usuario quien se conecta a un equipo y asociar a ese usuario con privilegios en la base de datos como SELECT, INSERT, UPDATE y DELETE. Además de permitir garantizar permisos para tareas administrativas.

La interfaz para cuentas de usuarios consisten en sentencias SQL: * CREATE USER * GRANT * REVOKE.

Los sistemas de privilegios garantizan que los usuarios realicen operaciones que están permitidas para ellos, estos ingresan a la base de datos mediante entrega de credenciales como una contraseña.

Internamente el sistema almacena la información, el servidor lee el contenido de las tablas en memoria cuando se inicia y el control de acceso a la base de datos toma las decisiones basadas en copias en memoria de las tablas *grant*.

Tiene dos partes:

1. El servidor acepta o rechaza la conexión basada en una identidad y verificada con una contraseña.
2. Si acepta, se comprueba si se tiene los privilegios suficientes para realizar las tareas que se requieran.

Limitaciones del sistema de privilegios

- No se puede explicitar que usuario no puede acceder, porque no se puede explícitamente coincidir con un usuario y rechazar su conexión.
- No se puede especificar que usuario tiene privilegios para crear o eliminar tablas en la database, pero no para crear o eliminar la database en sí.
- Una contraseña aplica globalmente a una cuenta, no se puede asociar una contraseña a un objeto específico como una base de datos, tablas o rutina.

Cuentas de usuario y contraseña

Se almacenan en tabla **user** de la base de datos **mysql**. Una cuenta se define en términos de:

- Nombre de usuario.
- Equipo al cual el usuario puede conectarse.

La cuenta puede autenticarse por contraseña, además MySQL soporta múltiples plugins de autenticación.

- Los usuarios del servidor MySQL no tiene relación con los usuario del sistema que hospeda al servidor MySQL ya sea Windows o Unx.
- MySQL soporta nombres hasta 32 carácteres de longitud.
- Autenticarse con métodos de autenticación incorporados se almacena en tabla **mysql.user**, es una contraseña totalmente diferente a del sistema operativo.
- Las contraseña almacenadas en **mysql.user** están encriptadas utilizando el algoritmo especificado o por defecto.
- Al utilizar carácteres ASCII, es posible que se conecte acorde a grupo de carácteres que utilice el sistema.

```
mysql -D dbName -h ipHOST -P port -u userName -p [password]
```

Privilegios

Los privilegios en MySQL determinan que operaciones puede realizar el usuario, tiene 3 niveles de operación:

1. **Privilegios Administrativos** : permite a los usuarios administrar operaciones en el servidor MySQL, son globales.
2. **Privilegios Database** : aplica a una base de datos y todo su contenido. Pueden ser específica o globales para todas las databases.
3. **Privilegios Objetos** : se especifica los privilegios a tablas, indices, views, rutinas almacenadas determinados dentro de una base de datos específica o son globales para los objetos del mismo tipo dentro de la misma base de datos.

Privilegios estáticos y dinámicos

> Privileges provided by MySQL - doc

MySQL soporta privilegios estáticos y dinámicos

- Privilegios estáticos están incorporados en el servidor, siempre están disponibles para garantizar a cuentas de usuarios y no se pueden quitar.
- Privilegios dinámicos pueden ser registrados y borrados. Un privilegio dinámico que no puede ser registrado no se puede asignar.

Roles

Un rol es una colección de privilegios, por ejemplo, un rol puede tener los privilegios de crear o eliminar cuentas de usuarios.

Capacidad de MySQL para administrar roles:

- CREATE ROLE y DROP ROLE.
- GRANT y REVOKE privilegios de roles y usuarios.
- SHOW GRANTS, mostrar privilegios asignados a un rol o usuario.
- SET DEFAULT ROLE, establecer roles por defecto a cuentas.
- SET ROLE, realizar cambios en los roles en una sesión activa.
- Mostrar el rol actual, CURRENT_ROLE().

CREATE ROLE

Un rol no tiene privilegios al ser creado, se debe garantizar mediante GRANT o revocar usando REVOKE.

```
CREATE ROLE 'nombre_rol';
```

- Garantizar los permisos.

```
GRANT [PERMISOS] ON nombreDB.[nombreTabla | *] TO 'nombre_rol';
```

En donde `[nombreTabla | * ]` indica una tabla en específico o `*` para todas las tablas.

- Revocar los permisos

```
REVOKE [PERMISOS] ON nombreDB.[nombreTabla | *] FROM 'nombre_rol';
```

En donde `[nombreTabla | * ]` indica una tabla en específico o `*` para todas las tablas.

- Asignar el rol a un usuario existente.

```
GRANT 'nombre_rol' TO 'nombreUser'@'ipUSER';
```

Siendo *ipUSER* el nombre del equipo en donde está el servidor MySQL.

Si el usuario necesita conectarse de varias direcciones, se debe crear un GRANT para cada ip que utilizará el usuario.

DROP ROLE

Elimina un o múltiples roles separados por coma.

```
DROP ROLE 'nombre_rol';
```

SET DEFAULT ROLE

```
SET DEFAULT ROLE [NONE | ALL | roleName] TO username;
```

- Ejemplo SET DEFAULT ROLE

```
SET DEFAULT ROLE 'admin', 'developer' TO 'joe'@'serverDB';
```

SET ROLE

```
SET ROLE [DEFAULT | NONE | ALL | ALL EXCEPT 'rolename', 'rolenameX' | rolename];
```

- Ejemplo SET ROLE

```
SET ROLE DEFAULT;
SET ROLE 'role1', 'role2';
SET ROLE ALL;
SET ROLE ALL EXCEPT 'role1', 'role2';
```

CURRENT_ROLE

Muestra el rol actual.

```
SELECT CURRENT_ROLE();
```

Administración de cuentas

MySQL permite crear, alterar, eliminar y garantizar o revocar permisos a usuarios.

Los usuarios y su información están en `mysql.user`. Los usuarios con permisos a la base de datos `mysql` pueden acceder a esta.

Es recomendable eliminar el usuario *root* y en su lugar crear un usuario nuevo que tenga los privilegios *root* (todos).

A los usuarios comunes, se debe seleccionar los permisos necesarios para que realice las tareas nada más, (*SELECT*, *INSERT*, *UPDATE*, *DELETE*, *CREATE TEMPORARY TABLES*, *LOCK TABLES*, *EXECUTE*). Este usuario puede ejecutar Stored procedures, Functions, Triggers, se debe tener cuidado en los *SP* debido a que si se ejecuta con este usuario puede alterar las tablas, dependiendo si el *SP* tiene algun *INSER*, *DELETE*, *UPDATE*, este usuario lo podrá ejecutar sin problemas.

Usuarios Backup para que realicen respaldos necesitan los permisos, *SELECT*, *RELOAD*, *LOCK TABLES*, *REPLICATION CLIENT*.

Usuarios solo lectura deben tener permisos *SELECT*, *EXECUTE*.

- CREATE USER
- ALTER USER
- DROP USER
- RENAME USER
- GRANT y REVOKE privilegios de roles y usuarios.
- SHOW GRANTS, mostrar privilegios asignados a un rol o usuario.
- Listar usuarios

CURRENT_USER

Muestra usuario actual.

```
SELECT CURRENT_USER();
```

CREATE USER

```
CREATE USER [IF NOT EXISTS]
'username'[@'ipUSER']
[authOPTIONS]
[DEFAULT ROLE roleName]
[WITH resourceOPTIONS]
[passwordOPTIONS]
[COMMENT comentario]
[ACCOUNT LOCK | ACCOUNT UNLOCK];
```

- authOPTIONS
  - IDENTIFIED BY 'auth_string'
  - INITIAL AUTHENTICATION IDENTIFIED BY {RANDOM PASSWORD | 'auth_string'

- resourceOPTIONS
  - MAX_QUERIES_PER_HOUR count
  - MAX_UPDATES_PER_HOUR count
  - MAX_CONNECTIONS_PER_HOUR count
  - MAX_USER_CONNECTIONS count
- passwordOPTIONS
  - PASSWORD EXPIRE [DEFAULT | NEVER | INTERVAL N DAY]
  - PASSWORD HISTORY {DEFAULT | N}
  - PASSWORD REUSE INTERVAL {DEFAULT | N DAY}
  - PASSWORD REQUIRE CURRENT [DEFAULT | OPTIONAL]
  - FAILED_LOGIN_ATTEMPTS N
  - PASSWORD_LOCK_TIME {N | UNBOUNDED}

```
CREATE USER 'nadie'@'serverDB' IDENTIFIED BY 'password';
```

ALTER USER

```
ALTER USER [IF EXISTS] userName
[authOPTIONS | passwordOPTIONS]
[WITH resourceOPTIONS]
[DEFAULT ROLE [NONE | ALL | roleName]]
[ACCOUNT LOCK | ACCOUNT UNLOCK];
```

- authOPTIONS
  - IDENTIFIED BY 'password' [REPLACE 'current_password'] [RETAIN CURRENT PASSWORD]
- passwordOPTIONS
  - PASSWORD EXPIRE [DEFAULT | NEVER | INTERVAL N DAY]
  - PASSWORD HISTORY {DEFAULT | N}
  - PASSWORD REUSE INTERVAL {DEFAULT | N DAY}
  - PASSWORD REQUIRE CURRENT [DEFAULT | OPTIONAL]
  - FAILED_LOGIN_ATTEMPTS N
  - PASSWORD_LOCK_TIME {N | UNBOUNDED}
- resourceOPTIONS
  - MAX_QUERIES_PER_HOUR count
  - MAX_UPDATES_PER_HOUR count
  - MAX_CONNECTIONS_PER_HOUR count
  - MAX_USER_CONNECTIONS count

```
ALTER USER 'username' IDENTIFIED BY 'password';

ALTER USER 'username'@'%' PASSWORD EXPIRE INTERVAL 180 DAY;

ALTER USER 'username' DEFAULT ROLE developer, administrador;
```

DROP USER

```
DROP USER [IF EXISTS] username[@'ipUSER'];
```

RENAME USER

Múltiples usuarios se deben separar entre comillas.

```
RENAME USER 'old_user'[@'ipUSER'] TO 'nsew_user'[@'ipUSER'];
```

SET PASSWORD

Permite asignar una contraseña a un usuario MySQL.

Utilizar ALTER USER en lugar de SET PASSWORD.

```
SET PASSWORD [FOR 'username'@'serverDB'] = 'password_String';
```

Establece a contraseña al usuario actual.

```
SET PASSWORD = "current_user_password";
```

GRANT

> Privileges MySQL - doc

Para garantizar el acceso a recursos se debe establecer privilegios para cada usuario, usando la identidad y la dirección de conexión hacia el servidor de estos, (conexiones).

```
GRANT [PRIVILEGIO] ON [database.tabla] TO 'username'@'ipUSER';

GRANT [ROLE] TO 'username'@'ipUSER';
```

- Ejemplo Privilegios root

```
GRANT ALL PRIVILEGES ON *.* TO 'nada'@'%' WITH GRANT OPTION;
```

- Ejemplo Privilegios Globales

Garantiza permisos para todas las base de datos y tablas en el servidor.

```
GRANT ALL ON *.* TO 'username'@'serverDB';
GRANT SELECT, INSERT ON *.* TO 'username'@'ipUSER';
```

- Ejemplo Privilegios Database

Garantiza permisos para una base de datos y su o sus tablas.

```
GRANT ALL ON myDB.* TO 'username'@'serverDB';
GRANT SELECT, INSERT ON myDB.* TO 'username'@'ipUSER';
```

- Ejemplo Privilegios Table

Garantiza permisos para una base de datos y una tabla tabla.

```
GRANT ALL ON mydb.mytbl TO 'someuser'@'somehost';
GRANT SELECT, INSERT ON mydb.mytbl TO 'someuser'@'ipUSER';
```

- Ejemplo Privilegios Columna

Garantiza permiso para una base de datos, en una tabla, en columna determinada.

```
GRANT SELECT (col1), INSERT (col1, col2) ON mydb.mytbl TO 'someuser'@'ipUSER';
```

- Ejemplo Grant Role

Garantiza un rol a un o unos usuario/s.

```
GRANT 'role1', 'role2' TO 'user1'@'ipUSER', 'user2'@'ipUSER';
```

REVOKE

Elimina privilegios en una base de datos y tablas para un usuario, roles de un usuario.

```
REVOKE [IF EXISTS] [privilegios] ON database.table FROM ['username'@'serverDB' | roleName];

REVOKE [IF EXISTS] ROLE FROM 'username'@'serverDB';
```

- Ejemplo REVOKE

```
REVOKE INSERT ON *.* FROM 'jeffrey'@'ipUSER';
REVOKE 'role1', 'role2' FROM 'user1'@'ipUSER', 'user2'@'ipUSER';
REVOKE SELECT ON world.* FROM 'role3';

REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'user4'@'localhost';
```

SHOW GRANTS

- Muestra los privilegios de un username.

```
SHOW GRANTS FOR '[username| role]'
```

- Muestra los privilegios de un usuario

```
SHOW GRANTS FOR 'username'@'hostServidor';
```

Listar usuarios

Muestra todos los usuarios registrados en el servidor MySQL.

```
SELECT user FROM mysql.user
```

Obtener las columnas de la tabla `mysql.user`.

```
DESCRIBE mysql.user
```

Backup and Recovery

Es importante tener respaldos de la información en caso de fallas de sistema, hardware, borrado accidentales de usuarios o razones externas.

Es buena práctica respaldar la data antes de realizar una actualización de *MySQL*.

Recordar respaldar los ficheros de configuración `my.ini` en Windows y `my.cnf` en Linux.

Bloquear y desbloquear un servidor MySQL en ejecución para realizar respaldos lógicos o físicos. Detener el servidor MySQL para realizar respaldos lógicos o físicos.

Herramienta mysqldump bloquea la base de datos a respaldar automáticamente.

Tipos de respaldos y recuperación

1. Respaldo físico (RAW)

Consisten en copias crudas de directorios y ficheros que almacenan el contenido de la base de datos. Este tipo de copia de seguridad es adecuado para grandes, base de datos importantes necesitan recuperarse rápidamente cuando ocurren problemas.

Se debe bloquear la base de datos para poder realizar un respaldo físico mientras está siendo ejecutado el servicio, LOCK UNLOCK.

Luego se puede copiar el directorio completo.

Características:

- El respaldo consisten de copias exactas del directorio de base de datos y ficheros.
- Son rápidos porque involucra el copiados de los ficheros sin necesidad de convertir estos ficheros.
- Utilizado en grandes base de datos.
- El respaldo y recuperación puede ser granulado, es decir, en el directorio de la base de datos, las tablas pueden ser ficheros por separados, permitiendo así respaldar y restaurar en operaciones separadas.
- Incluyen los ficheros *logs* y/o ficheros de configuración.
- Las tablas *MEMORY* son difíciles de respaldar porque no están almacenadas en el disco.
- Son portables solamente en máquinas iguales.
- El respaldo se realiza cuando el servidor MySQL *no está en ejecución*, si está ejecutándose se debe realizar bloqueos para que no se cambie la información en la base de datos durante el proceso.

2. Respaldo lógicos

Guardan la información representadas en una estructura lógica de base de datos (CREATE DATABASE, CREATE TABLE) y su contenido (INSERT o ficheros de texto).

mysqldump bloquea y desbloquea la base de datos automáticamente.

Este tipo de copia de seguridad es adecuado cuando se tiene pequeñas cantidades de información donde puedes editar los valores de información o estructura de tabla, o recrear la tabla en una máquina diferente.

Características:

- El respaldo se realiza mediante *queries* en el servidor MySQL.
- Es lento, porque se debe acceder al servidor y convertir la información en un formato lógico.
- La salida es más grande que respaldos físicos, particularmente cuando se guardan en ficheros de texto.
- La granularidad está disponible a:

  - nivel de servidor (todas las databases).
  - nivel de database (todas las tablas de la base de datos).
  - nivel de tabla.

- No incluyen los registros *log* y ficheros de configuración.
- Altamente portables, independiente de la máquina.
- Son realizados cuando el servidor está siendo ejecutado.
- Herramientas utilizadas como mysqldump, y la sentencia SELECT … INTO OUTFILE, guardan datos incluso los que están en *MEMORY*.
- Para restablecer respaldos lógicos, los ficheros *dump* pueden ser procesados por cliente `mysql`. Para data de ficheros de texto delimitados, usar LOAD DATA o mysqlimport.

Tabla comparando online y offline backup

Online backup	Offline backup
Se realizan cuando el servidor MySQL se está ejecutando (warm backup).	Se realizan cuando el servidor MySQL se está detenido (cold backup).
Es menos intrusivo con otros clientes, pueden acceder a la información dependiendo de lo que necesiten hacer.	Clientes están afectados porque el servidor no está disponible.
Se debe de cuidar de imponer bloqueos apropiados para evitar modificar la información mientras se respalda.	El procedimiento de respaldo es más simple porque no existe posibilidad de alterar la información debido a que no se tiene acceso a ella por el servidor detenido.

LOCK UNLOCK

LOCK UNLOCK - doc

Para realizar respaldo de la información se debe bloquear la base de datos para evitar conflictos datos durante el respaldo.

Una vez bloqueado se puede realizar respaldos usando comandos o mediante *MySQL Workbench*, una vez terminado, se debe desbloquear. (respaldos físicos y/o lógicos).

```
LOCK INSTANCE FOR BACKUP;

FLUSH TABLES [table_name1, table_nameX] WITH READ LOCK;


UNLOCK TABLES;

UNLOCK INSTANCE;
```

Mientras está bloqueado se puede realizar respaldo lógico, físico o ambos.

mysqldump

mysqldump - doc

Es un cliente que realiza respaldos lógicos, produciendo sentencias SQL que serán ejecutadas para reproducir las definiciones originales de objetos de base de datos e información de tablas. Puede generar ficheros *CSV*, ficheros de texto delimitados o XML.

```
mysqldump [OPCIONES] --databases db_name [tabla_name] > dump.sql
```

mysqlimport

mysqlimport - doc

Cliente provee interfaz CLI para sentencia LOAD DATA.

```
mysqlimport [OPCIONES] db_name fichero1 [fichero2 ...]
```

mysql

mysql - doc

Restaurar data mediante el uso del cliente.

```
mysql -h ipHOST -D nameDB -u usuario -p < fichero_dump.sql
```

SQL Básico

Operaciones básicas esenciales para poder usar el lenguaje SQL y MySQL.

Comentarios

Para establecer comentarios en SQL se debe utilizar `--`. Los comentarios se ignoran de la ejecución de SQL.

En Mysql se utiliza `/*  */`, `--`, incluso `#`.

Ejemplo

```
-- Selección de todo
SELECT * FROM tabla;

/* CONSULTAS */
```

CREATE DATABASE

CREATE DATABASE - doc

CREATE DATABASE o CREATE SCHEMA permite crear una base de datos.

Sintaxis

```
CREATE [DATABASE | SCHEMA]
[IF NOT EXISTS] nombre_database
[OPCIONES];
```

OPCIONES:

- `DEFAULT` : crea la base de datos con las opciones por defecto.
- `CHARACTER SET {codificación}` : establece el nombre de los carácteres que se utilizarán, *utf-8*, *utf-16*, etc.
- `COLLATE {nombre_collation}` : grupo de reglas que definen cómo se comparará y ordenará los strings.
- `ENCRYPTION ['Y' | 'N']` : encripta o no la base de datos, usando la encriptación por defecto. Mediante el uso de ALTER-DATABASE.

```
CREATE DATABASE IF NOT EXISTS
db_name
CHARACTER SET latin1
COLLATE latin1_swedish_ci
;
```

DROP DATABASE

DROP DATABASE o DROP-SCHEMA elimina la base de datos.

Sintaxis

```
DROP [DATABASE | SCHEMA] [IF EXISTS] nombre_database;
```

USE DATABASE

Utiliza una base de datos existente.

```
USE nombreDataBase;
```

SHOW DATABASES

Muestra las bases de datos existentes.

```
SHOW DATABASES;
```

SELECT DATABASES

Muestra la base de datos actualmente en uso.

```
SELECT DATABASES();
```

ALTER DATABASE

Modifica o cambia característica de una base de datos existente.

Sintaxis

```
ALTER DATABASE nombreDB [OPCIONES]
```

- **CHARACTER SET = charset_name** - establece un grupo de carácteres para la base de datos.
- **COLLATE = collation_name** - cambia la forma en que se realizarán las comparaciones.

- **ENCRYPTION = Y|N** - habilita o no la encriptación de la base de datos.
- **READ ONLY = 0|1** - establece el modo solo lectura para la base de datos.

```
ALTER DATABASE mydatabase COLLATE utf8_general_ci;
```

SELECT

Usado para realizar consultas a tablas.

Se puede unir tablas mediante uso de *WHERE*.

Sintaxis

```
SELECT * FROM nombre_tabla;

SELECT name, age, skills, city FROM date_personal;

SELECT datos.nombre, categorias.nombre
FROM datos, categorias
WHERE datos.categoria_id = categorias.id;
```

SELECT INTO

Permite almacenar los resultados en una variable o que sean escritos en un fichero.

- En una variable

```
SET @variable;
SELECT * INTO @variable FROM nombre_tabla;
```

SELECT INTO tiene una limitante, si se utiliza dentro de un bucle, este tomará el valor de la última iteración, no se creará una "lista de valores".

Para solventar esto, se utiliza un CURSOR que permite almacenar estos datos como si fuera un array.

- En un fichero

Para comprobar la ubicación por defecto en donde se guardarán los ficheros escritos, `SHOW VARIABLES LIKE "secure_file_priv";`

```
SELECT * FROM nombre_tabla
INTO OUTFILE '/var/lib/mysql-files/resultado.txt'
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

CURSOR

> CURSOR - doc

Es una estructura implementada en MySQL, que permite la interacción línea a línea mediante un orden determinado.

Utilizado dentro de un bucle LOOP, WHILE.

Teniendo el siguiente procedimiento:

- Declaración : definir la consulta que será depositada en el cursor.
- Abertura : abrir el cursor para recorrerlo.
- Recorrido : recorrer línea a línea hasta el final.
- Cierre : cerrar el cursor.
- Limpiar : limpiar el cursor de la memoria.

```
/*  declarar la variable para el cursor  */

DECLARE variableCursor TIPO_DATO;


/*  declarar el nombre del cursor  */

DECLARE nombreCursor CURSOR FOR [sentencia_SQL];


/*  abrir el cursor  */

OPEN nombreCursor;


/*  recorrer el cursor  y almacenar los datos*/
/*  en la variable del cursor  */
/*  debe usarse dentro de un bucle  */

FETCH nombreCursor INTO variableCursor;


/*  cerrar el cursor  */

CLOSE @variableCursor;
```

- *[sentencia_SQL]* si se quiere realizar una iteración específica se deben filtrar los datos usando WHERE o limitar la salida LIMIT o simplemente consultar y recorrer toda la tabla.
- `FETCH` soporta el uso de 1 o más campos, si se usan campos múltiples estos se deben las variables y los campos en la sentencia SQL que utilizará el CURSOR al momento de ser creado. Ejemplo múltiples campos.

Iterar un CURSOR

Para utilizar un CURSOR correctamente se debe implementar dentro de un bucle LOOP o WHILE.

Se puede utilizar *error handler*, este debe ubicar **después siempre después del cursor**.

Ejemplo, utilizando un Stored Procedure, dentro de un WHILE.

```
DELIMITER $$

CREATE PROCEDURE `cursor_looping` ()
BEGIN
DECLARE fin_c INT DEFAULT 0;
DECLARE varNombre VARCHAR(50);
```

```
/*  crear el cursor  */
DECLARE cursorTabla CURSOR for SELECT NOMBRE from tb_cliente;


/*  crear error handler para error de cursor  */
DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
    SET fin_c = 1;
END;

/*  abrir cursor  */
OPEN cursorTabla;

/*  iniciar while  */
WHILE fin_c = 0 DO

/*  iterar, obtener y almacenar los datos en la variable  */
FETCH cursorTabla INTO varNOMBRE;
IF fin_c =  0
    THEN SELECT varNOMBRE;
END IF;

END WHILE;

/*  cerrar el cursor  */
CLOSE cursorTabla;

END$$

DELIMITER ;
```

Llamar el *stored procedure*.

```
CALL cursor_looping();
```

Ejemplo un campo del CURSOR

El siguiente *stored procedure*, el cursor itera en todos los limites de créditos de los clientes, los sumas y muestra la suma total de estos.

```
DELIMITER $$

CREATE PROCEDURE limite_creditos ()
BEGIN
DECLARE total FLOAT DEFAULT 0;

DECLARE siguiente INT DEFAULT 1;

DECLARE variableCursor FLOAT DEFAULT 0;

DECLARE cursorIter CURSOR FOR SELECT LIMITE_CREDITO FROM tb_cliente;
```

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
    SET siguiente = 0;
END;


OPEN cursorIter;

WHILE siguiente = 1 DO

FETCH cursorIter INTO variableCursor;

IF siguiente = 1
    THEN SET total = total + variableCursor;
END IF;

END WHILE;

SELECT total;

CLOSE cursorIter;

END$$


DELIMITER ;
```

Ejemplo múltiples campos a un CURSOR

```
DELIMITER $$

CREATE PROCEDURE multiCampoCursor()
BEGIN

DECLARE continuar INT DEFAULT 1;

DECLARE varNOMBRE, varDIRECCION VARCHAR(100);
DECLARE varEDAD INT DEFAULT 0;


DECLARE iterador CURSOR FOR SELECT NOMBRE, EDAD, DIRECCION FROM tb_cliente;

DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
    SET continuar = 0;
END;

OPEN iterador;

WHILE continuar = 1 DO
FETCH iterador INTO varNOMBRE, varEDAD, varDIRECCION;
```

```
IF continuar = 1
    THEN SELECT CONCAT(varNOMBRE, ' -- ', varEDAD, ' -- ', varDIRECCION) AS resultado;
END IF;

END WHILE;

CLOSE iterador;

END$$


DELIMITER ;
```

Valores Nulos y no Nulos

`NULL` representa valores desconocidos. Estos no se pueden operar con operadores aritméticos, solamente se puede usar **IS**, **IS NOT**.

Sintaxis

```
SELECT * FROM personas WHERE apellido2 IS NULL;

SELECT name FROM personas WHERE name IS NOT NULL;
```

Funciones NULL

Si se quiere cambiar un **NULL** por algún valor, se debe usar **IFNULL**, **COALESCE**.

Sintaxis

```
SELECT producto, preciounidad * (
    unidadesstock + IFNULL(unidadespedido, 0)
)
FROM productos;
```

O usar:

```
SELECT producto, preciounidad * (
    unidadesstock + COALESCE(unidadespedido, 0)
) FROM productos;
```

En esta caso se cambio el valor NULL por 0.

DISTINCT

Devuelve los valores únicos de una consulta a una tabla.

Sintaxis

```
SELECT DISTINCT columna_nombre FROM nombre_tabla;
```

Ejemplo

```
SELECT DISTINCT nombre FROM personal;
```

WHERE

Condicionante de consulta.

Sintaxis

```
SELECT columna_nombre FROM nombre_tabla WHERE "condicion";
```

Ejemplo

```
SELECT nombre, skills FROM personal WHERE name = 'Bob';
```

AND NAND OR NOR

Clausula usada con `WHERE`. "Pipe" para condicionante `WHERE`, usado para realizar múltiples condicionales en una consulta.

Sintaxis

```
SELECT columna_nombre FROM nombre_tabla WHERE "condición" AND "condición";

SELECT columna_nombre FROM nombre_tabla WHERE "condición" OR "condición";
```

Operador de Comparación	Descripción
>	gt
>=	gte
<	lt

Operador de Comparación	Descripción
<=	lte
=	eq
<>, !=	not equal

Operador	Descripción
OR	operador or
NOR	operador not or
AND	operador and
NAND	operador not and

Ejemplo

```
SELECT name, age FROM personal WHERE name = 'Bob' AND (age > 20 AND age < 30);
```

IN

Clausula usada con `WHERE`. Similar a la sintaxis `WHERE columna_nombre = "valor"`, pero para múltiples valores. Se usa para múltiples valores.

Sintaxis

```
SELECT columna_nombre FROM nombre_tabla WHERE columna_nombre IN ("valor1", "valor2", ...);
```

Ejemplo

```
SELECT * FROM personal WHERE name IN ('Bob', 'Katy');
```

BETWEEN

Clausula usada con `WHERE`. Permite selección entre un rango de valores.

Sintaxis

```
SELECT columna_nombre FROM nombre_tabla
WHERE columna_nombre
BETWEEN "valor1" AND "valor2";
```

Ejemplo

```
SELECT * FROM personal WHERE skills BETWEEN 1 AND 3;
```

LIKE

Clausula usada con `WHERE`. Busca un patrón en la consulta realizada.

Sintaxis

```
SELECT columna_nombre FROM nombre_tabla WHERE columna_nombre LIKE patrón;
```

Uso de WILDCARDS

- `%` : sustituye a cero o más caracteres
- `_` : sustituye a 1 carácter cualquiera
- `[lista]` : sustituye a cualquier carácter de la lista
- `[^lista]` o `[!lista]` : sustituye a cualquier carácter excepto los caracteres de la lista

Donde, patrón:

- 'C_A' - todas las coincidencias que comienzan con "C" y terminan con "A", ejemplo: "casa".
- 'CEB%' - todas las coincidencias que comienzan con "CEB", ejemplo: "cebolla".
- '%AN' - todas las coincidencias que terminan en "AN", ejemplo: "pan".
- '%asa%' - todas las coincidencias que contienen "as", ejemplo: "casa", "brasas".

Ejemplo

```
SELECT * FROM personal WHERE name LIKE %%;
```

GROUP BY

Agrupa por columna la consulta realizada.

GROUP BY va antes que ORDER BY.

Utilizando funciones:

- AVG
- COUNT
- MAX
- MIN
- SUM

Al campo que se agrega la función se le debe agregar un alias **`AS nombre_alias`**, para identificarlo. Las columnas seleccionadas se deben utilizar para la agrupación. La función que se utiliza debe llevar un campo de la tabla y debe tener un alias.

Sintaxis

```
SELECT 'columna_nombre1', FUNCION('columna_nombre2') AS SUMA_TOTAL
FROM nombre_tabla GROUP BY 'columna_nombre1';

SELECT 'columna_nombre1', 'column_X', FUNCION('columna_nombre2') AS SUMA_TOTAL
FROM nombre_tabla GROUP BY 'columna_nombre1', 'column_X';
```

Ejemplo

```
SELECT store_name SUM(Sales) FROM Store_info GROUP BY store_name;
```

ORDER BY

Ordena el resultado de la consulta, ascendente o descendente, se pueden ingresar múltiples ORDER BY separadas por coma.

ORDER BY va después que GROUP BY

Las columnas nombradas al inicio de SELECT se utilizan una o todas en ORDER BY separadas por coma , siendo `ASC` o `DESC`.

Sintaxis

```
SELECT columna_nombre FROM nombre_tabla ORDER_BY columna_nombre [ASC, DESC];

SELECT columna_nombre FROM nombre_tabla WHERE condición ORDER BY columna_nombre [ASC, DESC];

SELECT columna_nombre FROM nombre_tabla
WHERE condición ORDER BY columna_nombre1 [ASC, DESC],
columna_nombre2 [ASC, DESC];
```

Ejemplo

```
SELECT name, age FROM personal ORDER BY age DESC name ASC;


SELECT name, age, SUM(salary) AS money
FROM nombre_tabla
WHERE age >= 20 GROUP BY name, age
ORDER BY name ASC, age DESC;
```

HAVING

Se usa en reemplazo de WHERE porque este no se puede usar con funciones agregadas.

Puede o no incluir GROUP BY y si existe **siempre va después** de GROUP BY.

Se puede incluir `ORDER BY` y otros operadores, como AND, OR, etc., operadores aritméticos.

Sintaxis

```
SELECT columna_nombre FROM nombre_tabla HAVING (condicion de función aritmética);
```

Ejemplo

```
SELECT Store_Name, SUM(Sales) FROM Store_Information
GROUP BY Store_Name HAVING SUM(Sales) > 1500;
```

Alias

Se pueden usar Alias de columna o Alias de tabla, existen para ayudar en la organización del resultado.

También se puede utilizar **AS** para crear un alias explícito.

Sintaxis

```
SELECT "alias_tabla"."columna_nombre" "alias_columna"
FROM "nombre_tabla" "alias_tabla";

SELECT "alias_tabla"."columna_nombre" AS "ALIAS_columna"
FROM "nombre_tabla" AS "ALIAS_tabla";
```

Ejemplo

```
SELECT A1.store_name store,SUM(A1.sales) "Total Sales"
FROM store_information A1 GROUP BY A1.store_name;
```

JOIN

Se usa para unir dos o más tablas basado en una columna relacionada entre ellas, también se pueden unir consultas SELECT mediante el uso de campos relacionados.

Se puede usar cualquier condicionante anterior, por ejemplo WHERE.

Deben tener campos en común.

Se pueden establecer múltiples JOIN de distintos tipos en una consulta.

Si se quiere realizar varios JOIN se deben escribir después de cada relación ON.

Tipos de `JOIN`

- `INNER JOIN` : retorna registro que coincidan en ambas tablas.
- `LEFT JOIN` : retorna registros desde la tabla izquierda y los registros coinciden desde la tabla derecha.
- `RIGHT JOIN` : retorna los registros desde la tabla derecha, y los registros coincidentes desde la tabla izquierda.
- `FULL JOIN` : MySQL no lo soporta, en su lugar se debe crear consultas con `LEFT JOIN` y `RIGHT JOIN`, uniéndolos usando UNION.

Sintaxis

```
SELECT columna_nombre1 FROM nombre_tabla1
INNER JOIN
nombre_tabla2 ON nombre_tabla1.columna_nombre = nombre_tabla2.columna_nombre;

SELECT columna_nombre FROM nombre_tabla
LEFT JOIN
nombre_tabla2 ON nombre_tabla1.columna_nombre = nombre_tabla2.columna_nombre;

SELECT columna_nombre FROM nombre_tabla1
RIGHT JOIN
nombre_tabla2 ON nombre_tabla1.columna_nombre = nombre_tabla2.columna_nombre;


/*  FULL JOIN  */
SELECT tablaA.nombre FROM tablaB.ciudad FROM tablaA LEFT JOIN tablaB
ON tablaA.id = tablaB.id
UNION
SELECT tablaC.rol, tablaD.ubicacion FROM tablaC RIGHT JOIN tablaD
ON tablaC.id = tablaD.id;
```

Ejemplo

Esta consulta muestra los clientes agrupados por el nombre, fecha (mes - año), la cantidad de compras por mes y año, la cantidad máxima de compra por cliente.

```
SELECT
A.DNI,
A.NOMBRE,
A.MES_AÑO,
A.CANTIDAD_MAXIMA - A.CANTIDAD_VENDIDA AS DIFERENCIA,
CASE
    WHEN A.CANTIDAD_MAXIMA - A.CANTIDAD_VENDIDA >= 0
THEN "VENTA VÁLIDA"
    WHEN A.CANTIDAD_MAXIMA - A.CANTIDAD_VENDIDA < 0 THEN "VENTA INVÁLIDA"
END AS ESTADO_VENTA
FROM (
    SELECT
    F.DNI,
    TC.NOMBRE,
    DATE_FORMAT(F.FECHA_VENTA, "%m - %Y") AS MES_AÑO,
```

```
    SUM(IFa.CANTIDAD) AS CANTIDAD_VENDIDA,
    MAX(TC.VOLUMEN_DE_COMPRA / 10) AS CANTIDAD_MAXIMA
    FROM
    facturas F
    INNER JOIN items_facturas IFa ON F.NUMERO = IFa.NUMERO
    INNER JOIN tabla_de_clientes TC ON TC.DNI = F.DNI
    GROUP BY F.DNI, TC.NOMBRE, DATE_FORMAT(F.FECHA_VENTA, "%m - %Y")
)
AS A;
```

ON

ON, usar con JOIN para coincidir con las mismas columnas, por ejemplo, ForeignKeys entre tablas.

```
SELECT ForeignTableDemo.Id,
ForeignTableDemo.Name,
PrimaryTableDemo.Address FROM ForeignTableDemo
JOIN
PrimaryTableDemo
ON
ForeignTableDemo.FK = PrimaryTableDemo.FK;
```

CONCAT

Concatenar los resultados de varios campos diferentes, los une para generar un registro.

Sintaxis

```
CONCAT (cadena1, cadena2, cadena3, ..)
```

Ejemplo

```
SELECT CONCAT (region_name, store_name) FROM geography WHERE store_name = 'Boston';
```

SUBSTR

Retorna un sub-string de una cadena.

Sintaxis

```
SUBSTR (str, pos)

SUBSTR (str, pos, len)
```

Ejemplo

```
SELECT SUBSTR (store_name, 3) FROM geography WHERE store_name = 'Los Angeles';

SELECT SUBSTR (Store_Name, 2, 4) FROM Geography WHERE Store_Name = 'San Diego';
```

TRIM LTRIM RTRIM

Funciones TRIM permite eliminar un prefijo o sufijo determinado de una cadena.

En MySQL:

- `TRIM(cadena)` : elimina los espacios en blanco de derecha e izquierda.
- `RTRIM(cadena)` : elimina los espacios en blanco de la derecha.
- `LTRIM(cadena)` : elimina los espacios en blanco de la izquierda.

Sintaxis

```
SELECT TRIM('  sample  ');

SELECT LTRIM('  sample  ');

SELECT RTRIM('  sample  ');
```

LIMIT

Especifica el número de filas a mostrar, este siempre va al final de la consulta, siempre.

Entregando dos parámetros, el primero indica hasta dónde es el corte y el segundo número indica hasta dónde mostrará después del límite.

Sintaxis

Se puede limitar por índice e índice y registros, siendo estos números empezando desde

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

SELECT * FROM tabla LIMIT [limite[, adicional]];
```

Ejemplo

Muestra el siguiente registro después del límite entregado.

```
SELECT * FROM registro LIMIT 2, 1;
```

Manipulación de Tablas

CREATE TABLE

Creación de tablas que almacenan información, usando los tipos de datos para crear las columnas.

Se permite crear relaciones mediante claves externas o claves foráneas al momento de la creación de esta.

Todas las instrucciones SQL que pueden tener una tabla están en Restricciones o Constraint

Sintaxis

```
CREATE TABLE nombre_tabla (
  columna_nombre1 datatype,
  columna_nombre2 datatype,
  columna_nombreX datatype,
);
```

Ejemplo

```
CREATE TABLE personal (
  nombre VARCHAR(50),
  apellido VARCHAR(50),
  edad INT(100),
  nacimiento DATE
);
```

FOREIGN KEY

Se pueden crear relaciones entre tablas utilizando FOREIGN KEY. Permitiendo borrar el registro creado al momento de eliminar el registro al cual se hace referencia mediante `ON DELETE CASCADE`.

```
CREATE TABLE nombre_tabla (
    columna_id INT AUTO_INCREMENT,
    columna_1 VARCHAR(255) NOT NULL,
    columna_related INT NOT NULL,
    PRIMARY KEY (columna_id),
    FOREIGN KEY (columna_related)
        REFERENCES tablaReferencia (columna_related)
        ON DELETE CASCADE
);
```

Ejemplo

```
CREATE TABLE personal (
  dni int(20),
  nombre VARCHAR(50),
  apellido VARCHAR(50),
```

```
  edad INT(100),
  nacimiento DATE,

  trabajoID int(20),

  PRIMARY KEY (dni),

  FOREIGN KEY (trabajoID)
        REFERENCES RolEmpresa (trabajoID)
        ON DELETE CASCADE
);
```

CREATE TABLE AS

Crear una tabla desde otra tabla existente, copiando los datos contenidos en esta última.

Sintaxis

Crear una tabla desde otra tabla.

```
CREATE TABLE new_table AS (SELECT * FROM old_table);
```

Crear tabla desde otra tabla usando algunas columnas.

```
CREATE TABLE new_table AS (SELECT column_1, column2, column_N FROM old_table);
```

Crear tabla desde otras tablas.

```
CREATE TABLE new_table
AS
(
    SELECT column_1, column2, column_N
    FROM
    old_table_1, old_table_2, old_table_N
);
```

Cargar Data en Tabla

LOAD DATA - doc

Después de crear una tabla se necesita llenar de datos, se puede realizar mediante INSERT INTO o cargando ficheros con información teniendo los campos con separaciones como *.csv*, *txt*.

```
LOAD DATA LOCAL
INFILE '/path/to/file'
INTO TABLE nombre_tabla
[LINES TERMINATED BY 'delimitadorFichero'];
```

delimitadorFichero es el delimitador que utiliza el fichero para separar los valores de los distintos campos. Algunos carácteres de separaciones más usados: * `\n` : nueva línea. * `\r` : carácter retorno. * `\t` : tabulación. * `\b` : carácter retroceso.

Restricciones o Constraint

Las restricciones se deben ingresar cuando se crea una tabla, luego se puede cambiar usando ALTER TABLE.

- **ZEROFILL** : llena con *0* los espacios restantes de un campo hasta su máximo determinado.
- **SIGNED** o **UNSIGNED** : establece el campo numérico debe llevar signo + o -.
- **NOT NULL** : valor no nulo, por defecto se aceptan valores `NULL`.
- **UNIQUE** : valor único, no se repiten.
- **CHECK** : valores de una columna cumplan condiciones.
- **DEFAULT** : establece un valor por defecto en una columna, usado en CREATE TABLE o ALTER TABLE, por ejemplo, `DEFAULT NULL` quiere decir que si no existe dato ingresado por defecto es nulo.
- **PRIMARY KEY** : identificar el campo de llave **identificador único y no nulo**, esta pueden tener más de una entrada estando separadas por coma(,). `PRIMARY KEY (COL_1, COL_2).`
- **AUTO_INCREMENT** : permite generar un número único cada vez que insertamos un nuevo registro en la tabla. AUTO_INCREMENT
- **FOREIGN KEY** : valor numérico que apunta a otro campo **PRIMARY KEY** de otra tabla, `ALTER TABLE nombre_tabla ADD CONSTRAINT FK_[nombreDescriptivo] FOREIGN KEY(DNI) REFERENCES cliente(DNI);`
- Puede existir más de una clave primaria en una misma tabla. La combinación de los campos que son clave primaria no se puede repetir en ningún otro registro de la tabla garantizando la integridad de los datos.
- La clave primaria ayuda a mantener la integridad de los datos al evitar duplicidad en los registros.

Sintaxis

```
CREATE TABLE Customer (
  SID integer NOT NULL ,
  Last_Name varchar (30) NOT NULL,
  First_Name varchar(30)
);


CREATE TABLE Customer (
  SID integer NOT NULL,
  Last_Name varchar (30) NOT NULL,
  First_Name varchar(30)
);


CREATE TABLE Customer (
  SID integer CHECK (SID > 0),
  Last_Name varchar (30),
  First_Name varchar(30)
);
```

```sql
CREATE TABLE Customer (
  SID integer,
  Last_Name varchar(30),
  First_Name varchar(30),
  PRIMARY KEY (SID)
);
```

```sql
CREATE TABLE ORDERS (
  Order_ID integer,
  Order_Date date,
  Customer_SID integer,
  Amount double,
  PRIMARY KEY (Order_ID),
  FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID)
);
```

AUTO_INCREMENT

AUTO_INCREMENT - doc

Atributo que es usado para generar identificador único para nuevas filas.

No es necesario establecer un valor en el campo auto-incremental.

Solamente se tiene un campo auto-incremental por tabla y tiene que ser la clave primaria.

Al insertar una fila nueva utilizando un valor arbitrario en el campo auto-incremental, el contador secuencial se establecerá en dicho valor y afectará a las nuevas filas agregadas.

Sintaxis

```sql
CREATE TABLE tabla (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(30) DEFAULT NULL,
    PRIMARY KEY (id)
);
```

Cambiar AUTO_INCREMENT

Permite establecer un nuevo valor inicial para el auto incremento.

```sql
ALTER TABLE tabla AUTO_INCREMENT = [NUMERO];
```

Obtener el último indice de la última fila agregada

```sql
SELECT LAST_INSERT_ID();
```

Revisar las relaciones creadas

Se pueden obtener todas las relaciones creadas mediante la consulta a la tabla y utilizando WHERE para filtrar los datos: `information_schema.TABLE_CONSTRAINTS`.

- Muestra todas las relaciones en la base de datos actual.

```
SELECT * FROM information_schema.TABLE_CONSTRAINTS;
```

- Filtra por nombre de tabla o columna las relaciones existentes.

```
SELECT
    TABLE_NAME,
    COLUMN_NAME,
    CONSTRAINT_NAME,
    REFERENCED_TABLE_NAME,
    REFERENCED_COLUMN_NAME
FROM
    information_schema.KEY_COLUMN_USAGE
WHERE
    REFERENCED_TABLE_SCHEMA = 'db_name'
    [AND REFERENCED_TABLE_NAME = 'table_name']
    [AND REFERENCED_COLUMN_NAME = 'column_name']
;
```

Habilitar o Deshabilitar comprobación FOREIGN KEY

Al deshabilitar, temporalmente, la comprobación permite eliminar tablas que tengan relaciones.

Tener cuidado de realizar dicha operación y activarla después.

```
/*  Habilita comprobación  */
SET FOREIGN_KEY_CHECKS=1;

/*  Deshabilita comprobación  */
SET FOREIGN_KEY_CHECKS=0;
```

CREATE VIEW

Tablas lógica o virtuales que se construyen a partir de una tabla existente, no almacena datos físicamente, algo así como un enlace simbólico a una tabla.

Puede ser usada para cualquier consulta, pero cada utilización de una VIEW impacta en el rendimiento, porque en cada uso se realiza las consultas para generar la VIEW.

Se pueden crear vistas desde otras vistas.

Se puede consultar los datos utilizando como una tabla común, para mostrar las tablas `SHOW TABLES;`.

Sintaxis

```sql
CREATE VIEW "nombre_tabla_virtual" AS instrucciones_sql;
```

Ejemplo

```sql
CREATE VIEW nombre_tabla_virtual AS SELECT name, age, city FROM persons;
```

CREATE INDEX

Los índices ayudan a obtener datos de tables de forma más rápida, pudiendo cubrir una o más de una columna. Es recomendable crear índices en tablas mediante comando CREATE INDEX o ALTER TABLE ADD INDEX.

Se crea una tabla auxiliar para la tabla original, esta tabla contiene dos columnas:

- La columna del valor a indexar.
- La columna que hace referencia a la posición en la tabla original o una *PK*.

Las consultas a los índices de una tabla debe tener el mismo tipo de dato para hacer efectiva el uso de estos.

Estructuras sin índices recorren toda la tabla hasta encontrar el registro.

InnoDB crea índice automáticamente al usar PRIMARY KEY. Crear un índice para usando una columna y su índice es la clave primaria. Índices que no son PRIMARY KEY poseen estructuras separadas y toman como referencia el valor de la *PK*. Si se crea un índice y no se usa la clave primaria como índice el rendimiento es igual a **MyISAM**.

MyISAM necesita de una tabla auxiliar para almacenar el índice, esta se debe actualizar siempre que se altere la tabla original. Es más lenta que **InnoDB**.

Por contraparte, muchos índices harán que las sentencias INSERT, UPDATE, DELETE sean más lentos.

Sintaxis

```sql
CREATE INDEX IDX_indice_nombre ON nombre_tabla (columna_nombre);
```

El uso del prefijo `IDX_` para "índice_nombre" es una práctica recomendada, o el uso de cualquier prefijo que identifique un índice.

Ejemplo

```sql
CREATE INDEX IDX_personal ON personal (name, skills);


ALTER TABLE nombre_tabla ADD INDEX  IDX_personal(name);
```

HASH y BTREE

Son algoritmos de búsqueda en listas ordenadas.

InnoDB implementa índices `B-TREE`. **MyISAM** implementa índices `HASH` y `B-TREE`.

B-TREE

Balanced - Binary Tree (nodos distribuidos de forma balanceada).

Mantiene los datos ordenados, las inserciones y eliminaciones se realizan en tiempo logarítmico armonizado.

Gran numero de llaves que pueden ser almacenados en un único nodo, cada nodo puede poseer más de dos hijos.

Permite menos uso de disco con resultados más rápidos de búsqueda e insercion.

Usado en base de datos y sistema de ficheros.

Comenzando por la raíz, va comparando el valor de la consulta con el valor del nodo, estos están dispuestos de la siguiente forma: * Valor izquierdos son menores. * Valores derechos son mayores.

HASH

Mapea datos grandes de tamaño variables en un tamaño fijo.

Usado en criptografía, almacenar contraseña, entre otros.

Búsqueda muy rápida en índices.

El valor hash generado se guarda en una tabla haciendo referencia a al valor indexado.

SHOW INDEX

- Listar todos los índices de una tabla.

```
SHOW INDEX FROM table_name FROM db_name;

SHOW INDEX FROM db_name.table_name;
```

- Listar todos los índices de un esquema.

```
SELECT
    DISTINCT TABLE_NAME,
    INDEX_NAME
FROM
    INFORMATION_SCHEMA.STATISTICS
WHERE
    TABLE_SCHEMA = `schema_name`;
```

DROP INDEX

```
DROP INDEX [nombre_indice] ON tabla_name;
```

```
ALTER TABLE tabla_nombre DROP INDEX nombre_index;
```

ALTER TABLE

Altera o modifica una tabla, agregando, eliminando, cambiando nombre de columna o el tipo de datos.

Se puede agregar llaves primarias, relaciones claves foráneas, columnas, etc. Revisar CONSTRAINTS

Instrucciones	Descripción	Ejemplo
ADD	agrega una columna.	ADD "columna_2" "tipo_de_datos"
DROP	elimina una columna.	DROP "columna_2"
CHANGE	cambia el nombre de columna.	CHANGE "antiguo_nombre" "nuevo_nombre" "tipo_de_datos"
MODIFY	cambia el tipo de datos de una columna	MODIFY "columna_2" "nuevo_tipo_dato"
RENAME	cambia el nombre de una tabla o columna	ALTER TABLE tabla RENAME nuevo_nombre;

Sintaxis

```
ALTER TABLE nombre_tabla ADD columna_X varchar(50);

ALTER TABLE nombre_tabla ADD PRIMARY KEY (columna);

ALTER TABLE nombre_tabla DROP columna_X;

ALTER TABLE nombre_tabla CHANGE columna_X columna_Y varchar(50);

ALTER TABLE nombre_tabla MODIFY columna_Y int(20);

ALTER TABLE nombre_tabla
ADD CONSTRAINT FK_[nombreDescriptivo]
FOREIGN KEY(CAMPO_TABLA)
REFERENCES tablaRelacionada(CAMPO_TABLA);

ALTER TABLE nombre_tabla ADD INDEX  IDX_columna(COLUMNA);

ALTER TABLE nombre_tabla DROP INDEX  IDX_columna;
```

RENAME Table

Cambia el nombre de una tabla o de columna.

Sintaxis

```
ALTER TABLE nombre_tabla RENAME nuevo_nombre;
ALTER TABLE nombre_tabla RENAME COLUMN viejo_nombre TO nuevo_nombre;
```

DROP TABLE

Elimina una tabla completa, todos los datos que contiene se pierden.

Sintaxis

```
DROP TABLE "nombre_tabla";
```

TRUNCATE TABLE

Elimina parte de los datos de una tabla, NO la tabla completa a diferencia de DROP.

Sintaxis

```
TRUNCATE TABLE "nombre_tabla";
```

DESCRIBE TABLE

Obtiene información sobre la estructura de una tabla.

```
DESCRIBE TABLE nombre_tabla;
```

INSERT INTO

Inserta información en una tabla especificada. Se puede usar cláusulas `WHERE`, `GROUP BY`, `HAVING`, también `JOIN` y `ALIAS`. Usar comillas simples para los strings.

Se pueden ingresar datos masivamente desde un archivo, otra base de datos, desde algún lenguaje de programación.

Sintaxis

1. Insertar datos poca cantidad.

Se pueden ingresar 1 o más de 1 cantidad de datos, respetando siempre los tipos de datos respectivos a las columnas.

```
INSERT INTO nombre_tabla (
  columna_1,
  columna_2,
  columna_X,
) VALUES (
  'data1',
  'data2',
  'dataX'
);

INSERT INTO nombre_tabla (
  columna_1,
  columna_2,
  columna_X,
)
VALUES
('data1', 'data2', 'dataX'),
('data1', 'data2', 'dataX');
```

2. Insertar datos en lote.

Traer los datos desde otra base de datos estando en otro servidor o en el mismo.

Puede existir diferencia de nombre de los campos, esto se soluciona mediante el uso de alias ALIAS.

- Se debe hacer un SELECT a la tabla de la otra base de datos.
- A cada campo diferente darles un ALIAS para que no existan problemas de coherencia.
- Luego que se haya creado el SELECT se debe utilizar WHERE comparando el campo único, usando el operador `NOT IN` y la query SELECT que consulte el código de la tabla destino.

```
INSERT INTO tabla_datos_destino
SELECT
CODIGO_DEL_PRODUCTO AS CODIGO,
NOMBRE_DEL_PRODUCTO AS DESCRIPCION,
SABOR,
TAMANO,
ENVASE,
PRECIO_DE_LISTA AS PRECIO_LISTA
FROM
origen_database.tabla_datos_origen
WHERE
CODIGO_DEL_PRODUCTO
NOT IN
(SELECT CODIGO FROM tabla_datos_destino);
```

3. Insertar datos desde ficheros externos

- Usando comandos:

```
mysql -D [nombreDB] -h ipHOST -P [PORT] -u [userDataBase] -p  < ficheroDumpData.sql
```

- Usando MySQL WorkBench

- Segundo clic en la tabla a importar los datos
- Seleccionar *Table Data Import Wizard*, buscar el fichero *csv* o *json*.
- Automáticamente seleccionará las columnas relacionadas, también se puede seleccionar manualmente.
- Se puede seleccionar para agregar o quitar campos, esto importará o no los datos del campo.
- Next, Finish.

UPDATE SET

Actualiza valor de un item dentro columna en una tabla. Ingresando múltiples columnas y valores separadas por coma,

Especificando la fila usando WHERE, dependiendo de la condición se puede alterar 1 fila o más filas.

Respetando el tipo de datos de las columnas.

Se puede hacer manualmente o de importando datos desde otra base de datos, esto último es útil para sincronizar datos.

Sintaxis

1. Actualizar de forma manual y por un determinado grupo de valores. Se pueden filtrar los registros a actualizar o aplicar a todos los registro la actualización.

```
UPDATE "nombre_tabla"
SET
"columna_nombre1" = "nuevo Valor1",
"columna_nombreX" = "nuevo ValorX"
WHERE
"condición";
```

Ejemplo

```
UPDATE Tabla_personal
SET
age = 30
WHERE nombre = 'Vito'
AND
apellido = "Corleone";


UPDATE tb_cliente
SET
VOLUMEN_COMPRA = VOLUMEN_COMPRA / 10;
```

2. Desde otra base de datos que actualice un o unos campos a una tabla de otra base de datos.

- Se debe crear un INNER JOIN para saber que datos tienen en común para actualizarlos, de no tener simplemente se igualan usando ALIAS. Los campos a igualar, después de ON, deben ser únicos (PRIMARY KEY).

```sql
SELECT * FROM tabla_Actualizar AS A
INNER JOIN
otra_base_datos.tabla_Datos_Origen AS B
ON
A.IDENTIFICACION = SUBSTRING(B.IDENTIFICACION,3,3)
```

- Actualizar los datos, son traídos desde otra tabla en otra base de datos o en la misma base de datos hacia una tabla que se quiera actualizar. Usando INNER JOIN para crear una relación y actualizar mediante `SET`.

```sql
UPDATE tabla_Actualizar AS A
INNER JOIN
otra_base_datos.tabla_Datos_Origen AS B
ON
A.IDENTIFICACION = SUBSTRING(B.IDENTIFICACION,3,3)
SET
A.ColumnaDato = B.ColumnaDato;
```

En este caso, la columna matricula de la otra base de datos tenia el campo con ZEROFILL, por lo que para igualar se utilizó la función SUBSTRING.

DELETE FROM

Para borrar un o múltiples registros de una tabla.

Se puede filtrar las filas utilizando WHERE y AND, NAND, OR, NOR. Al omitir WHERE se eliminan todas.

Se pueden borrar registros de una tabla para sincronizar otra tabla y que ambos tengan los mismos registros.

Sintaxis

1. Borrando registro manualmente.

```sql
DELETE FROM "nombre_tabla" WHERE "condición";


/* CUIDADO ELIMINA TODOS LOS REGISTROS*/
DELETE FROM nombre_tabla;
```

2. Borrar registro para sincronizar las tablas.

Esto borra registros que estén en la tabla destino y no en la tabla original, y viceversa.

```sql
DELETE FROM
tb_producto
WHERE
CODIGO
NOT IN
(SELECT CODIGO_DEL_PRODUCTO FROM jugos_ventas.tabla_de_productos);
```

```sql
DELETE A FROM tb_facturas A
INNER JOIN
tb_clientes B
ON A.DNI = B.DNI
WHERE B.EDAD < 18;
```

Ejemplo

```sql
DELETE FROM personal WHERE nombre = 'Fredo' and apellido = 'Corleone';
```

TRANSACTION

> Transaction - doc

Son unidades atómicas de trabajo que pueden ser **commited** o **rolled back**. Cuando una transacción realiza múltiples cambios en la base de datos, los cambios exitosos se realizan todos o todos los cambios se deshacen cuando la transacción se revierte.

Las transacciones son implementadas por **InnoDB**, tienen la propiedad que son conocidas por el acrónimo **ACID (Atomicity, Consistency, Isolation, Durability)**.

Unidad lógica de procesamiento que busca preservar la integridad y consistencia de los datos.

Por defecto, MySQL utiliza autocommit, generando un *commit* automático por cada sentencia SQL.

MySQL provee sentencias de control de transacciones:

- Se inicia una transacción mediante `START TRANSACTION`.
- Realizar una operación correspondiente al Lenguaje de Definición de Información (DDL) como CREATE, ALTER, DROP, TRUNCATE.
- Luego de finalizada la operación se realiza COMMIT o ROLLBACK.

COMMIT

Una sentencia SQL que termina una transacción, crea cambios permanentes para dicha transacción. Esto es el opuesto a rollback la cual deshace los cambios realizados en la transacción.

InnoDB utiliza un mecanismo optimizado para *commits*, para que los cambios sean escritos en los fichero de la data antes que los *commits* actuales ocurran, esto hace que se realicen más rápido, en consecuencia, requiere de más trabajo en caso de un rollback.

ROLLBACK

> Rollback - doc

Deshace cualquier cambio realizado por una transacción, es el opuesto a commit.

Se pueden realizar los *rollback* que se quieran, pero no surtirán efectos porque al primer *rollback* de una operación de transacción iniciada se revierten los cambios, luego no hace nada.

Las operaciones DDL no pueden ser revertidas.

Modo Autocommit

autocommit - doc

Es una opción que causa que una operación se realice después de cada sentencia SQL. No se recomienda en tablas **InnoDB** con transacciones que tienen varias operaciones.

Realizando operaciones **read-only transactions** en tablas **InnoDB**, minimizando la sobrecarga de bloqueo y generación de información *undo*, especialmente en MySQL 5.6.4+.

Este modo es apropiado para tablas **MyISAM**, donde las transacciones no son aplicables.

Habilitar o Deshabilitar AutoCommit

```
/*  Habilita autocommit  */
SET autocommit = 1;
SET autocommit = ON;

/*  Deshabilita autocommit  */
SET autocommit = 0;
SET autocommit = OFF;
```

Uso de Transaction

Tiene una sintaxis:

```
START TRANSACTION;

/*  Realizar la operaciones DML que se requiera  */
[OPERACION  --    DML]


/*  Realizar los cambios  */
COMMIT;


/*  En caso de error, revertir los cambios  */
ROLLBACK;
```

SQL Avanzado

Palabras claves:

- UNION : selecciona unos valores, combina los resultados de dos consultas juntas.
- UNION ALL : selecciona todos los valores, combina los resultados de dos consultas juntas.
- INTERSECT : opera en dos instrucciones SQL, actúa como operador `AND`, es decir si el valor existe entre ambas instrucciones.
- MINUS : toma todos los resultados de la primera instrucción SQL y luego sustrae aquellos que se encuentran presentes en la segunda instrucción SQL.
- EXISTS : verifica si la consulta interna arroja alguna fila.
- CASE : se utiliza para brindar un tipo de lógica "si entonces otro".
- Consultas Anidadas : instrucciones SQL dentro de otra, al usar `WHERE` o `HAVING`.

UNION

Permite unir 2 o más tablas, implícitamente ejecuta `DISTINCT`

```
[Instrucción SQL 1] UNION [Instrucción SQL 2];
```

Ejemplo

```
SELECT Txn_Date FROM Store_Information UNION SELECT Txn_Date FROM Internet_Sales;
```

UNION ALL

Permite unir 2 o más tablas, no ejecuta implícitamente `DISTINCT`

```
[Instrucción SQL 1] UNION ALL [Instrucción SQL 2];
```

Ejemplo

```
(SELECT Txn_Date FROM Store_Information) UNION ALL (SELECT Txn_Date FROM Internet_Sales);
```

INTERSECT

```
[Instrucción SQL 1] INTERSECT [Instrucción SQL 2];
```

Ejemplo

```
SELECT Txn_Date FROM Store_Information INTERSECT SELECT Txn_Date FROM Internet_Sales;
```

MINUS

```
[Instrucción SQL 1] MINUS [Instrucción SQL 2];
```

Ejemplo

```
SELECT Txn_Date FROM Store_Information MINUS SELECT Txn_Date FROM Internet_Sales;
```

EXISTS

```
SELECT "nombre1_columna" FROM "nombre1_tabla" WHERE EXISTS (
  SELECT * FROM "nombre2_tabla" WHERE "Condición"
);
```

Ejemplo

```
SELECT SUM(Sales) FROM Store_Information WHERE EXISTS (
  SELECT * FROM Geography WHERE region_name = 'West'
);
```

Triggers

> Trigger uso - doc

Los *Triggers* son un programa almacenado invocado automáticamente en respuesta a un evento como INSERT, UPDATE, DELETE que ocurre en una tabla asociada.

En cada *Trigger* puede haber 1 o más comandos dentro de este.

- **No se puede ejecutar sentencias SELECT desde un trigger.**
- **Tampoco se puede llamar SELECT dentro de un Stored Procedures que es ejecutado por un trigger.**

Los *triggers* se puede asignar tareas programadas, pero no son la forma más óptima para manipular datos, acá entra el uso de Stored Procedures;

Ventajas

- Proveen otra forma de comprobar la integridad de la data.
- Manejan errores desde la capa de base de datos.
- Entregan una forma alternativa para *ejecutar tareas programadas*. No se tiene que esperar por un *evento programado* para ser ejecutado porque los *triggers* lo invocan automáticamente *antes* y *después* que los cambios se han realizado en la tabla.
- Se utilizan para auditar cambios en las tablas.

Desventajas

- Solo proveen validaciones extendidas, no todas las validaciones. Para validaciones simples se pueden usar NOT NULL, UNIQUE, CHECK, y FOREIGN KEY.
- Son difíciles de depurar porque se ejecutan automáticamente en la base de datos, no son visibles para aplicaciones clientes.
- Pueden incrementar la carga al servidor MySQL.

Limitaciones

- *trigger* no pueden usar sentencia `CALL` para invocar *stored procedures* que retornan información al cliente o que usen SQL dinámica.
- *trigger* no pueden usar sentencias que explícita o implícitamente comience o termine con una transacción, como START TRANSACTION, COMMIT, ROLLBACK.

Variable `NEW`

`NEW` almacena el valor que aporta la consulta a la base de datos y se puede acceder a los valores de los campos utilizando el nombre del campo.

INSERT, solamente se permite el uso de `NEW`. **UPDATE**, apunta al nuevo valor con que se actualizará.

Por ejemplo: `NEW.NUMERO` corresponde al campo *NUMERO* cuyo valor es del insertado.

Variable `OLD`

`OLD` almacena el valor de las columnas que serán actualizadas o borradas.

UPDATE, corresponde al viejo valor que será reemplazado. **DELETE**, corresponde al valor que será eliminado.

Por ejemplo: `OLD.NUMERO` corresponde al valor del campo *NUMERO* cuyo valor será actualizado o borrado.

Sintaxis TRIGGER

Al crear un *trigger* necesitamos: * Declarar un `DELIMITER //`, cualquier par de carácteres para iniciar las líneas del trigger. * Nombrar el *trigger*. * Establecer cuando se ejecutará el *trigger*, `BEFORE` o `AFTER`. * La operación que se realizará `INSERT`, `UPDATE`, o `DELETE`. * En qué tabla se activará ON y el nombre la tabla. * `FOR EACH ROW` * Uso de `BEGIN`. * Las instrucciones *SQL* que se ejecutarán. * Cierre con `END;` y el delimitador usado anteriormente. * Terminando el `DELIMITER //`, para el funcionamiento normal de las sentencias.

```
DELIMITER //
CREATE TRIGGER `trigger_name`
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON
tablaVIGILADA
FOR EACH ROW
BEGIN
    /*    sentencias SQL que se ejecutarán al    */
    /*    momento de realizar la operación       */
END//

DELIMITER ;
```

TRIGGER INSERT

Este tipo de *trigger* se ejecutará cada vez que se inserte una nueva fila en una tabla que se esté “vigilando”.

La tabla que se escribirán los datos dentro del *trigger* debe existir previamente.

- Después de insertar un valor en la tabla "tabla_producto", se insertará en la tabla *producto_historico* los valores que se declararon en el `INSERT` de los campos *descripcion*, *precio* y la fecha actual.

```
DELIMITER $$

CREATE TRIGGER `trigger_producto_historico`
AFTER INSERT ON tabla_producto
FOR EACH ROW
BEGIN
   INSERT INTO producto_historico(id, descripcion, precio, fecha)
   VALUES (NEW.id, NEW.descripcion, NEW.precio, CURDATE());

END $$

DELIMITER ;
```

- Otro ejemplo, comprobar si la edad de los clientes ingresados es menor a 0, si es así, se establece la edad en 0. Con esto se evita ingresar edades negativas.

```
DELIMITER $$

CREATE TRIGGER `comprobar_edad`
AFTER INSERT ON tabla_cliente
FOR EACH ROW
BEGIN
    IF NEW.edad<0
      THEN SET NEW.edad = 0;
    END IF;

END$$

DELIMITER ;
```

- Otro ejemplo, actualiza las edades de los clientes al insertar una fila a la tabla.

```
DELIMITER //

CREATE TRIGGER `TG_EDAD_CLIENTES_INSERT`
BEFORE INSERT ON tb_clientes
FOR EACH ROW BEGIN
SET NEW.EDAD = timestampdiff(YEAR, NEW.FECHA_NACIMIENTO, NOW());
END //

DELIMITER ;
```

TRIGGER UPDATE Se ejecutará cada vez que se actualice un registro de la tabla.

```
DELIMITER //
CREATE TRIGGER `nombre_trigger`
{BEFORE | AFTER} UPDATE
ON nombre_tabla
```

```
FOR EACH ROW
BEGIN
    /*   UPDATE SENTENCE   */
    /*   uso de OLD y NEW   */
END//
```

TRIGGER DELETE Se ejecutará cada vez que se elimine un registro de la tabla.

```
DELIMITER //
CREATE TRIGGER `nombre_trigger`
{BEFORE | AFTER} DELETE
ON nombre_tabla
FOR EACH ROW
BEGIN
    /*   DELETE SENTENCE   */
    /*   uso de OLD   */
END//
```

SHOW TRIGGERS

Listar todos los *triggers* en una base de datos, pudiendo usar filtros para obtener los requeridos.

- SHOW TRIGGERS

```
SHOW TRIGGERS [{FROM | IN} db_name] [LIKE '%patron%' | WHERE expr];
```

- Consultar a INFORMATION_SCHEMA.TRIGGERS

```
SELECT * FROM INFORMATION_SCHEMA.TRIGGERS
WHERE TRIGGER_SCHEMA='test' AND TRIGGER_NAME='ins_sum';
```

DROP TRIGGER

```
DROP TRIGGER [IF EXISTS] nombre_trigger;
```

Stored Procedures

Son códigos SQL que se pueden almacenar y ser reusado, se ejecutan al ser llamados. Se les puede pasar parámetros.

El nombre tiene un máximo de 64 carácteres, símbolos $ y _, alfanuméricos, y es case sensitive. El nombre debe ser único.

- Sin Parámetros

```
DELIMITER $$
CREATE [DEFINER = user] PROCEDURE [IF NOT EXISTS]
nombre_procedimiento ()
BEGIN
DECLARE <declaración de variables>;
...
<ejecuciones del procedimiento>;
...
END$$

DELIMITER;
```

- Con Parámetros

```
DELIMITER $$
CREATE [DEFINER = user] PROCEDURE [IF NOT EXISTS]
nombre_procedimiento (
    [ IN | OUT | INOUT ] nombre_Parametro1 TIPO_DATO,
    [ IN | OUT | INOUT ] nombre_ParametroX TIPO_DATO,
)
BEGIN
DECLARE <declaración de variables>;
...
<ejecuciones del procedimiento>;
...
END$$

DELIMITER;
```

- Al crear un *Stored Procedure* este puede o no llevar parámetros.
- **DEFINER** : asigna el usuario que ejecutará el *stored procedure*, cualquier usuario puede ejecutar dicho procedimiento almacenado y se ejecutará como si fuera el usuario *dueño* este, y con ello, con los permisos que tiene el usuario asignado, (algo así como un SUID, el que ejecuta el fichero tendrá los mismos permisos que el dueño del fichero).

Manipulación de datos con Stored Procedure

Podemos manipular tablas existentes con *Stored Procedure*, insertar, eliminar, consultar, actualizar datos, utilizar variables y/o parámetros para utilizar en el cuerpo del *stored procedure*.

```
/*   Stored Procedure actualiza la edad de todos los clientes   */
DELIMITER $$

CREATE PROCEDURE `calcula_edad` ()
BEGIN
UPDATE tb_cliente SET EDAD = TIMESTAMPDIFF(YEAR, FECHA_NACIMIENTO, CURDATE());
END$$

DELIMITER ;
```

```
/*  Llama al SP   */
CALL calcula_edad();
```

Parámetros en Stored Procedures

Las variables definidas (`OUT`, `INOUT`)por el usuarios en una sentencia `CALL` que invoca el procedimiento, se pueden recuperar cuando el procedimiento los retorna.

Si no se entrega un parámetro que requiere el *stored procedure* generará un error.

Si se está llamando un *stored procedure* dentro de un TRIGGERS, se puede usar la variable `NEW` (de los triggers) como un parámetro de tipo `OUT` o `INOUT` de un *stored procedure* que lo requiera.

Existen 3 tipos de parámetros:

1. `IN`
2. `OUT`
3. `INOUT`

Parámetro	Descripción	Uso
`IN`	Por defecto, este parámetro se pasará dentro del procedimiento. Este parámetro puede ser modificado por el procedimiento, pero no es visible cuando se retorna el valor.	`IN texto VARCHAR(100)`
`OUT`	Pasa el valor del parámetro desde el procedimiento hacia fuera (un *return* del valor). Valor inicial es `NULL` dentro del procedimiento, es visible cuando se retorna el valor.	`OUT texto VARCHAR(100)`
`INOUT`	Inicializado cuando se llama, puede ser modificado por el procedimiento y cualquier cambio hecho es visible cuando se retorna.	`INOUT texto VARCHAR(100)`

Parámetro IN

Ejemplo `IN`, encuentra todas las oficinas en un país especificado por el parámetro *countryName*.

```
DELIMITER //

CREATE PROCEDURE `GetOfficeByCountry` (
    IN countryName VARCHAR(255)
)
BEGIN
    SELECT *
    FROM offices
    WHERE country = countryName;
END //
```

```
DELIMITER ;
```

```
CALL GetOfficeByCountry('USA');
```

Parámetro OUT

Ejemplo `OUT`, retorna un número de órdenes por el estado de orden.

```
DELIMITER $$

CREATE PROCEDURE `veces_comprador` (
    IN nombreCliente VARCHAR(30),
    OUT totalCliente INT
)
BEGIN
    SET totalCliente = (
        SELECT COUNT(A.NOMBRE) FROM tb_cliente AS A
        INNER JOIN tb_factura AS B
        WHERE A.DNI = B.DNI AND
        A.NOMBRE LIKE CONCAT('%', nombreCliente, '%')
    );
END$$

DELIMITER ;
```

Para llamar el parámetro `OUT` se debe utilizar el mismo nombre del parámetro creado en el *stored procedure*.

```
CALL veces_comprador('edson',@totalCliente);
SELECT @totalCliente;
```

Parámetro INOUT

Ejemplo, se crea un contador, con parámetro `INOUT` y un incremental de tipo `IN`.

```
DELIMITER $$

CREATE PROCEDURE `SetCounter` (
    INOUT counter INT,
    IN incremental INT
)
BEGIN
    SET counter = counter + incremental;
END$$

DELIMITER ;
```

```
SET @counter = 1;
CALL SetCounter(@counter,1); -- 2
CALL SetCounter(@counter,1); -- 3
CALL SetCounter(@counter,5); -- 8
SELECT @counter; -- 8
```

Variables en Stored Procedures

Se puede establecer variables dentro de la declaración del *Stored Procedure (SP)*, después de `BEGIN`. Variables del mismo tipo, pueden ir en la misma línea separados por coma.

```
DECLARE nombre_variable TIPO_DATO [DEFAULT valor];
```

Ejemplo DECLARE

```
DELIMITER $$
CREATE PROCEDURE `desafio_1` ()
BEGIN
DECLARE Nombre, Apellido VARCHAR(10);
DECLARE Edad INT;
DECLARE Fecha_Nacimiento DATE;
DECLARE Costo FLOAT;

SET Nombre = 'Juan';
SET Apellido = 'Soto';
SET Edad = 10;
SET Fecha_Nacimiento = '20170110';
SET Costo = 10.23;

SELECT Nombre;
SELECT Apellido;
SELECT Edad;
SELECT Fecha_Nacimiento;
SELECT Costo;

END$$

DELIMITER ;
```

En este *SP* las variables se pueden ver al momento de ser ejecutado el stored procedure.

DECLARE ... HANDLER

> Error-handling - doc DECLARE … HANDLER - doc

Especifica cómo lidiar con una o más condiciones, si estas ocurren, se ejecuta la sentencia.

Línea `DECLARE ... HANDLER FOR ...` NO TERMINA CON PUNTO Y COMA ;.

Las sentencias son solamente `SET variable = valor`, dentro de un `BEGIN` y `END;`.

```
DECLARE variableUtilidad TIPO_CAMPO;

DECLARE handlerAction HANDLER FOR condicionValue
BEGIN
    -- cuerpo del handler
END;
```

- handlerAction
  - `CONTINUE` : ejecución del programa continúa.
  - `EXIT` : ejecución termina para la sentencia `BEGIN` y `END` en la que fue declarada.
- condicionValue : indica la condición o clase de condición que activa el handler, o el valor que tiene un error MySQL.

Manejo de errores

El siguiente ejemplo, maneja errores para el código *1051* de MySQL, indica que la tabla es desconocida.

```
DECLARE mensaje VARCHAR(50);
DECLARE EXIT HANDLER FOR 1051
BEGIN
    SET mensaje = "Tabla Ingresada Desconocida";
    SELECT mensaje;
END;
```

El mensaje se declara mediante `DECLARE` y el tipo de datos. Al generarse el error, se muestra el mensaje, en lugar del error estándar de MySQL, generando una salida más amigable. Se establece un valor para *mensaje* y se muestra mediante SELECT.

Recordar que siempre se debe declarar la variable que se utilizará dentro del *error handler* o en el cuerpo del *stored procedure*.

Ejemplo Completo DECLARE HANDLER

```
DELIMITER $$
CREATE PROCEDURE `incluir_producto_parametros` (
    vcodigo VARCHAR(20), vnombre VARCHAR(20),
    vsabor VARCHAR(20), vtamano VARCHAR(20),
    venvase VARCHAR(20),  vprecio DECIMAL(4,2)
)
BEGIN
    DECLARE mensaje VARCHAR(40);
    DECLARE EXIT HANDLER FOR 1062
    BEGIN
        SET mensaje = 'Producto duplicado! ';
        SELECT mensaje;
    END;
    INSERT INTO tb_productos (
        CODIGO,
        DESCRIPCION,
        SABOR,TAMANO,ENVASE,PRECIO_LISTA
    )
    VALUES (
        vcodigo, vnombre, vsabor, vtamano, venvase, vprecio
    );
    SET mensaje = 'Producto incluido con éxito!';
    SELECT mensaje;
END $$

DELIMITER ;
```

```
CALL incluir_producto_parametros(
    '1000801','Sabor del Mar', 'Naranja', '700 ml', 'Botella de Vidrio', 3.25
);
```

Si el producto no existe mostrará el mensaje de éxito, si ya existe mostrará el mensaje de duplicado.

Usar un stored procedure

Para llamar un *stored procedure* se debe utilizar CALL.

```
CALL nombre_stored_procedure();

CALL nombre_StoredProcedure(parametro1, parametoX);
```

- `CALL` : invoca un procedimiento almacenado que se haya definido. Stored procedure que no requieren argumentos se puede llamar sin paréntesis.

Modificar Stored Pprocedure

> Modificar Stored Procedure - doc

No se puede alterar un *Stored Procedure* como tal, se debe eliminar y crear un nuevo procedimiento almacenado.

Mediante el uso de `ALTER PROCEDURE nombre_procedure` se puede alterar carácterísticas del procedimiento almacenado, pero *NO el cuerpo y parámetros que utiliza.*

En *WorkBench* se puede alterar haciendo segundo clic en la entrada *Stored Procedure* a modificar, y seleccionar *Alter Store Procedure….*

```
DROP PROCEDURE nombre_procedure;

CREATE PROCEDURE ...
```

DROP Stored Procedure

```
DROP PROCEDURE [IF EXISTS] nombre_procedure;
```

Listar Stored Procedure

Muestra los procedimientos almacenados, mostrando los procedimientos que tienes acceso.

```
SHOW PROCEDURE STATUS [LIKE '%patron%' | WHERE filtro_sentencia];
```

Muestra todos los procedimientos almacenados en el servidor actual MySQL.

```
SHOW PROCEDURE STATUS;
```

- Utilizando **information_schema**.

```
SELECT
    routine_name
FROM
    information_schema.routines
WHERE
    routine_type = 'PROCEDURE'
        AND routine_schema = '<database_name>';
```

SHOW CREATE PROCEDURE

show create procedure - stackoverflow

```
SHOW CREATE PROCEDURE stored_procedure_name;
```

Control de Flujo

Flow Control - doc

MySQL soporta:

- IF
- CASE
- ITERATE
- LEAVE
- LOOP
- REPEAT
- RETURN
- WHILE

IF ELSEIF ELSE

IF - doc

Evalúa una condición y ejecuta la sentencia correspondiente.

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list]
    ...
    [ELSE statement_list]
END IF
```

Ejemplo IF

Usado dentro de un *Stored Procedure*.

```
DELIMITER //

CREATE FUNCTION `VerboseCompare` (n INT, m INT)
  RETURNS VARCHAR(50)

  BEGIN
    DECLARE s VARCHAR(50);

    IF n = m THEN SET s = 'equals';
    ELSE
      IF n > m THEN SET s = 'greater';
      ELSE SET s = 'less';
      END IF;

      SET s = CONCAT('is ', s, ' than');
    END IF;

    SET s = CONCAT(n, ' ', s, ' ', m, '.');

    RETURN s;
```

```
  END //

DELIMITER ;
```

CASE

CASE - doc

CASE permite implementar condicionales complejas.

```
CASE [case_value]
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list]
    ...
    [ELSE statement_list]
END CASE;
```

[case_value] es opcional.

CASE NOT FOUND

En caso de algún error, se debe utilizar un DECLARE … HANDLER, declarando una variable, usar par `BEGIN` y `END` para el **error handler** y mostrar un mensaje de error más amistoso.

```
DECLARE mensaje VARCHAR(50);

DECLARE CONTINUE HANDLER codigoErrorMySQL
BEGIN
    SET mensaje = "Un error ha ocurrido";
    SELECT mensaje;
END;


CASE [case_value]
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list]
    ...
    [ELSE statement_list]
END CASE;
```

Ejemplo en Query

```
SELECT NOMBRE,
CASE
    WHEN YEAR(fecha_de_nacimiento) < 1990 THEN 'Viejos'
    WHEN YEAR(fecha_de_nacimiento) >= 1990
    AND YEAR(fecha_de_nacimiento) <= 1995 THEN 'Jóvenes'
    ELSE 'Niños'
END AS CLASIFICACION_EDAD
FROM tabla_de_clientes;
```

Ejemplo CASE Stored Procedure

```
DELIMITER $$

CREATE PROCEDURE `p` ()
BEGIN
    DECLARE v INT DEFAULT 1;

    CASE v
        WHEN 2 THEN SELECT v;
        WHEN 3 THEN SELECT 0;
    ELSE
        BEGIN
        END;
    END CASE;

END $$

DELIMITER ;
```

ITERATE

ITERATE - doc

Sentencia aparece dentro de sentencias LOOP, REPEAT, WHILE.

Indica que el loop se inicie nuevamente.

```
ITERATE label
```

LEAVE

LEAVE - doc

Es usado para salir del control de flujo que ha recibido la etiqueta.

Usado dentro de `BEGIN` y `END` o LOOP, REPEAT, WHILE.

```
LEAVE label
```

LOOP

LOOP

Implementa un constructor simple loop, permitiendo ejecución de lista de sentencias con uno o más elementos, cada una terminada con punto y coma `;`.

Algo así como un blucle *FOR*, es usualmente usado con LEAVE, RETURN.

```
[begin_label:] LOOP
    statement_list
END LOOP [end_label]
```

Ejemplo LOOP

```
CREATE PROCEDURE `doiterate` (p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN
      ITERATE label1;
    END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;
END;
```

REPEAT

REPEAT - doc

Repite hasta que la condición sea verdadera, cada sentencia termina con punto y coma ;.

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition
END REPEAT [end_label]
```

Ejemplo REPEAT

```
DELIMITER //

CREATE PROCEDURE `dorepeat` (p1 INT)
       BEGIN
         SET @x = 0;
         REPEAT
           SET @x = @x + 1;
         UNTIL @x > p1 END REPEAT;
       END
       //

DELIMITER ;
```

```
mysql> CALL dorepeat(1000);
mysql> SELECT @x;
+------+
| @x   |
```

```
+------+
| 1001 |
+------+
1 row in set (0.00 sec)
```

RETURN

RETURN - doc

Termina la ejecución y retorna un valor.

NO es usado en Stored Procedure, TRIGGERS, o eventos.

```
RETURN expresion
```

WHILE

WHILE - doc

Repite tantas veces siempre y cuando la condición sea verdadera.

Múltiples sentencias SQL deben terminar con punto y coma ;.

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label];
```

Ejemplo WHILE

```
CREATE PROCEDURE `dowhile` ()
BEGIN
  DECLARE v1 INT DEFAULT 5;

  WHILE v1 > 0 DO
    ...
    SET v1 = v1 - 1;
  END WHILE;
END;
```

FUNCTION

Es un Programa almacenado que siempre retornan valor.

La diferencia entre una función y una subrutina es que una función ejecuta una serie de comandos y devuelve un resultado; una subrutina ejecuta una serie de comandos pero no necesariamente devuelve un resultado.

Un Stored Procedure utiliza el comando `CALL` y una función utiliza el comando SELECT para su ejecución, respectivamente.

Se puede usar como una función normal en SELECT, como un filtro en WHERE.

Los parámetros se nombran y `RETURNS` se declaran el tipo de los datos en el orden en que los parámetros son declarados.

```
DELIMITER $$

CREATE [DEFINER = user] FUNCTION [IF NOT EXISTS] nombre_funcion (
    parametro1,
    parametro2,
    parametroX,
)
RETURNS TIPO_DATO
[DETERMINISTIC | NOT DETERMINISTIC]
BEGIN
    DECLARE variableRetorno TIPO_DATO;

    [cuerpo funcion ]

    RETURN variableRetorno;
END$$

DELIMITER ;
```

- En casi todas las funciones se utiliza `DETERMINISTIC`, por lo que no dará error MySQL por defecto.
- Dentro del cuerpo de la función se debe establecer los datos de la o las variables que se retornarán mediante `SET`. `RETURN` indica las variables que se retornarán.

Para llamar la función se debe utilizar `SELECT`.

```
SELECT nombre_funcion(parametro);
```

Para utilizar en una query se debe usar como cualquier otra función MySQL.

```
SELECT nombre, sabor, nombre_funcion(sabor) AS TIPO FROM tabla_jugos;
```

---

log_bin_trust_function_creators - doc

MySQL no permite que el usuario normal cree funciones por defecto.

Se debe cambiar la configuración global:

```
SET GLOBAL log_bin_trust_function_creators = 0;
```

Utilizando DETERMINISTIC se puede crear funciones estando activada dicha opción.

Ejemplo FUNCTION

La siguiente función, categoriza por el tipo de sabor y retorna una variables con un tipo de dato *VARCHAR*.

```
DELIMITER $$

CREATE FUNCTION `function_define_sabor` (
    varSabor VARCHAR(40)
)
RETURNS VARCHAR(40) DEFAULT ""
DETERMINISTIC
CASE varSabor
    WHEN "Uva" THEN   SET varSabor = 'Normal';
    WHEN "Piña" THEN   SET varSabor = 'Dulce';
    WHEN "Limón" THEN   SET varSabor = 'Cítrico';
    ELSE SET varSabor = 'Jugo Común';
END CASE;

RETURN varSABOR;
END$$

DELIMITER;
```

```
SELECT NOMBRE, SABOR, function_define_sabor(SABOR) FROM tb_productos;
```

```
SELECT NOMBRE, SABOR FROM tb_productos WHERE  function_define_sabor(SABOR) = 'Normal';
```

Función que cuenta la cantidad de facturas por año.

```
DELIMITER $$

CREATE FUNCTION `f_numero_facturas`(fecha DATE)
RETURNS INTEGER
DETERMINISTIC
BEGIN
DECLARE n_facturas INT;
SELECT COUNT(*) INTO n_facturas FROM tb_factura WHERE FECHA_VENTA = fecha;
RETURN n_facturas;
END $$

DELIMITER ;
```

```sql
SELECT f_numero_facturas('20160201') AS RESULTADO;
```

DROP FUNCTION

```sql
DROP FUNCTION [IF EXISTS] nombre_funcion;
```

Listar funciones

Permite mostrar todas las funciones y/o filtrar para obtener funciones determinadas.

```sql
SHOW FUNCTION STATUS [LIKE 'patron' | WHERE condición];
```

Consultas Anidadas

Los resultados de una consulta se pueden utilizar en otras consultas. Pudiendo ser operadas mediante el uso de operadores de comparación, sentencias como IN, ON, WHERE, etc.

```sql
SELECT "nombre1_columna" FROM "nombre1_tabla" WHERE "nombre2_columna"
[Operador]
(
  SELECT "nombre3_columna"
  FROM "nombre2_tabla"
  WHERE "Condición"
);
```

Operador de Comparación	Descripción
>	gt
>=	gte
<	lt
<=	lte
=	eq
<>, !=	not equal

Fuera de paréntesis son consultas externas, dentro del paréntesis son consultas internas.

Ejemplo

```sql
SELECT SUM(Sales) FROM Store_Information
WHERE Store_Name IN (
  SELECT Store_Name FROM Geography WHERE Region_Name = 'West'
);
```

Funciones MySQL

Permite realizar cálculos matemáticos, como, mínimo, máximo, contar, promedio, entre otras características de MySQL y SQL.

Sintaxis

```
SELECT nombre_función('columna_nombre') FROM nombre_tabla;
```

AVG

```
SELECT AVG(age) FROM personal;
```

COUNT

```
SELECT COUNT(name) FROM personal;

SELECT COUNT(DISTINCT name) FROM personal;
```

FIRST

Devuelve el primer valor de la columna seleccionada.

Sintaxis

```
SELECT FIRST(precio) FROM pedidos;
```

LAST

Devuelve el último valor de la columna seleccionada.

Sintaxis

```
SELECT LAST(precio) FROM pedidos;
```

MAX

Retorna el número mayor de la columna seleccionada.

Sintaxis

```
SELECT MAX(precio) FROM pedidos;
```

MIN

Retorna el número menor de la columna seleccionada.

Sintaxis

```
SELECT MIN(precio) FROM pedidos;
```

SUM

Retorna una suma de los valores de la columna seleccionada, deben ser de tipo número.

Sintaxis

```
SELECT sum(precio) FROM pedidos;
```

UCASE

Convierte string en mayúscula, campo tipo string.

Sinónimo de función **UPPER()**

Sintaxis

```
SELECT UCASE(nombre) FROM pedidos;
```

LCASE

Convierte string en minúscula, dato de tipo string.

Sinónimo de función **LOWER()**

Sintaxis

```sql
SELECT LCASE(nombre) FROM pedidos;
```

MID

Extrae carácteres de un campo de texto, es decir, corta el string. Recibe tres parámetros: el campo, INT_inicio, INT_final.

Sintaxis

```sql
SELECT MID(cliente,1,3) FROM pedidos;
```

LENGTH

Calcula la longitud del string del campo seleccionado, tipo texto.

Sintaxis

```sql
SELECT LENGTH(cliente) AS cliente FROM pedidos;
```

FORMAT

Usado para dar formato en el campo seleccionado. Se le entrega dos parámetros: columna, formato.

Sintaxis

```sql
SELECT producto, precio, FORMAT(NOW(), 'YYYY-MM-DD') AS fecha FROM productos;
```

DIV

Se utiliza para la división de enteros (x se divide por y). Se devuelve un valor entero.

Sintaxis

```sql
SELECT 8 DIV 3;
```

CEIL

Retorna el valor entero más pequeño que es mayor o igual que un número.

Sintaxis

```
SELECT CEIL(numero);
```

CEILING

La función devuelve el valor entero más pequeño que es mayor o igual que un número. Nota: Esta función es igual a la función CEIL().

Sintaxis

```
SELECT CEILING(numero);
```

FLOOR

Retorna el valor entero más grande que será igual o menor que el de un número de entrada dado.

Sintaxis

```
SELECT FLOOR(numero_float);
```

RAND()

Retorna un número float aleatorio entre 0 y 1 (no se incluye).

```
SELECT RAND();
```

- Generar un número entre un rango de números.

```
RAND() * (N_MAX -  N_MIN + 1) + N_MIN
```

Ejemplo

```
SELECT RAND() * (10 - 5 + 1) + 5;
```

ROUND()

Función que redondea un número especificando el número y la cantidad de decimales.

```sql
SELECT ROUND(numero, decimales);
```

CONVERT()

Convierte el tipo de dato a otro.

```sql
SELECT CONVERT(dato_original,  tipo_dato_final);
```

- dato_original

Parameter	Description
value	Required. The value to convert

- tipo_dato_final

type	Required. The datatype to convert to
DATE	Converts value to DATE. Format: "YYYY-MM-DD"
DATETIME	Converts value to DATETIME. Format: "YYYY-MM-DD HH:MM:SS"
DECIMAL	Converts value to DECIMAL. Use the optional M and D parameters to specify the maximum number of digits (M) and the number of digits following the decimal point (D).
TIME	Converts value to TIME. Format: "HH:MM:SS"
CHAR	Converts value to CHAR (a fixed length string)
NCHAR	Converts value to NCHAR (like CHAR, but produces a string with the national character set)
SIGNED	Converts value to SIGNED (a signed 64-bit integer)
UNSIGNED	Converts value to UNSIGNED (an unsigned 64-bit integer)
BINARY	Converts value to BINARY (a binary string)

charset | Required. The character set to convert to |

- Ejemplo

Convierte `VARCHAR` a `INT`.

```sql
SELECT CONVERT('12', UNSIGNED INT);
```

CURDATE()

Retorna la fecha actual.

```
SELECT CURDATE();
```

CURRENT_TIMESTAMP()

Retorna la fecha y hora actuales.

```
SELECT CURRENT_TIMESTAMP();
```

YEAR()

Retorna la parte del año de la fecha entregada.

```
SELECT YEAR(date);
```

MONTH()

Retorna la parte del mes de la fecha entregada.

```
SELECT MONTH(date);
```

DAY()

Retorna la parte del mes de la fecha entregada.

```
SELECT DAY(date);
```

MONTHNAME()

Retorna el nombre del mes de la fecha.

```
SELECT MONTHNAME(date);
```

DAYNAME()

```
SELECT DAYNAME(date);
```

DATEDIFF

Retorna el número de días entre dos fechas.

```
SELECT DATEDIFF
```

DATE_SUB

Resta tiempo o fecha entre intervalo de fecha y retorna la fecha resultante.

```
SELECT DATE_SUB(fecha, INTERVAL valor TipoIntervalo);
```

TipoIntervalo:

- MICROSECOND
- SECOND
- MINUTE
- HOUR
- DAY
- WEEK
- MONTH
- QUARTER
- YEAR
- SECOND_MICROSECOND
- MINUTE_MICROSECOND
- MINUTE_SECOND
- HOUR_MICROSECOND
- HOUR_SECOND
- HOUR_MINUTE
- DAY_MICROSECOND
- DAY_SECOND
- DAY_MINUTE
- DAY_HOUR
- YEAR_MONTH

DATE_FORMAT

Da formato a una fecha especificada, el formato va entre comillas.

```
SELECT DATE_FORMAT(fecha, "FORMATO");
```

FORMATO	Descripción
%a	Abbreviated weekday name (Sun to Sat)
%b	Abbreviated month name (Jan to Dec)
%c	Numeric month name (0 to 12)
%D	Day of the month as a numeric value, followed by suffix (1st, 2nd, 3rd, …)
%d	Day of the month as a numeric value (01 to 31)
%e	Day of the month as a numeric value (0 to 31)
%f	Microseconds (000000 to 999999)
%H	Hour (00 to 23)
%h	Hour (00 to 12)

FORMATO	Descripción
%I	Hour (00 to 12)
%i	Minutes (00 to 59)
%j	Day of the year (001 to 366)
%k	Hour (0 to 23)
%l	Hour (1 to 12)
%M	Month name in full (January to December)
%m	Month name as a numeric value (00 to 12)
%p	AM or PM
%r	Time in 12 hour AM or PM format (hh:mm:ss AM/PM)
%S	Seconds (00 to 59)
%s	Seconds (00 to 59)
%T	Time in 24 hour format (hh:mm:ss)
%U	Week where Sunday is the first day of the week (00 to 53)
%u	Week where Monday is the first day of the week (00 to 53)
%V	Week where Sunday is the first day of the week (01 to 53). Used with %X
%v	Week where Monday is the first day of the week (01 to 53). Used with %x
%W	Weekday name in full (Sunday to Saturday)
%w	Day of the week where Sunday=0 and Saturday=6
%X	Year for the week where Sunday is the first day of the week. Used with %V
%x	Year for the week where Monday is the first day of the week. Used with %v
%Y	Year as a numeric, 4-digit value
%y	Year as a numeric, 2-digit value

SUBSTRING

Extrae una parte de un string.

```
SELECT SUBSTRING(string, start, length);
```

DBA

Profesional responsable de administrar la base de datos.

Funciones DBA

- Evalúa el ambiente Hardware necesario, mantiene MySQL de acuerdo a las necesidades de la empresa.
- Configura acceso de forma segura (conexiones, ide, otras interfaces).
- Mantiene un buen desempeño de la base de datos. Trabaja con índices que permiten mejorar las consultas a la base de datos.
- Almacena los datos, realiza respaldos.
- Apoya el área del desarollo, elimina datos no deseados, defragmenta la base de datos, entre otros.
- Monitorea la instalación MySQL, los recursos usados por esta y los adecua según los usuarios.
- Configura ambiente y sus propiedades, fichero **my.ini** (Windows) o **my.conf** (Linux).
- Administra usuarios de la base de datos, las creaciones y permisos.

Conexiones

Las conexiones utilizan el modo *cliente-servidor*.

El puerto por defecto usado por MySQL es `3306`.

En *WorkBench* podemos crear, editar, eliminar conexiones en la sección *MySQL Connections*.

Podemos copiar el fichero de conexiones ubicados, este fichero de configuración es un *XML*:

- Windows : `C:\Usuarios\userName\AppData\Roaming\MySQL\Workbench`
- Linux : `~/.mysql/workbench/`

Para eliminar una conexión se debe ir a configuración, *Delete* y cerrar la ventana.

El cliente se debe conectar mediante un dirección del servidor MySQL.

Dicha dirección utilizada en el comando *mysql* debe ser declarada al momento de crear el usuario para que pueda conectarse.

MySQL soporta el uso de wildcards `%` y `_`.

```
192.168.1.%   -->   192.168.1.0 - 192.168.1.255
192.168.1.1_  -->   192.168.1.100 - 192.168.1.255

client__.company.com   -->   clientXY.company.com
```

Por lo tanto:

- Una conexión hacia una base de datos sería.

```
mysql -h server-mysql.company.com -p 3306 -u unUser -p
```

- de un usuario con una ip específica, GRANT.

```
GRANT SELECT ON db.tabla TO 'usuario'@'192.168.10.10';
```

- usando wildcard, esto indica que acepta conexiones desde cualquier ip del cliente, GRANT.

```
GRANT SELECT ON db.tabla TO 'usuario'@'%';
```

Servicio MySQL

Para manejar el servicio y poder realizar tareas administrativas, se debe interrumpir el servicio MySQL, editar el fichero **my.ini** en Windows y **my.conf** en Linux, etc. Variables de servidor.

- En Windows, Iniciar, detener, reiniciar servicio, ir a **Servicios**.

Detener usando comandos, `net stop mysql80`. Iniciar mediante comandos, `net start mysql80`.

- En Linux, se debe manejar mediante **systemctl**.

```
systemctl start mysql
systemctl stop mysql
systemctl restart mysql
```

Optimization MySQL

> Optimization - doc

Optimizar la base de datos MySQL se puede realizar de 4 formas:

- Esquemas
- Índices
- Variables internas MySQL (`mysqld`)
- Hardware y sistema operativo.

Hardware y sistema operativo

Se debe priorizar sistemas 64-bits, pero es posible encontrar sistemas 32-bits por lo que se debe adecuar el servicio según el hardware presente.

Como la cantidad de RAM es limitada, se puede asignar un límite de consumo del servicio para impedir que la máquina colapse. Es recomendado no exceder el 50% de la RAM disponible.

Por ejemplo, 1 DB de 1GB no consumirá 8 GB de RAM, pero en un ambiente con muchas conexiones ejecutando operaciones, esas 8 GB no serán suficientes.

Disco duro que se esté usando, HDD/SSD. Se puede encontrar discos SCSI, pero son viejos, se debe priorizar discos SATA o SAS y por sobre todo SAS, estos tienen un alto precio, por su rendimiento y confiabilidad es la mejor opción.

RAID **Grupo/matriz redundante de discos independientes** o **RAID** es un sistema de almacenamiento de datos que utiliza múltiples unidades (HDD o SSD), entre las cuales se distribuyen o replican los datos. Combina varios discos duros en una sola unidad lógica.

Existe RAID por hardware y por software, este último es más flexible debido a que es el sistema operativo quién maneja, mediante un software la creación, administra, elimina raid virtuales (en Linux utilizando la herramienta **mdadm**, soporta raid 0, 1, 5, 10).

- Mayor integridad.
- Tolerancia frente a fallos.
- Tasa de transferencia.
- Capacidad.

Es común encontrar RAID 0, 1, 5 y 10, para almacenar las bases de datos y tener respaldos de estas.

- RAID 0 : (conjunto dividido), divide los datos entre dos discos duros distintos, el sistema operativo verá solamente 1 (el primario). Proporciona **redundancia**, alto rendimiento en escritura porque se usa más de un disco en paralelo,
- RAID 1 : (espejo), un disco duro es un espejo del otro, estos tendrán los mismos datos sincronizados. Este sistema es tan grande como el menor de los discos. Se recomiendan usar controladoras de disco independientes, una para cada disco (*splitting* o *duplexing*).
- RAID 5 : (distribuido con paridad), divide los datos en más de dos discos duros, el sistema operativo verá solamente 1. Necesita de un mínimo de 3 discos duros para ser implementado. Bajo costo de *redundancia*
- RAID 01 : (RAID 0+1), es un espejo de divisiones. Replica y comparte los datos entre varios discos, la localización de cada nivel de RAID está dentro del conjunto final. No es ta robusto como RAID 10, no tolera fallos simultáneos dentro del mismo conjunto.
- RAID 10 : (RAID 1+0), los discos duros tienen espejos, pueden fallar todos excepto 1 y los datos estarían a salvo, si no se reemplazan los que han fallado, los datos se perderán.

Los *RAID* más recomendados es **RAID 1** y **RAID 10**, aunque gasten más espacio físico con redundancia son los más usados porque guardan respaldos de los datos almacenados.

Puede hacer

- Mejora *uptime* o tiempo en línea, los raid 1, 0+1 o 10, 5 y 6 permiten que falle un disco y los datos se mantienen accesibles.
- Permite mejorar el rendimiento de algunas aplicaciones, debido al *stripping* (raid 0, 5), varios discos duros atienden simultáneamente operaciones de lectura lineales.

No puede hacer

- Protege los datos, al tener un sistema de ficheros este es vulnerable a fallos ante una variedad de riesgos de fallo físico de disco. No impedirá que los datos se corrompan, se afecten por un virus, modificación o borrado accidental, o que se cree un fallo en otro componente y afecte al sistema.
- No simplifica la recuperación de un desastre. Se necesitan controladores software específicos.
- No mejora el rendimiento en todas las aplicaciones (aplicaciones de escritorios típicas).
- No facilita traslado a un sistema nuevo. La BIOS RAID debe ser capaz de leer los metadatos de los miembros del conjunto para reconocerlos y hacerlo disponible para el sistema operativo.

Variables de ambiente MySQL

> Variable MySQL - doc Variable References - doc

Se declaran fuera del programa, al momento de inicialización del servicio estas variables quedan configuradas de forma predeterminada.

Existen alrededor de 250+ variables de ambiente, esta cantidad varía según la versiones.

Variables de Servidor

> Variables Servidor - doc

Estas variables pueden ser establecidas dentro de una conexión y son temporales, o se establecen en el fichero `my.cnf` (Linux) o `my.ini` (Windows) para que sean permanentes.

Algunas variables importantes:

Variables	Descripción
`datadir`	establece el directorio en donde se almacenarán los ficheros contenedores de información de mysql (db, tablas, indices, etc).
`socket`	Unix, fichero socket que es usado para conexiones locales. Default */tmp/mysql.sock*.
`log-error`	
`pid-file`	ruta del fichero que tiene el process ID del servicio mysql.
`bind_address`	IPv4/IPv6, establece la dirección TCP/IP del servidor, pudiendo establecer múltiples redes separadas por coma (`*`, `0.0.0.0`, `192.168.0.2/24`, `::`) . Por defecto *localhost* o 127.0.0.1.
`port`	establece el puerto a usar, default *3306*
`admin_address`	establece dirección TCP/IP para tareas administrativas. Crea una interfaz administrativo.
`admin_port`	establece puerto para tareas administrativas.

SHOW STATUS

Muestra valores actuales de las variables de ambiente.

```
SHOW [GLOBAL | SESSION] STATUS [LIKE 'pattern'];
```

SHOW VARIABLES

```
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern'];
```

- Muestra el directorio en donde se almacenan la base de datos.

```
SHOW VARIABLES WHERE Variable_Name LIKE '%dir';
```

SET variable

SET - doc

Permite asignar valores a diferentes variables que afectan las operaciones del servidor o clientes. Variables temporales.

- Variables de configuración

```
SET [GLOBAL | SESSION] variable = valor;
```

- Variables comunes

```
SET @nombre_variable = valor;
```

Dentro de *stored procedures* se puede utilizar `SET` sin el uso de *@* para las variables.

Tipo de variables

1. `GLOBAL` : configura variables para todas las conexiones del entorno MySQL, **mysqld**.
2. `SESSION` : configura variables solamente para sesión actual, **client**.

Costo de ejecución

La ausencia de claves primarias y foráneas perjudicará el rendimiento de cada sentencia SQL aplicada a dicha tabla, teniendo un uso mayor del servidor.

Por lo tanto, es recomendado crear tablas con claves primarias y/o claves foráneas.

El costo no está expresado en ninguna unidad. Tan solo usamos este valor para compararlo con otros planes de ejecución. Cuanto menor sea el valor, más rápida va a ser la consulta.

En *Workbench*, ir a la tabla, en un nuevo script de queries, buscar *Execution Plan*, acá mostrará la información de forma visual y más amigable.

EXPLAIN

Obtiene el plan de ejecución de un consulta, explica cómo MySQL ejecuta una consulta.

Requiere de privilegio `SHOW_VIEW` para poder ser usado por un usuario.

La sentencia no termina con `;`, sino con `\G`.

```
EXPLAIN  [ FORMAT = [ TRADITIONAL | TREE | JSON ] ]  query_sql \G
```

Usando `FORMAT=JSON` muestra la información de forma más amigable y más completa.

Se puede usar en sentencias SELECT, DELETE, INSERT, REPLACE, UPDATE, `TABLE`.

Ejemplo:

```
EXPLAIN FORMAT = JSON SELECT * FROM tablaDato \G
```

EXPLAIN ANALYZE

Ejecuta sentencias y produce una salida EXPLAIN con tiempo e información adicional, basada en iteradores, sobre cómo las expectativas del optimizador coincidieron con la ejecución real.

Muestra información de:

- Costo de ejecución.
- Filas retornadas.
- Tiempo de retorno primera fila.
- Tiempo gastado en el iterador.
- Número de filas retornadas por el iterador.
- Número de loops.

```
EXPLAIN ANALYZE [ FORMAT = [ TREE |  TRADITIONAL ] ]  query_sql  \G
```

mysqlslap

> mysqlslap - doc

Es una herramienta de diagnóstico que emula carga de cliente en un servidor MySQL y reporta el tiempo de cada etapa, simula el uso de clientes múltiples en el servidor.

`mysqlslap` se ejecuta en 3 estados:

1. Crea un esquema, tabla y opcionalmente cualquier programa almacenado o información para usarlo para la prueba. Es una etapa que usa una conexión única.
2. Ejecuta la carga de pruebas. Es una etapa de múltiples conexiones de clientes.
3. Limpia (desconecta, elimina tablas si se especifica). Esta etapa usa una conexión de cliente única.

```
mysqlslap [opciones]
```

Opciones	Descripción
`--create-schema`	especifica el esquema que se usará las pruebas.
`--delimiter`	delimitador sentencias SQL.
`--create`	fichero o string que contiene la sentencia de uso para crear la tabla.
`--query`	fichero o string que contiene una sentencia SELECT para recibir la información.
`--concurrency`	número de clientes para simular cuando se está usando sentencia SELECT.
`--iterations`	número de iteraciones para ejecutar pruebas.
`--user`	usuario usado para la conexión.
`--password`	contraseña usada para conexión.
`--port`	numero de puerto TCP/IP.
`--host`	servidor MySQL.

```
mysqlslap --delimiter=";"
  --query="SELECT * FROM table"
  --concurrency=50
  --iterations=200
```

Uso mysqlslap

Comparando una tabla sin clave primaria, foránea o índice contra otra tabla que tiene clave primaria, foránea e índice.

En los siguientes casos se utilizaron tablas InnoDB. Recordar que MyISAM no soportan *Foreign Key*.

```
mysqlslap --user root -pparalelepipedo --concurrency=5 --iterations=15 --create-schema=ventas_jugos --q
```

s_tb_factura - Sin *PK*, *FK*, *INDEX* Resultado:

```
Benchmark
    Average number of seconds to run all queries: 0.204 seconds
    Minimum number of seconds to run all queries: 0.190 seconds
    Maximum number of seconds to run all queries: 0.233 seconds
    Number of clients running queries: 5
    Average number of queries per client: 1
```

```
mysqlslap --user root -pparalelepipedo --concurrency=5 --iterations=15 --create-schema=ventas_jugos --q
```

tb_factura - Con *PK*, *FK*, *INDEX* Resultado:

```
Benchmark
    Average number of seconds to run all queries: 0.001 seconds
    Minimum number of seconds to run all queries: 0.001 seconds
    Maximum number of seconds to run all queries: 0.002 seconds
    Number of clients running queries: 5
    Average number of queries per client: 1
```

Los resultados son contundentes, se debe utilizar *PK*, *FK*, e *INDEX* en tablas

s_tb_factura - agregado INDEX Ahora la tabla *s_tb_factura* tendrá un índice en el campo `FECHA_VENTA`, aún no tiene *PK* y *FK*.

```
mysqlslap --user root -pparalelepipedo --concurrency=5 --iterations=15 --create-schema=ventas_jugos --q
```

Resultado:

```
Benchmark
    Average number of seconds to run all queries: 0.001 seconds
    Minimum number of seconds to run all queries: 0.001 seconds
    Maximum number of seconds to run all queries: 0.003 seconds
    Number of clients running queries: 5
    Average number of queries per client: 1
```

Como se ve en el resultado, esta consulta se demoró mucho menos gracias al índice en *FECHA_VENTA*.

Mecanismos de almacenamiento

> InnoDB - doc Alternativas a InnoDB - doc

Es la forma en que se almacenan la información en las tablas dentro de base de datos, una base de datos puede usar varios mecanismos en sus tablas.

MySQL soporta hasta 9 motores de almacenamiento de los cuales destacan 3 por que son más utilizados.

- `ENGINE` : indica el mecanismo de almacenamiento empleado en una tabla.

MySQL tiene 3 motores principales.

- InnoDB (default)
- MyISAM
- MEMORY

Creando tablas estableciendo un tipo de motor de almacenamiento o modificar la tabla para establecer un motor determinado.

SHOW ENGINES;

Muestra la información del motor utilizado.

```
SHOW ENGINES;
```

SHOW TABLE STATUS

Muestra información sobre la tabla creada.

```
SHOW TABLE STATUS WHERE Name = 'nombre_tabla';
```

MyISAM

Se establece al momento de crear una tabla o se puede modificar mediante ALTER TABLE.

No es transaccional, no es para usuarios múltiples transacciones.

Solamente permite operaciones de lectura.

Tablas que varían en el tiempo, más estáticas, no se modifican en un largo tiempo y se realizan operaciones mayoritariamente de lecturas.

Para cualquier alteración, esta tabla completa se debe bloquear primero para realizar respaldo.

Se debe actualizar la tabla de índice ante cualquier cambio en la tabla. Entre más índices más tiempo de ejecución. Debido a que no soporta clave foránea.

Clave foránea no soporta *Full Text*.

Almacena datos de forma más compacta, optimiza el espacio de almacenamiento.

Tiene la ventaja de implementar índices *HASH* y *BTREE*.

Variables

- *key_buffer_size* : determina el caché para almacenar índices MyISAM. 8MB - 4GB depende del OS.
- *concurrent_insert* : controla el comportamiento de inserciones concurrentes dentro de una tabla MyISAM.
  - 0 = inserciones simultáneas desactivadas.
  - 1 = inserciones simultáneas sin intervalo de datos.
  - 2 = inserciones simultáneas con intervalo de datos.
- *delay_key_write* : delay entre actualización de índices y el momento en que se crea la tabla. Espera que los registros se inserten y luego actualiza los índices. + Consistencia, - Rapidez.
- *max_write_lock_count* : determina el número de grabaciones en las tablas antes que las lecturas.
- *preload_buffer_size* : tamaño del buffer a ser usado antes de cargar los índices de caché de claves de las tablas: 32 KB.

Herramientas MyISAM

- *myisamchk* : analiza, optimiza y repara tablas MyISAM.
- *myisampack* : crea tablas MyISAM compactas de solo lectura.
- *myisam_ftdump* : exhibe información más completa de los campos de tipo texto.

InnoDB

Mecanismo por defecto, es transaccional, permite operaciones múltiples de usuarios simultáneos.

Se establece al momento de crear una tabla o se puede modificar mediante ALTER TABLE.

Se ordena automáticamente con PRIMARY KEY.

Soporte de claves externas.

Al crear clave primaria, automáticamente se crea un índice en dicha tabla. Se puede crear un índice usando una fila y la clave primaria de la fila.

Cache de buffer configurado de forma separada tanto de la base como el índice.

Bloque de tabla a nivel de línea.

Índice *BTREE*.

Soporta respaldo online.

Variables

- *innodb_data_file_path* : determina la ruta y el tamaño máximo del fichero en donde se almacenará la información.
- *innodb_data_home_dir* : ruta en donde se almacenarán todos los ficheros *innodb*. Por defecto **/mysql-data**.
- *innodb_file_per_table* : separa el almacenamiento de los datos y sus índices en ficheros. Por defecto almacena los datos e índices de forma compartida.
- *innodb_buffer_pool_size* : determina tamaño de almacenamiento que innodb utiliza para almacenar índices y datos en cache.
- *innodb_flush_log_at_trx_commit* : determina la frecuencia con que el buffer de log-in es habilitado en el disco. Crece con el uso y luego de un tiempo es guardado en el disco duro.
- *innodb_log_file_size* : tamaño en Bytes para los ficheros logs. Por defecto es 5MB.

MEMORY

Mecanismo de almacenamiento que creas tablas en la memoria volátil (RAM) no en el disco duro.

Se establece al momento de crear una tabla o se puede modificar mediante ALTER TABLE.

No soporta clave externa.

Necesita de una tabla auxiliar, se debe crear un índice que guarde una columna y su ubicación ser guardada y referenciada.

Muy rápida.

Bloqueo a nivel de tabla.

Índice por defecto *HASH* y *BTREE*.

Formato de línea de longitud fija, no soporta tipos de datos *BLOB/TEXT*.

Seguridad

Por defecto, al instalar MySQL este viene con un usuario por defecto **root**, este usuario tiene control sobre todo el servidor MySQL, esto es un riesgo de seguridad debido a que un atacante lo primero que hará es tratar de ingresar con **root**.

Por esto, se debe eliminar este usuario, pero antes se debe crear otro usuario que tenga los mismos privilegios que **root**.

En *Workbench* en el menú *Server*, *Users and Privileges* permite administrar los usuarios y privilegios en un entorno gráfico.

Mediante el uso de consola también se puede realizar, Administración y permisos de usuario.

Resumen con ejemplos

- Select

```
SELECT "nom de colonne" FROM "nombre_tabla";
```

- Distinct

```
SELECT DISTINCT "nombre_columna"
FROM "nombre_tabla";
```

- Where

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "condition";
```

- And/Or

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "condición simple"
{[AND|OR] "condición simple"}+;
```

- In

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" IN ('valor1', 'valor2', ...);
```

- Between

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" BETWEEN 'valor1' AND 'valor2';
```

- Like

```
SELECT "nombre_columna"
FROM "nombre_tabla"
WHERE "nombre_columna" LIKE {patrón};
```

- Order By

```
SELECT "nombre_columna"
FROM "nombre_tabla"
[WHERE "condición"]
ORDER BY "nombre_columna" [ASC, DESC];
```

- Count

```
SELECT COUNT("nombre_columna")
FROM "nombre_tabla";
```

- Group By

```
SELECT "nombre_columna 1", SUM("nombre_columna 2")
FROM "nombre_tabla"
GROUP BY "nombre_columna 1";
```

- Having

```
SELECT "nombre_columna 1", SUM("nombre_columna 2")
FROM "nombre_tabla"
GROUP BY "nombre_columna 1"
HAVING (condición de función aritmética);
```

- Create Table

```
CREATE TABLE "nombre_tabla"
("columna 1" "tipo_de_datos_para_columna_1",
"columna 2" "tipo_de_datos_para_columna_2",
... );
```

- Drop Table

```
DROP TABLE "nombre_tabla";
```

- Truncate Table

```
TRUNCATE TABLE "nombre_tabla";
```

- Insert Into

```
INSERT INTO 'nombre_tabla' ('colonne 1', 'colonne 2', ...)
VALUES ('valor 1', 'valor 2', ...);
```

Update

```
UPDATE "nombre_tabla"
SET "colonne 1" = [nuevo valor]
WHERE "condición";
```

Delete From

```
DELETE FROM "nombre_tabla"
WHERE "condición";
```

- JOIN

Muestra los clientes que excedieron en un 50% el volumen máximo del cliente durante el año 2018.

```sql
SELECT
A.DNI,
A.NOMBRE,
A.MES_AÑO,
A.CANTIDAD_MAXIMA - A.CANTIDAD_VENDIDA AS DIFERENCIA,
A.CANTIDAD_MAXIMA / 2 AS '50%'
FROM
(
    SELECT
        F.DNI,
        TC.NOMBRE,
        DATE_FORMAT(F.FECHA_VENTA, "%m - %Y") AS MES_AÑO,
        SUM(IFa.CANTIDAD) AS CANTIDAD_VENDIDA,
        MAX(TC.VOLUMEN_DE_COMPRA / 10) AS CANTIDAD_MAXIMA
        FROM
        facturas F
        INNER JOIN items_facturas IFa ON F.NUMERO = IFa.NUMERO
        INNER JOIN tabla_de_clientes TC ON TC.DNI = F.DNI
        GROUP BY F.DNI, TC.NOMBRE, DATE_FORMAT(F.FECHA_VENTA, "%m - %Y")
) AS A
WHERE
A.CANTIDAD_MAXIMA - A.CANTIDAD_VENDIDA < 0
AND
(A.CANTIDAD_MAXIMA - A.CANTIDAD_VENDIDA) * -1 > A.CANTIDAD_MAXIMA / 2;
```

Ordena las ventas por tamaño de botella, de orden descendente.

```sql
SELECT
VENTAS_TAMANO.TAMANO,
VENTAS_TAMANO.AÑO,
VENTAS_TAMANO.CANTIDAD_TOTAL,
ROUND((VENTAS_TAMANO.CANTIDAD_TOTAL / VENTA_TOTAL.CANTIDAD_TOTAL)*100,2)
AS PORCENTAJE
FROM (
    SELECT
    P.TAMANO,
    SUM(IFa.CANTIDAD) AS CANTIDAD_TOTAL,
    YEAR(F.FECHA_VENTA) AS AÑO
    FROM
    tabla_de_productos P
    INNER JOIN
    items_facturas IFa
    ON P.CODIGO_DEL_PRODUCTO = IFa.CODIGO_DEL_PRODUCTO
    INNER JOIN
    facturas F
    ON F.NUMERO = IFa.NUMERO
    WHERE YEAR(F.FECHA_VENTA) = 2016
    GROUP BY P.TAMANO, YEAR(F.FECHA_VENTA)
    ORDER BY SUM(IFa.CANTIDAD) DESC
) AS VENTAS_TAMANO
INNER JOIN
(
    SELECT SUM(IFa.CANTIDAD) AS CANTIDAD_TOTAL,
```

```
    YEAR(F.FECHA_VENTA) AS AÑO
    FROM
    tabla_de_productos P
    INNER JOIN
    items_facturas IFa
    ON P.CODIGO_DEL_PRODUCTO = IFa.CODIGO_DEL_PRODUCTO
    INNER JOIN
    facturas F
    ON F.NUMERO = IFa.NUMERO
    WHERE YEAR(F.FECHA_VENTA) = 2016
    GROUP BY YEAR(F.FECHA_VENTA)
) AS VENTA_TOTAL
ON VENTA_TOTAL.AÑO = VENTAS_TAMANO.AÑO
ORDER BY VENTAS_TAMANO.CANTIDAD_TOTAL DESC;
```