SQL

Última actualización: *May 01 2023*

Temas

Introducción

SQL significa *Structured Query Language*, es un lenguaje de programación que se utiliza para manejar bases de datos relacionales. *SQL* se utiliza para crear, modificar y consultar bases de datos y tablas, insertar y actualizar registros, borrar y eliminar datos, y realizar otras operaciones relacionadas con la gestión de datos.

SQL es un estándar de facto en el mundo de las bases de datos relacionales, lo que significa que se utiliza en la mayoría de los sistemas de bases de datos relacionales y que es compatible con diferentes sistemas y plataformas. También es un lenguaje fácil de aprender y muy utilizado en el mundo empresarial.

Los inicios de *SQL* se remonta a la década de 1970, cuando un equipo de investigadores de *IBM* liderado por *Donald D. Chamberlin* y *Raymond F. Boyce* creó el lenguaje de programación *SEQUEL* (Structured English Query Language) para gestionar datos en sistemas de bases de datos relacionales.

SEQUEL se convirtió en *SQL* cuando se desarrolló el sistema de gestión de bases de datos relacionales (*RDBMS*) llamado *System R* de *IBM*.

En la década de 1980, la empresa *Oracle Corporation* también comenzó a utilizar *SQL* en su sistema de bases de datos relacionales *Oracle Database*. A medida que los sistemas de bases de datos relacionales se hicieron más populares, *SQL* se convirtió en el estándar de facto para interactuar con ellos.

En la década de 1990, *SQL* se convirtió en un estándar *ANSI* y se añadieron características más avanzadas, como consultas complejas con múltiples tablas y subconsultas. A lo largo de las décadas siguientes, SQL continuó evolucionando y añadiendo nuevas características, como funciones de agregación, procedimientos almacenados y desencadenadores.

Hoy en día, *SQL* es el lenguaje de programación más utilizado en el mundo de las bases de datos relacionales, y es una habilidad esencial para aquellos que trabajan en campos como la programación, la gestión de datos y la inteligencia empresarial.

🔼 Regresar

Tipos de *SGBDs*

Estos son algunos ejemplos de sistemas de gestión de bases de datos basados en *SQL*, más populares.

1. ***MySQL***: es un sistema de gestión de bases de datos relacional de código abierto desarrollado por *Oracle*. Es uno de los sistemas de bases de datos más populares y ampliamente utilizados en el mundo.
2. ***PostgreSQL***: es otro sistema de gestión de bases de datos relacional de código abierto, que se enfoca en la conformidad con los estándares y la extensibilidad. Es conocido por su capacidad para manejar grandes volúmenes de datos y su soporte para funciones avanzadas como la replicación y la georreferenciación.
3. ***Microsoft SQL Server***: es un sistema de gestión de bases de datos relacional desarrollado por *Microsoft*. Se utiliza principalmente en entornos empresariales y es compatible con una amplia variedad de aplicaciones de software de *Microsoft*.
4. ***Oracle Database***: es un sistema de gestión de bases de datos relacional desarrollado por *Oracle*. Es uno de los sistemas de bases de datos más antiguos y ampliamente utilizados en el mundo empresarial, y se utiliza principalmente en grandes empresas y organizaciones gubernamentales.
5. ***SQLite***: es un sistema de gestión de bases de datos relacional ligero que se utiliza principalmente en aplicaciones móviles y de escritorio. Es conocido por su facilidad de uso y su capacidad para manejar pequeñas bases de datos.
6. ***MariaDB***: es un sistema de gestión de bases de datos de código abierto basado en el lenguaje *SQL*. Fue creado como un *fork* de *MySQL* en 2009, después de que la empresa matriz de *MySQL*, *Sun*

Microsystems, fuera adquirida por *Oracle Corporation*. El objetivo principal de la creación de *MariaDB* era proporcionar una alternativa de código abierto a *MySQL* y asegurar la continuidad del desarrollo.

🔼 Regresar

Tipos de Datos

Tipo	Descripción
VARCHAR	Cadena de texto variable, con una longitud máxima especificada.
CHAR	Cadena de texto fija, con una longitud específica.
INT	Número entero, positivo o negativo.
BIGINT	Número entero grande, positivo o negativo.
FLOAT	Número decimal de precisión simple.
DOUBLE	Número decimal de doble precisión.
DECIMAL	Número decimal con precisión fija.
DATE	Fecha, con valores de año, mes y día.
TIME	Hora, con valores de hora, minutos y segundos.
DATETIME	Fecha y hora combinadas.
TIMESTAMP	Marca de tiempo, que indica un momento específico en el tiempo.
BOOLEAN	Valor booleano, verdadero o falso.
BLOB	Objeto binario grande, para almacenar datos binarios, como imágenes o archivos.
JSON	Formato de texto estructurado para el intercambio de datos.

🔼 Regresar

Tipos de Sentencias *SQL*

En *SQL* existen varios tipos de sentencias que se utilizan para realizar diferentes operaciones en una base de datos. Aquí te presento algunos de los tipos de sentencias más comunes:

1. **Sentencias *DDL* (*Data Definition Language*)**: se utilizan para definir y modificar la estructura de la base de datos. Por ejemplo para crear o modificar la estructura de una tabla.
2. **Sentencias *DML* (*Data Manipulation Language*)**: se utilizan para manipular los datos dentro de la base de datos. Por ejemplo las instrucciones del *CRUD* de datos (*INSERT*, *SELECT*, *UPDTE* y *DELETE*).
3. **Sentencias *DCL* (*Data Control Language*)**: se utilizan para controlar el acceso a la base de datos y garantizar la seguridad. Por ejemplo para otorgar permisos a los usuarios para acceder a las bases de datos.

4. **Sentencias *TCL* (*Transaction Control Language*)**: se utilizan para controlar las transacciones en una base de datos.

Estos son solo algunos ejemplos de los tipos de sentencias en SQL. Es importante tener en cuenta que diferentes sistemas de bases de datos pueden tener variaciones en las sentencias específicas que se utilizan, pero los conceptos básicos de los tipos de sentencias generalmente se aplican a la mayoría de los sistemas de bases de datos relacionales.

🔼 Regresar

Comandos y Funciones *SQL*

A continuación te enlisto, algunos de los comandos y funciones más utilizados en *SQL*.

Comandos *SQL*

Comando	Descripción
SELECT	Recupera datos de una o varias tablas. Es una de las sentencias más comunes en SQL.
INSERT	Agrega nuevos datos a una tabla.
UPDATE	Actualiza los datos existentes en una tabla.
DELETE	Elimina datos de una tabla.
CREATE	Crea una nueva tabla, vista, índice, procedimiento almacenado u otra estructura de base de datos.
ALTER	Modifica la estructura de una tabla existente, como agregar o eliminar columnas.
DROP	Elimina una tabla, vista, índice, procedimiento almacenado u otra estructura de base de datos.
GRANT	Concede permisos a un usuario o grupo de usuarios para realizar operaciones en una tabla o en la base de datos en general.
REVOKE	Retira los permisos previamente otorgados a un usuario o grupo de usuarios.
JOIN	Combina datos de dos o más tablas en función de una columna común.
UNION	Combina los resultados de dos o más consultas en una sola tabla.
GROUP BY	Agrupa los resultados de una consulta en función de una o más columnas.
HAVING	Permite filtrar los resultados de una consulta agrupada.
ORDER BY	Ordena los resultados de una consulta en función de una o más columnas.
LIMIT	Limita el número de filas devueltas por una consulta.

Funciones *SQL*

Función	Descripción
AVG()	Devuelve el valor promedio de una columna numérica.
COUNT()	Cuenta el número de filas o valores distintos en una columna.
MAX()	Devuelve el valor máximo de una columna.
MIN()	Devuelve el valor mínimo de una columna.
SUM()	Devuelve la suma de los valores de una columna numérica.
CONCAT()	Concatena dos o más cadenas de texto.
SUBSTRING()	Devuelve una parte de una cadena de texto.
UPPER()	Convierte una cadena de texto a mayúsculas.
LOWER()	Convierte una cadena de texto a minúsculas.
LEFT()	Devuelve los caracteres iniciales de una cadena de texto.
RIGHT()	Devuelve los caracteres finales de una cadena de texto.
DATE()	Extrae la fecha de un valor de fecha y hora.
YEAR()	Devuelve el año de una fecha.
MONTH()	Devuelve el mes de una fecha.
DAY()	Devuelve el día de una fecha.
ROUND()	Redondea un valor numérico al número de decimales especificado.

🔼 Regresar

Sintaxis *SQL* Básica

Conceptos básicos

- Una **base de datos** tiene **tablas**.
- Una **tabla** tiene **campos** (columnas) y **registros** (filas).
- El conjunto de campos generan un registro.
- **Campo**: un dato que por sí sólo no dice mucho.
- **Registro**: conjunto de campos que genera información.

Buenas prácticas

SQL, NO distingue entre MÁYUSCULAS y minúsculas pero:

- Comandos y palabras reservadas de SQL van en MÁYUSCULAS.
- Nombres de objetos y datos van en minúsculas con *snake_case*.
- Para strings usar comillas simples (`''`).
- Todas las sentencias terminan con punto y coma (`;`).

🔼 Regresar

Gestionando Bases de Datos

```sql
SHOW DATABASES;

CREATE DATABASE nombre_base;

CREATE DATABASE IF NOT EXISTS nombre_base;

DROP DATABASE nombre_base;

DROP DATABASE IF EXISTS nombre_base;

USE nombre_base;
```

🔼 Regresar

Usuarios y Privilegios

```sql
CREATE USER 'mi_usuario'@'servidor' IDENTIFIED BY 'mi_password';

--contraseña en hash
SELECT PASSWORD('mi_password');

DROP USER 'mi_usuario'@'servidor';

GRANT ALL PRIVILEGES ON nombre_base.tabla TO 'mi_usuario'@'servidor' IDENTIFIED BY 'mi_password';

FLUSH PRIVILEGES;

SHOW GRANTS for 'mi_usuario'@'servidor';

REVOKE ALL, GRANT OPTION FROM 'mi_usuario'@'servidor';
```

🔼 Regresar

Gestionando Tablas

```sql
SHOW TABLES;

DESCRIBE nombre_tabla;
```

```sql
CREATE TABLE nombre_tabla(
  campo1 TIPO_DATO ATRIBUTOS,
  campo2 TIPO_DATO
);

ALTER TABLE nombre_tabla ADD COLUMN nombre_campo TIPO_DATO;

ALTER TABLE nombre_tabla MODIFY nombre_campo TIPO_DATO;

ALTER TABLE nombre_tabla RENAME COLUMN nombre_viejo TO nombre_nuevo;

ALTER TABLE nombre_tabla DROP COLUMN nombre_campo;

DROP TABLE nombre_tabla;

--Ejemplo de Tabla
CREATE TABLE usuarios(
    usuario_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(30) NOT NULL,
    apellidos VARCHAR(30) NOT NULL,
    correo VARCHAR(50) UNIQUE,
    direccion VARCHAR(100) DEFAULT "Sin dirección",
    edad INT DEFAULT 0
);
```

🔼 Regresar

CRUD de Datos

- Create - `INSERT`.
- Read - `SELECT`.
- Update - `UPDATE`.
- Delete - `DELETE`.

CREATE

Insertar un registro:

```sql
INSERT INTO tabla (campo_1, campo_2, ..., campo_n)
  VALUES (valor_1, valor2, ..., valor_n);

INSERT INTO tabla
  SET campo_1 = 'valor_1', campo_2 = 'valor_2', ..., campo_n = valor_n;
```

Insertar varios registros:

```sql
INSERT INTO tabla (campo_1, campo_2, ..., campo_n) VALUES
  (valor_1, valor2, ..., valor_n),
  (valor_1, valor2, ..., valor_n),
  ...,
  (valor_1, valor2, ..., valor_n);
```

READ

Leer todos los campos de la tabla:

```
SELECT * FROM tabla;
```

Leer algunos campos de la tabla:

```
SELECT campo_1, campo_2, campo_n FROM tabla;
```

Saber cuantos registros tiene mi tabla:

```
SELECT COUNT(*) FROM tabla;
```

Leer un registro en particular buscando el valor de un campo:

```
SELECT * FROM tabla WHERE campo_1 = 'valor_1';
```

Leer registros en particular buscando diferentes valores en un campo:

```
SELECT * FROM tabla WHERE campo_1 IN ('valor_1', 'valor_2', 'valor_n');
```

Leer un registro en particular buscando el valor similar en un campo:

```
SELECT * FROM tabla WHERE campo_1 LIKE '%valor_1';
SELECT * FROM tabla WHERE campo_1 LIKE 'valor_1%';
SELECT * FROM tabla WHERE campo_1 LIKE '%valor_1%';
```

Leer un registro en particular buscando el valor con operadores lógicos:

```
SELECT * FROM tabla WHERE campo_1 = 'valor_1' AND campo_2 = 'valor_2';
SELECT * FROM tabla WHERE campo_1 = 'valor_1' OR campo_2 = 'valor_2';
SELECT * FROM tabla WHERE NOT campo_1 = 'valor_1';
SELECT * FROM tabla WHERE campo_1 != 'valor_1';
```

UPDATE

Siempre agregar la clausula ***WHERE*** para evitar actualizar toda la tabla:

```
UPDATE tabla
  SET campo_1 = 'valor_1', campo_2 = 'valor_2', ..., campo_n = valor_n
  WHERE campo = valor;
```

DELETE

Siempre agregar la clausula ***WHERE*** para evitar eliminar toda la tabla:

NO TE OLVIDES DEL WHERE EN EL DELETE FROM

```
DELETE FROM tabla WHERE campo = valor;
```

🔼 Regresar

Sentencias de Agrupamiento

GROUP BY

La cláusula *GROUP BY* se utiliza para agrupar los registros en una consulta basándose en una o más columnas.

Supongamos que tenemos la siguiente tabla llamada "*Ventas*":

id	producto	cantidad	fecha
1	Zapatos	5	2022-03-01
2	Camisas	3	2022-03-02
3	Zapatos	2	2022-03-03
4	Pantalón	4	2022-03-03
5	Camisas	7	2022-03-04

Podemos utilizar la cláusula *GROUP BY* para obtener la cantidad total de cada producto vendido, agrupando por el nombre del producto.

```
SELECT producto, SUM(cantidad) AS total_vendido
  FROM ventas
  GROUP BY producto;
```

Esta consulta agrupa los registros de la tabla "*Ventas*" por la columna "producto" y calcula la suma total de la columna "cantidad" para cada producto. El resultado sería el siguiente:

producto	total_vendido
Zapatos	7
Camisas	10
Pantalón	4

Como puedes ver, la cláusula *GROUP BY* es muy útil para realizar operaciones de agregación, como contar, sumar o promediar valores, en diferentes grupos de registros.

🔼 Regresar

HAVING

La cláusula *HAVING* se utiliza en *SQL* para filtrar los resultados de una consulta que utiliza la cláusula *GROUP BY*.

Supongamos que tenemos una tabla llamada "*Ventas*" que contiene información sobre las ventas de una empresa:

id	producto	cantidad	fecha
1	A	10	2022-01-01
2	B	15	2022-01-02
3	C	20	2022-01-02
4	A	5	2022-01-03
5	B	8	2022-01-03
6	C	12	2022-01-04

Si queremos obtener la cantidad total de ventas para cada producto, podemos utilizar la cláusula *GROUP BY* de la siguiente manera:

```
SELECT producto, SUM(cantidad) AS total_ventas
  FROM ventas
  GROUP BY producto;
```

Esta consulta devuelve el siguiente resultado:

producto	total_ventas
A	15
B	23
C	32

Ahora, supongamos que queremos obtener solamente los productos que han tenido un total de ventas mayor a 20. Para ello, podemos utilizar la cláusula *HAVING* de la siguiente manera:

```
SELECT producto, SUM(cantidad) AS total_ventas
  FROM ventas
  GROUP BY producto
  HAVING SUM(cantidad) > 20;
```

Esta consulta devuelve el siguiente resultado:

producto	total_ventas
B	23
C	32

Como puedes ver, la cláusula *HAVING* nos permite filtrar los resultados de una consulta que utiliza GROUP BY, basándonos en una condición que se aplica a los resultados agrupados. En este caso, hemos filtrado los productos que han tenido un total de ventas mayor a 20.

🔼 Regresar

DISTINCT

La cláusula *DISTINCT* se utiliza en *SQL* para eliminar las filas duplicadas de un conjunto de resultados.

Supongamos que tenemos la siguiente tabla llamada "*Clientes*":

id	nombre	apellido
1	Juan	Perez
2	Ana	Garcia
3	Juan	Martinez
4	Maria	Rodriguez
5	Ana	Jimenez

Si queremos obtener la lista de nombres únicos de los clientes, podemos utilizar la cláusula DISTINCT de la siguiente manera:

```
SELECT DISTINCT nombre
  FROM Clientes;
```

Esta consulta devuelve los nombres únicos de los clientes de la tabla "*Clientes*", sin importar si tienen apellidos diferentes. El resultado sería el siguiente:

nombre
Juan
Ana
Maria

Como puedes ver, la cláusula *DISTINCT* nos permite obtener resultados únicos y reducir la cantidad de datos redundantes en las consultas *SQL*.

🔼 Regresar

ORDER BY

La cláusula *ORDER BY* en *SQL* se utiliza para ordenar los resultados de una consulta en un orden específico. Se puede ordenar por una o varias columnas y en orden ascendente (*ASC*) o descendente (*DESC*).

Por ejemplo, si tenemos una tabla "*empleados*" con las columnas "nombre", "apellido" y "salario", podemos ordenar los registros por el salario de forma ascendente con la siguiente consulta:

```sql
SELECT * FROM empleados ORDER BY salario ASC;
```

Esto nos devolvería todos los registros de la tabla "*empleados*" ordenados por el salario de forma ascendente. Si quisiéramos ordenarlos de forma descendente, cambiaríamos "*ASC*" por "*DESC*":

```sql
SELECT * FROM empleados ORDER BY salario DESC;
```

De esta manera, se pueden ordenar los resultados de una consulta de acuerdo a un criterio específico.

🔼 Regresar

LIMIT

La cláusula *LIMIT* se utiliza en *SQL* para limitar el número de resultados devueltos en una consulta. Permite especificar el número de filas que se deben recuperar desde la tabla, lo que puede ser útil en consultas que devuelven grandes cantidades de datos.

La sintaxis básica de la cláusula *LIMIT* es la siguiente:

```sql
SELECT columna_1, columna_2, ..., colunmna_n
  FROM tabla
  LIMIT cantidad_de_filas;
```

Donde **cantidad_de_filas** es el número máximo de filas que se deben devolver en la consulta.

También es posible especificar un punto de inicio desde el cual se deben recuperar las filas, lo que se logra utilizando dos valores separados por una coma. El primer valor especifica el índice de la primera fila que se debe devolver, y el segundo valor especifica el número máximo de filas que se deben devolver.

```sql
SELECT columna_1, columna_2, ..., columna_n
  FROM tabla
  LIMIT indice_inicio, cantidad_de_filas;
```

Donde **indice_inicio** es el índice de la primera fila que se debe devolver, y **cantidad_de_filas** es el número máximo de filas que se deben devolver a partir de la fila de inicio.

Por ejemplo, la siguiente consulta devuelve los primeros 10 registros de la tabla "*clientes*":

```sql
SELECT * FROM clientes
  LIMIT 10;
```

Y la siguiente consulta devuelve los registros 11 al 20 de la tabla "*clientes*":

```sql
SELECT * FROM clientes
  LIMIT 10, 10;
```

Es importante tener en cuenta que el uso de la cláusula *LIMIT* puede afectar el rendimiento de la consulta, especialmente cuando se utiliza con tablas grandes.

🔼 Regresar

Sintaxis *SQL* Avanzada

Índices

En *SQL* existen varios tipos de índices, los principales son:

- Índice único (***UNIQUE***): asegura que los valores de la columna indexada sean únicos en la tabla.
- Índice primario (***PRIMARY KEY***): es un tipo especial de índice único que identifica de forma única cada fila de una tabla.
- Índice secundario (***INDEX***): es un índice que no tiene restricciones de unicidad y se utiliza para mejorar el rendimiento de consultas que involucran la columna indexada.
- Índice de texto completo (***FULLTEXT***): se utiliza para hacer búsquedas de texto completo en columnas de texto grandes, como *VARCHAR* y *TEXT*.

Ejemplo

```
CREATE TABLE una_tabla(
  campo_id INTEGER UNSIGNED PRIMARY KEY,
  campo_unico VARCHAR(80) UNIQUE,
  campo_index VARCHAR(80),
  campo_3 VARCHAR(80),
  campo_4 VARCHAR(80),
  INDEX i_campo_index(campo_index)
  FULLTEXT INDEX fi_campo_fulltext(campo_3, campo_4)
);
```

Ejecutando una consulta de tipo *FULLTEXT*

```
SELECT * FROM una_tabla
  WHERE MATCH(campo_3, campo_4)
  AGAINST('una_búsqueda' IN BOOLEAN MODE);
```

🔼 Regresar

Foreign Keys

En *SQL*, una llave foránea (*Foreign Key*) es un campo o conjunto de campos en una tabla que hacen referencia a una clave única en otra tabla, estableciendo así una relación entre ambas tablas. Se utilizan para mantener la integridad referencial de los datos, lo que significa que garantizan que los datos en las tablas relacionadas sean coherentes y precisos.

Sintaxis

Se define dentro de una tabla de la siguiente forma:

```
FOREIGN KEY (nombre_campo) REFERENCES tabla_referencia(campo_referencia)
```

Ejemplo

```sql
CREATE TABLE lenguajes (
  lenguaje_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  lenguaje VARCHAR(30) NOT NULL
);

CREATE TABLE entornos (
  entorno_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  entorno VARCHAR(30) NOT NULL
);

CREATE TABLE frameworks (
  framework_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  framework VARCHAR(30) NOT NULL,
  lenguaje INT UNSIGNED,
  entorno INT UNSIGNED,
  FOREIGN KEY (lenguaje) REFERENCES lenguajes(lenguaje_id),
  FOREIGN KEY (entorno) REFERENCES entornos(entorno_id)
);
```

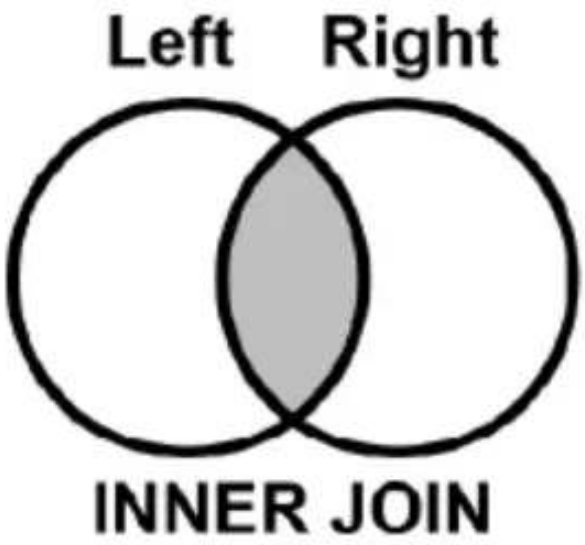
🔼 Regresar

JOINS

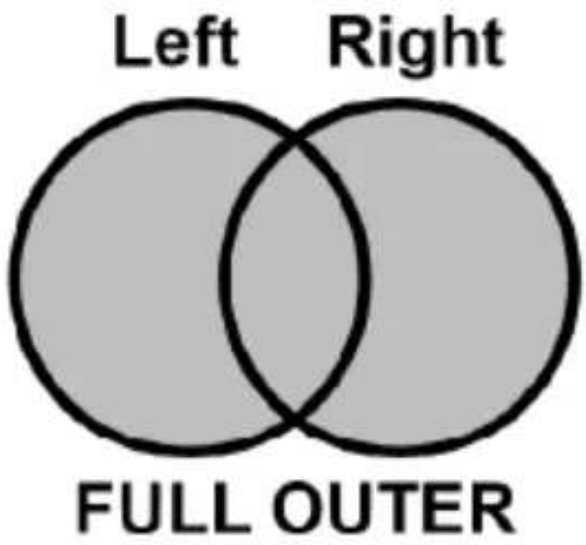
Los *JOINs* en *SQL* sirven para combinar filas de dos o más tablas basándose en un campo común entre ellas, devolviendo por tanto datos de diferentes tablas. Un *JOIN* se produce cuando dos o más tablas se juntan en una sentencia *SQL*.

Los más importantes son los siguientes:

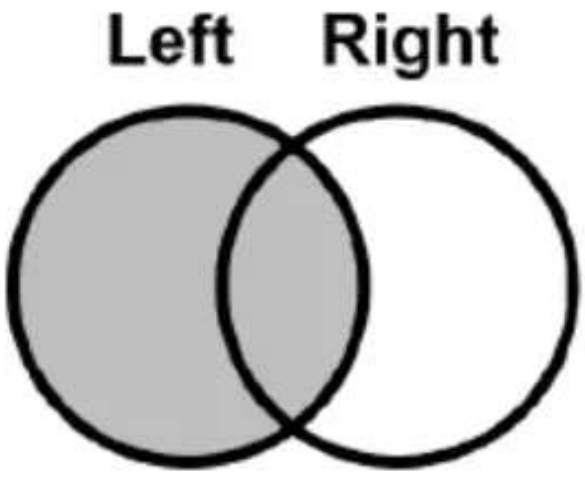
- *INNER JOIN*: Devuelve todas las filas cuando hay al menos una coincidencia en ambas tablas.
- *LEFT JOIN*: Devuelve todas las filas de la tabla de la izquierda, y las filas coincidentes de la tabla de la derecha.
- *RIGHT JOIN*: Devuelve todas las filas de la tabla de la derecha, y las filas coincidentes de la tabla de la izquierda.
- *OUTER JOIN*: Devuelve todas las filas de las dos tablas, la izquierda y la derecha, también se llama *FULL OUTER JOIN*.



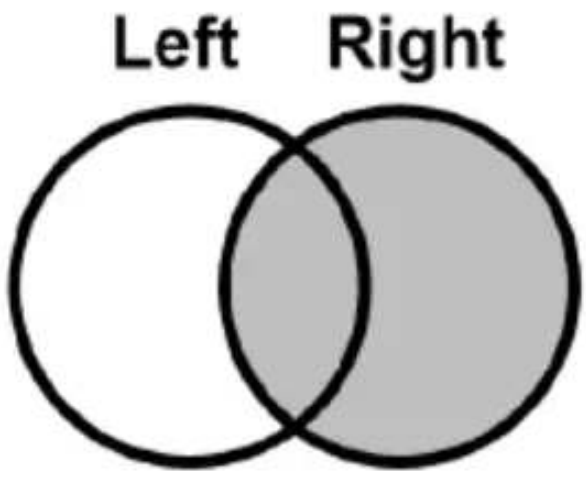
INNER JOIN



FULL OUTER JOIN



LEFT OUTER JOIN



RIGHT OUTER JOIN

```sql
SELECT * FROM tabla_1 AS t1
  INNER JOIN tabla_2 AS t2;

SELECT * FROM tabla_1 AS t1
  INNER JOIN tabla_2 AS t2
  ON t1.a_campo = t2.a_campo;

SELECT t1.campo_1, t1.campo_2, t1.campo_3, t2.campo_1, t2.campo_5
  FROM tabla_1 AS t1
  INNER JOIN tabla_2 AS t2
  ON t1.campo_1 = t2.campo_5
  WHERE t1.campo_1 = 'valor'
  ORDER BY t1.campo_3 DESC;

/* Con FULLTEXT */
SELECT t1.campo_1, t1.campo_2, t2.campo_1, t2.campo_4
  FROM tabla_1 AS t1
  INNER JOIN tabla_2 AS t2
  ON t1.campo_1 = t2.campo_4
  WHERE MATCH(t1.campo_1, t1.campo_2, t2.campo_1, t2.campo_4)
  AGAINST('una_búsqueda' IN BOOLEAN MODE);
```

🔼 Regresar

Subconsultas

Es una consulta dentro de otra:

```sql
SELECT t1.campo_1, t1.campo_2, (
    SELECT COUNT(*)
    FROM tabla_2 AS t2
    WHERE t2.campo_1 = t1.campo_1
  ) AS sub_consulta_campo
  FROM tabla_1 AS t1;

SELECT t1.campo_1, t1.campo_2, t1.campo_3, (
    SELECT campo_1
    FROM tabla_2 AS t2
    WHERE t2.campo_1 = t1.campo_1
  ) AS sub_consulta_campo
  FROM tabla_1 AS t1;
```

🔼 Regresar

Vistas

Una vista es una tabla virtual que se deriva de una o más tablas existentes en una base de datos. En otras palabras, una vista no almacena datos físicamente, sino que es una consulta predefinida que se ejecuta cada vez que se accede a ella.

La vista se define utilizando una consulta SELECT, y una vez definida, puede ser tratada como cualquier otra tabla en la base de datos, permitiendo que se realicen consultas, actualizaciones y eliminaciones de registros.

Las vistas son útiles porque pueden proporcionar una capa adicional de abstracción para los usuarios que no necesitan conocer los detalles de la estructura de la base de datos subyacente.

Por ejemplo, se puede crear una vista que presente solo ciertas columnas de una tabla, o que filtre los registros en función de ciertos criterios.

Además, las vistas pueden ser utilizadas para simplificar consultas complejas, ya que una vista puede contener una consulta que combine datos de varias tablas. En lugar de tener que escribir la consulta completa cada vez, los usuarios pueden simplemente consultar la vista.

```sql
CREATE VIEW nombre_vista AS
    SELECT * FROM tabla WHERE campo_1 = 'valor_1';

DROP VIEW nombre_vista;

SELECT * FROM nombre_vista;

SHOW FULL TABLES IN nombre_bd WHERE TABLE_TYPE LIKE 'VIEW';
```

🔼 Regresar

Motores de Tablas

Un motor de tablas (también conocido como motor de almacenamiento) en *SQL* es el componente del sistema de gestión de bases de datos (*SGBD*) que se encarga de la forma en que se almacenan, recuperan y manipulan los datos en una base de datos.

Los motores de tablas son responsables de la forma en que se organizan los datos físicamente en el disco, así como de cómo se accede a ellos y se realiza el mantenimiento de la base de datos.

Cada motor de tablas tiene su propia forma de gestionar las tablas, los índices, los bloqueos, las transacciones y la concurrencia. Algunos motores de tablas son más adecuados para aplicaciones de alta concurrencia y transacciones complejas, mientras que otros son más adecuados para aplicaciones con menos concurrencia y consultas más simples.

En MySQL, por ejemplo, existen varios motores de tablas disponibles, cada uno con sus propias características y limitaciones. Los motores de tablas más comunes son *InnoDB*, *MyISAM*, *MEMORY*, *CSV* y *ARCHIVE*.

InnoDB es el motor de almacenamiento predeterminado en *MySQL*, y es compatible con transacciones y restricciones de clave foránea, lo que lo hace más adecuado para aplicaciones que requieren una alta integridad de los datos.

MyISAM es un motor de almacenamiento antiguo que ya no se recomienda su uso debido a sus limitaciones en cuanto a la integridad referencial y la seguridad de los datos.

Los otros motores de tablas en *MySQL* tienen características específicas, como la capacidad de almacenar datos en memoria, acceder a datos en archivos *CSV* o comprimir datos.

Ejemplo

```sql
CREATE TABLE armaduras_myisam (
    armadura_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    armadura VARCHAR(30) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4;

CREATE TABLE armaduras_innodb (
    armadura_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    armadura VARCHAR(30) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

🔼 Regresar

Restricciones

En *SQL*, las restricciones *ON DELETE* y *ON UPDATE* se utilizan para especificar qué acciones se deben realizar en una tabla relacionada cuando se realiza una operación de eliminación o actualización en la tabla principal.

Las acciones que se pueden especificar para las restricciones *ON DELETE* y *ON UPDATE* son:

- *CASCADE*: elimina o actualiza automáticamente los registros relacionados en la tabla relacionada.
- *SET NULL*: establece los valores de la columna relacionada en *NULL* cuando se elimina o actualiza un registro en la tabla principal.

- *SET DEFAULT*: establece los valores de la columna relacionada en su valor predeterminado cuando se elimina o actualiza un registro en la tabla principal.
- *RESTRICT*: evita la eliminación o actualización de un registro en la tabla principal si hay registros relacionados en la tabla relacionada.

Ejemplo

```sql
CREATE TABLE lenguajes (
  lenguaje_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  lenguaje VARCHAR(30) NOT NULL
);

CREATE TABLE entornos (
  entorno_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  entorno VARCHAR(30) NOT NULL
);

CREATE TABLE frameworks (
  framework_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  framework VARCHAR(30) NOT NULL,
  lenguaje INT UNSIGNED,
  entorno INT UNSIGNED,
  FOREIGN KEY (lenguaje)
    REFERENCES lenguajes(lenguaje_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE,
  FOREIGN KEY (entorno)
    REFERENCES entornos(entorno_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
);
```

🔼 Regresar

Transacciones

En *SQL*, una transacción es una secuencia de operaciones que se ejecutan como una sola unidad lógica e indivisible, como si fueran una única operación. Una transacción garantiza la integridad de los datos y la consistencia de la base de datos en caso de errores o fallas.

Una transacción típica implica una serie de operaciones que se realizan en una base de datos, como agregar, modificar o eliminar registros en una o más tablas.

Todas las operaciones de la transacción se realizan como una sola unidad, lo que significa que todas las operaciones deben completarse con éxito o ninguna de ellas debe ser efectiva.

Para iniciar una transacción en *SQL*, se utiliza la sentencia *START TRANSACTION*. Luego, se realizan las operaciones de la transacción, y si todas ellas se ejecutan correctamente, se utiliza la sentencia *COMMIT* para confirmar los cambios en la base de datos.

En caso de que se produzca un error o falla en alguna de las operaciones, se utiliza la sentencia *ROLLBACK* para deshacer todos los cambios y volver a un estado consistente de la base de datos.

```sql
START TRANSACTION;

  INSERT INTO tabla_1 (campo_1, campo_2, campo_3)
    VALUES ('valor_1', 'valor_2', 'valor_3');

  INSERT INTO tabla_2 (campo_1, campo_2, campo_3)
    VALUES ('valor_1', 'valor_2', 'valor_3');

  INSERT INTO tabla_3 (campo_1, campo_2, campo_3)
    VALUES ('valor_1', 'valor_2', 'valor_3');

COMMIT;

START TRANSACTION;

  INSERT INTO tabla_1 (campo_1, campo_2, campo_3)
    VALUES ('valor_1', 'valor_2', 'valor_3');

  INSERT INTO tabla_2 (campo_1, campo_2, campo_3)
    VALUES ('valor_1', 'valor_2', 'valor_3');

  INSERT INTO tabla_3 (campo_1, campo_2, campo_3)
    VALUES ('valor_1', 'valor_2', 'valor_3');

ROLLBACK;
```

Encriptación

En *SQL* existen varias funciones de encriptación que se pueden utilizar para proteger la información sensible en las bases de datos.

Algunas de las funciones de encriptación más comunes son:

MD5

Esta función convierte una cadena de caracteres en un valor *hash* de 128 *bits*. El valor resultante es único para cada cadena de entrada y se utiliza comúnmente para almacenar contraseñas en la base de datos.

```sql
SELECT MD5('password');
```

SHA1

Esta función también genera un valor *hash*, pero utiliza un algoritmo más seguro que *MD5* y produce un valor *hash* de 160 *bits*.

```sql
SELECT SHA1('password');
```

SHA2

Esta función es similar a *SHA1*, pero permite especificar la longitud del valor hash generado (en *bits*). Se puede utilizar para crear valores *hash* más largos y más seguros que *SHA1*.

```sql
SELECT SHA2('password', 256);
```

AES_ENCRYPT y *AES_DECRYPT*

Estas funciones se utilizan para encriptar y desencriptar datos utilizando el algoritmo *AES*.

```sql
SELECT AES_ENCRYPT('password', 'secret_key'),
  AES_DECRYPT('encrypted_value', 'secret_key');
```

Es importante tener en cuenta que la encriptación no es una solución completa para la seguridad de la base de datos y se deben tomar otras medidas de seguridad para proteger la información sensible.

Ejemplo

```sql
CREATE TABLE pagos_recurrentes(
    cuenta VARCHAR(8) PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    tarjeta BLOB
);

INSERT INTO pagos_recurrentes VALUES
    ('12345678', 'Jon', AES_ENCRYPT('1234567890123488', '12345678')),
    ('12345677', 'Irma', AES_ENCRYPT('1234567890123477', '12345677')),
    ('12345676', 'Kenai', AES_ENCRYPT('1234567890123466', '12345676')),
    ('12345674', 'Kala', AES_ENCRYPT('1234567890123455', 'super_llave')),
    ('12345673', 'Miguel', AES_ENCRYPT('1234567890123444', 'super_llave'));

SELECT * FROM pagos_recurrentes;

SELECT CAST(AES_DECRYPT(tarjeta, '12345678') AS CHAR) AS tdc, nombre
    FROM pagos_recurrentes;

SELECT CAST(AES_DECRYPT(tarjeta, 'super_llave') AS CHAR) AS tdc, nombre
    FROM pagos_recurrentes;
```

🔼 Regresar

Procedimientos Almacenados

Un procedimiento almacenado o ***Stored Procedure*** en *SQL* es un conjunto de instrucciones que se almacenan en la base de datos y se pueden llamar y ejecutar varias veces mediante una sola llamada al procedimiento.

Estos procedimientos pueden aceptar parámetros de entrada y devolver valores de salida, y pueden ser utilizados para realizar operaciones complejas en la base de datos de manera eficiente y segura.

Los procedimientos almacenados también pueden ser utilizados para encapsular lógica de negocio y reducir la complejidad de las aplicaciones cliente al mover la lógica de la base de datos al servidor.

Sintaxis

```
DELIMITER //

CREATE PROCEDURE nombre_procedimiento(
  IN valor_entrada TIPO_DATO,
  IN valor_entrada_2 TIPO_DATO,
  OUT valor_salida TIPO_DATO
)

  BEGIN
    Código del Procedimiento Almacenado
  END //

DELIMITER ;
```

Una vez creado el procedimiento almacenado, podemos llamarlo con el siguiente código:

```
CALL nombre_procedimiento();
```

Eliminar un procedimiento y mostrar los procedimientos de una base de datos:

```
DROP PROCEDURE nombre_procedimiento();

SHOW PROCEDURE STATUS WHERE db = 'nombre_base_datos';
```

Ejemplo

Primero creamos las tablas necesarias para el ejemplo del procedimiento:

```
CREATE TABLE suscripciones (
  suscripcion_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  suscripcion VARCHAR(30) NOT NULL,
  costo DECIMAL(5,2) NOT NULL
);

INSERT INTO suscripciones VALUES
  (0, 'Bronce', 199.99),
  (0, 'Plata', 299.99),
  (0, 'Oro', 399.99);

CREATE TABLE clientes (
  cliente_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(30) NOT NULL,
  correo VARCHAR(50) UNIQUE
);

CREATE TABLE tarjetas (
  tarjeta_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  cliente INT UNSIGNED,
  tarjeta BLOB,
  FOREIGN KEY (cliente)
    REFERENCES clientes(cliente_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
);

CREATE TABLE servicios(
  servicio_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  cliente INT UNSIGNED,
```

```
  tarjeta INT UNSIGNED,
  suscripcion INT UNSIGNED,
  FOREIGN KEY(cliente)
    REFERENCES clientes(cliente_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE,
  FOREIGN KEY(tarjeta)
    REFERENCES tarjetas(tarjeta_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE,
  FOREIGN KEY(suscripcion)
    REFERENCES suscripciones(suscripcion_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
);
```

Ahora el código del procedimiento:

```
CREATE PROCEDURE sp_asignar_servicio(
  IN i_suscripcion INT UNSIGNED,
  IN i_nombre VARCHAR(30),
  IN i_correo VARCHAR(50),
  IN i_tarjeta VARCHAR(16),
  OUT o_respuesta VARCHAR(50)
)

  BEGIN

    DECLARE existe_correo INT DEFAULT 0;
    DECLARE cliente_id INT DEFAULT 0;
    DECLARE tarjeta_id INT DEFAULT 0;

    START TRANSACTION;

      SELECT COUNT(*) INTO existe_correo
        FROM clientes
        WHERE correo = i_correo;

      IF existe_correo <> 0 THEN

        SELECT 'Tu correo ya ha sido registrado' INTO o_respuesta;

      ELSE

        INSERT INTO clientes VALUES (0, i_nombre, i_correo);
        SELECT LAST_INSERT_ID() INTO cliente_id;

        INSERT INTO tarjetas
          VALUES (0, cliente_id, AES_ENCRYPT(i_tarjeta, cliente_id));
        SELECT LAST_INSERT_ID() INTO tarjeta_id;

        INSERT INTO servicios VALUES (0, cliente_id, tarjeta_id, i_suscripcion);

        SELECT 'Servicio asignado con éxito' INTO o_respuesta;

      END IF;

    COMMIT;

  END //

DELIMITER ;
```

Finalmente lo ejecutamos y vemos el resultado de la variable de respuesta y en las correspondientes tablas la inserción de datos:

```
CALL sp_asignar_servicio(2, 'Kenai', 'kenai@gmail.com', '1234567890123490', @res);
SELECT @res;

SELECT * FROM clientes;
SELECT * FROM tarjetas;
SELECT * FROM servicios;
```

🔼 Regresar

Disparadores

Un disparador o ***Trigger*** es un objeto que se utiliza para ejecutar automáticamente una acción en respuesta a ciertos eventos en una base de datos, como *INSERT*, *UPDATE* o *DELETE* en una tabla específica.

Los disparadores se pueden utilizar para asegurarse de que ciertas acciones se realicen automáticamente después de que se realice un cambio en una tabla, o para evitar que se realicen ciertas acciones.

Por ejemplo, un disparador se puede utilizar para actualizar automáticamente una tabla de resumen después de que se realice un cambio en una tabla de detalles, o para evitar que se elimine un registro importante de una tabla.

Sintaxis

```
DELIMITER //
CREATE TRIGGER nombre_disparador
  [BEFORE | AFTER] [INSERT | UPDATE | DELETE]
  ON nombre_tabla
  FOR EACH ROW

  BEGIN
    Código del Disparador
  END //

DELIMITER ;
```

Eliminar un disparador y mostrar los disparadores de una base de datos:

```
DROP TRIGGER nombre_disparador;

SHOW TRIGGERS FROM base_de_datos;
```

Ejemplo

Para el ejemplo de este disparador, seguiremos usando el código de ejemplo de los procedimeintos almacenados, primero creamos una tabla donde se almacene el resultado de nuestro disparador:

```
CREATE TABLE actividad_clientes(
  ac_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
```

```
  cliente INT UNSIGNED,
  fecha DATETIME,
  FOREIGN KEY (cliente)
    REFERENCES clientes(cliente_id)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
);
```

Ahora creamos nuestro disparador:

```
CREATE TRIGGER tg_actividad_clientes
  AFTER INSERT
  ON clientes
  FOR EACH ROW

  BEGIN

    INSERT INTO actividad_clientes VALUES (0, NEW.cliente_id, NOW());

  END //

DELIMITER ;
```

Finalmente ejecutamos nuevamente el ejemplo del procedimiento almacenado y en cuanto se haga el *INSERT* a la tabla "*clientes*", el disparador se lanzará automáticamente.

```
CALL sp_asignar_servicio(2, 'Kenai', 'kenai@gmail.com', '1234567890123490', @res);
SELECT @res;
```

Para comprobar que el disparador se ejecuto revisamos la tabla "*actividad_clientes*":

```
SELECT * FROM actividad_clientes;
```

🔼 Regresar

Aprende más

Si estás interesado en aprender más sobre bases de datos, no te pierdas mis cursos totalmente **gratuitos** en mi canal de *YouTube*.

¡¡¡Accede ya!!!

Ver Cursos

Figma

Bases de Datos Edited 2 months ago

Regresar