**﻿﻿Módulo 6**

**Bases de datos**

**Tipos de bases de datos**

* Bases de Datos Relacionales (RDBMS)
* Bases de Datos NoSQL
* Bases de Datos en Memoria (In-Memory Databases)
* Bases de Datos de Grafos
* Bases de Datos de Columnas (Columnar Databases)
* Bases de Datos Distribuidas

**Bases de Datos Relacionales -** RDBMS

Modelo de Datos Relacional

Lenguaje SQL (Structured Query Language):

Transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad):

Normalización

Integridad Referencial

Escalabilidad Vertical y Horizontal

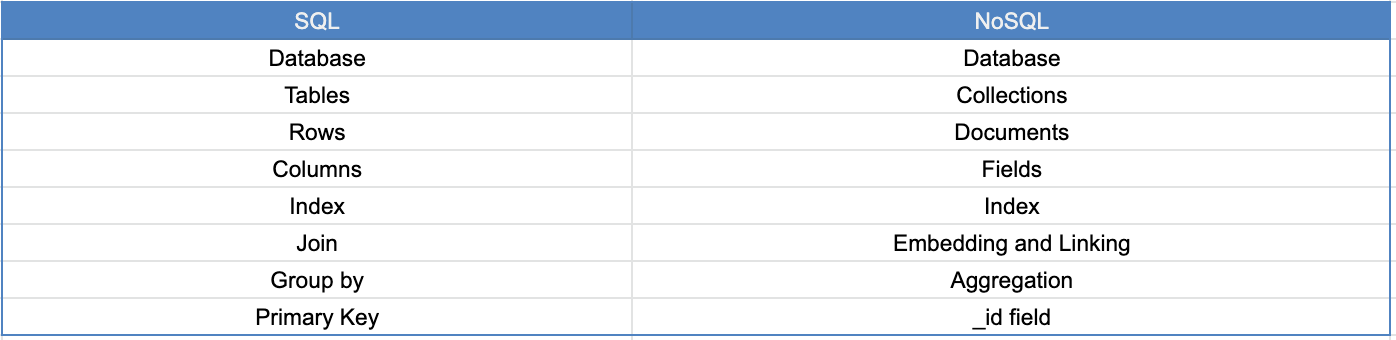
|  |  |
| --- | --- |
| **Lenguaje SQL (Structured Query Language)** | |
| * Primeras tareas que se pueden realizar con SQL * Definir esquemas de base de datos * Inserción de datos * Actualización y eliminación de datos * Recuperación de datos * Creación y gestión de índices |  |
| * Creación de una Base de Datos * Creación de Tablas y Columnas * Inserción de Datos * Consulta de Datos * Actualización y Eliminación de Datos |
| **Sintaxis de SQL** Algunos de los DBMS populares   * ventajas * desventajas |  |
| * Grupos de comandos * Tipos de datos mas comunes * Comando SQL * WHERE * Tipos de Operadores Lógicos * Tipo de JOIN * CRUD * SQLite3 |

.

|  |  |
| --- | --- |
| **Características de SQLite3** | |
| Primeros pasos   * import sqlite3 * connect * Cursor * execute * conexión.commit * conexion.close() * conexion .rollback() |  |

|  |  |
| --- | --- |
| Métodos para volcar datos desde DDBB a Python | Join |
| * fetchall() * fetchone() * fetchmany () | * JOIN * INNER JOIN * LEFT JOIN * RIGHT JOIN * FULL OUTER * SELF JOIN |

**Bases de datos**



**Tipos de bases de datos**

Existen varios tipos de bases de datos, cada una diseñada para un propósito específico y con sus propias características. Aquí te proporciono una breve descripción de los tipos más comunes de bases de datos y para qué se utilizan:

* Bases de datos relacionales.
* Bases de datos NoSQL.
* Bases de datos en memoria.
* Bases de datos de grafos.
* Bases de datos de objetos.

|  |
| --- |
| **Bases de Datos Relacionales (RDBMS):**  MariaDB, MySQL, PostgreSQL(ORDBMS), Oracle, SQL Server, etc.   * Se utilizan tablas con filas y columnas para almacenar datos. * Utilizan el lenguaje SQL (Structured Query Language) para realizar consultas y manipular datos. * Son ideales para aplicaciones que requieren estructura y relaciones complejas entre los datos, como sistemas de gestión de inventario, sistemas de contabilidad y aplicaciones empresariales. |
| **Bases de Datos NoSQL:**  MongoDB, Cassandra, Redis, Neo4j.   * Las bases de datos NoSQL son una alternativa a las bases de datos relacionales. * No utilizan tablas y claves como las bases de datos relacionales, sino que utilizan una variedad de estructuras de datos, como documentos, gráficos y pares clave-valor. * Las bases de datos NoSQL son muy escalables y ofrecen un rendimiento excelente para grandes conjuntos de datos no estructurados, como datos de redes sociales o de Internet de las cosas (IoT). * Se utilizan en aplicaciones web y móviles que manejan grandes volúmenes de datos no estructurados o semiestructurados, como redes sociales, juegos en línea y análisis de registros. |
| **Bases de Datos en Memoria (In-Memory Databases):**  Redis, Memcached, Apache Cassandra.   * Almacenan datos en la memoria RAM en lugar de en el disco, lo que las hace extremadamente rápidas para operaciones de lectura y escritura. Suelen utilizarse para caché y almacenamiento temporal. * Son útiles en aplicaciones que requieren acceso rápido a datos, como aplicaciones en tiempo real, sistemas de recomendación y procesamiento de transacciones. |
| **Bases de Datos de Grafos:**  Neo4j, Amazon Neptune.   * Diseñadas para almacenar y consultar datos que tienen estructuras de grafo, como redes sociales o relaciones de datos complejas. * Se utilizan en aplicaciones que necesitan modelar y consultar relaciones complejas, como análisis de redes sociales, recomendaciones de productos y sistemas de recomendación. |
| **Bases de Datos de Columnas (Columnar Databases):**  Apache Cassandra, Amazon Redshift, Google BigQuery.   * Almacenan datos en columnas en lugar de filas, lo que las hace eficientes para consultas analíticas y agregaciones en grandes conjuntos de datos. * Son ideales para aplicaciones de análisis de datos, inteligencia empresarial y almacenamiento de registros. |
| **Bases de Datos Distribuidas:**  Apache Hadoop, Apache Kafka, CockroachDB.   * Diseñadas para operar en clústeres de servidores distribuidos para garantizar la disponibilidad, escalabilidad y redundancia. * Se utilizan en aplicaciones que requieren alta disponibilidad y escalabilidad, como sistemas de seguimiento en tiempo real, procesamiento de grandes volúmenes de datos y almacenamiento en la nube. |

La elección de la base de datos adecuada depende de los requisitos y características específicas de tu aplicación. Debes considerar factores como la estructura de tus datos, el rendimiento, la escalabilidad, la consistencia y la complejidad de las consultas antes de seleccionar la base de datos que mejor se adapte a tus necesidades.

**Bases de Datos Relacionales**

RDBMS- Relational Database Management Systems

Modelo de Datos Relacional

En las bases de datos relacionales, los datos se organizan en tablas que consisten en filas y columnas.

Cada tabla representa una entidad o relación, y cada fila representa una instancia individual de esa entidad.

Las tablas están relacionadas entre sí mediante claves primarias (primary keys) y claves externas (foreign keys), lo que permite establecer relaciones y mantener la integridad de los datos.

La llave foránea es un campo que apunta a la llave primaria de otra tabla

Lenguaje SQL (Structured Query Language):

Las bases de datos relacionales utilizan SQL para definir la estructura de las tablas, realizar consultas, modificar datos y administrar la base de datos.

SQL es un lenguaje poderoso que permite realizar operaciones como SELECT (para recuperar datos), INSERT (para agregar nuevos datos), UPDATE (para modificar datos existentes) y DELETE (para eliminar datos) de manera eficiente.

Transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad):

**Atomicidad:** Las transacciones se realizan en su totalidad o no se realizan en absoluto.

**Consistencia:** La base de datos debe estar en un estado consistente antes y después de una transacción.

**Aislamiento:** Las transacciones deben ser independientes entre sí y no deben interferir.

**Durabilidad:** Los cambios realizados en una transacción deben ser permanentes incluso después de un fallo del sistema.

Normalización

La normalización es un proceso utilizado en las bases de datos relacionales para organizar los datos de manera eficiente y reducir la redundancia.

Se logra dividiendo las tablas en varias tablas más pequeñas y relacionándolas entre sí mediante claves primarias y externas. Esto evita la duplicación innecesaria de datos y mantiene la integridad de los datos.

Integridad Referencial

Las bases de datos relacionales mantienen la integridad referencial mediante el uso de claves foráneas. Esto garantiza que no se puedan insertar datos que violen las relaciones definidas entre las tablas.

Escalabilidad Vertical y Horizontal

Las bases de datos relacionales pueden escalar verticalmente (añadiendo más recursos a una sola máquina) o horizontalmente (dividiendo los datos entre múltiples servidores) para satisfacer las demandas de aplicaciones de diferentes tamaños.

Las bases de datos relacionales se utilizan comúnmente en aplicaciones empresariales, sistemas de gestión de contenido, sistemas de seguimiento de pedidos, sistemas de contabilidad y cualquier aplicación donde se requiere un alto grado de estructura y relaciones complejas entre los datos.

Las bases de datos relacionales ofrecen una estructura sólida y coherente para almacenar y gestionar datos. Su capacidad para garantizar la integridad de los datos y admitir consultas SQL complejas las convierte en una elección sólida para una amplia variedad de aplicaciones comerciales y de misión crítica. Sin embargo, también es importante considerar las necesidades de escalabilidad y el rendimiento al seleccionar una base de datos para un proyecto específico.

**Lenguaje SQL**

**SQL (Structured Query Language)** es un lenguaje de programación específico del dominio utilizado para gestionar y manipular bases de datos relacionales. Fue desarrollado originalmente por IBM en la década de 1970 y se ha convertido en un estándar de facto en la gestión de bases de datos relacionales. SQL se utiliza para realizar una amplia variedad de tareas relacionadas con bases de datos, como la creación, modificación, recuperación y eliminación de datos.

SQL es un lenguaje de consulta estructurado que permite a los usuarios interactuar con bases de datos relacionales. A través de **sentencias SQL**, los usuarios pueden definir estructuras de datos, manipular datos almacenados en la base de datos y realizar consultas complejas para recuperar información específica.

La comprensión de SQL es esencial para cualquier persona que trabaje con bases de datos, ya que es una herramienta fundamental en el mundo de la gestión de datos.

|  |
| --- |
| SQL funciona mediante la ejecución de comandos y consultas en una base de datos. Estos pueden ser proporcionados desde Python y el resultado se recibirá en estructuras Python |

·

|  |
| --- |
| **Primeras tareas que se pueden realizar con SQL** |
| **Definir esquemas de base de datos**  SQL permite definir la estructura de la base de datos, incluyendo tablas, columnas, tipos de datos, claves primarias y restricciones de integridad referencial. |
| **Inserción de datos**  Puedes utilizar SQL para agregar datos a las tablas de la base de datos utilizando la sentencia `INSERT`. |
| **Actualización y eliminación de datos**  SQL proporciona las sentencias `UPDATE` y `DELETE` para modificar y eliminar datos existentes en la base de datos. |
| **Recuperación de datos**  La sentencia `SELECT` se utiliza para recuperar datos de la base de datos. Puedes especificar condiciones, filtrar datos y ordenar resultados según tus necesidades. |
| **Creación y gestión de índices**  SQL permite crear índices en columnas específicas para mejorar el rendimiento de las consultas. |

· ﻿

|  |  |
| --- | --- |
| Creación de una Base de Datos | Para empezar, necesitas tener una base de datos que puedas usar. Puedes crear una nueva base de datos en tu sistema de gestión de bases de datos elegido utilizando la instrucción "CREATE DATABASE".(En sqlite3 es un archivo, si existe lo abre, sino al API lo crea automáticamente sin necesidad de create) |
| Creación de Tablas y Columnas | Luego, debes crear tablas para almacenar información en la base de datos. Esto se logra mediante la instrucción "CREATE TABLE". Debes definir los campos (columnas) de la tabla y especificar su tipo de datos (texto, número, fecha, etc.). |
| Inserción de Datos | Una vez que hayas creado las tablas y las columnas, puedes introducir datos y registros en la base de datos. Utiliza la instrucción "INSERT INTO", especificando la tabla y las columnas en las que deseas insertar los datos. |
| Consulta de Datos | La parte central de SQL es la consulta de datos. Para ello, emplea la instrucción "SELECT". Indica las tablas y las columnas que deseas consultar. También puedes aplicar condiciones utilizando "WHERE" para filtrar los datos que necesitas. |
| Actualización y Eliminación de Datos | Por último, puedes modificar y eliminar datos utilizando las instrucciones "UPDATE" y "DELETE". Estas operaciones te permiten mantener la integridad y precisión de la información almacenada en la base de datos. |

### ·

|  |
| --- |
| **Sintaxis de SQL** |
| SELECT: Se utiliza para recuperar datos de una tabla.  Por ejemplo: SELECT nombre, edad FROM clientes WHERE ciudad = 'Nueva York'; |
| INSERT: Agrega nuevos registros a una tabla.  Por ejemplo: INSERT INTO productos (nombre, precio) VALUES ('Portátil', 1000); |
| UPDATE: Actualiza registros existentes en una tabla.  Por ejemplo: UPDATE empleados SET salario = salario + 500 WHERE departamento = 'Ventas'; |
| DELETE: Elimina registros de una tabla.  Por ejemplo: DELETE FROM pedidos WHERE fecha < '2023-01-01'; |
| CREATE TABLE: Crea una nueva tabla en la base de datos.  Por ejemplo: CREATE TABLE libros (id INT PRIMARY KEY, titulo TEXT, autor TEXT); |
| ALTER TABLE: Modifica la estructura de una tabla existente.  Por ejemplo: ALTER TABLE clientes ADD COLUMN telefono VARCHAR(20); |
| CREATE DATABASE: Crea una nueva base de datos.  Por ejemplo: CREATE DATABASE mi\_nueva\_db; |

La seguridad es una consideración importante en SQL. Para prevenir la inyección SQL y otros ataques, se deben utilizar consultas parametrizadas y procedimientos almacenados en lugar de concatenar valores directamente en las consultas SQL.

Algunos de los DBMS populares

SQL es un lenguaje estándar, pero diferentes sistemas de gestión de bases de datos (DBMS) lo implementan de manera ligeramente diferente.

Cada uno de estos DBMS puede tener características y extensiones específicas, pero todos siguen el estándar SQL en su núcleo.

|  |  |
| --- | --- |
| **Bases de datos** | **Descripción** |
| **MariaDB** | Creada en 2009 por Michael "Monty" Widenius, uno de los creadores originales de MySQL, en MySQL AB, una empresa sueca que fue adquirida por Sun Microsystems en 2008. Luego, en 2010, Sun Microsystems fue adquirida por Oracle Corporation. Para mantenerla fuera de las fauces de Oracle, se genero este folk en código abierto |
| ventajas | Es un proyecto de código abierto  Altamente compatibilidad con MySQL  Alto Rendimiento como el uso del motor de almacenamiento InnoDB, la gestión de subconsultas y la optimización de consultas.  Es escalable y puede gestionar grandes volúmenes de datos y cargas de trabajo de alto rendimiento.  Ofrece una variedad de motores de almacenamiento, InnoDB (default), Aria, TokuDB y más, para las diferentes necesidades. |
| desventajas | MariaDB es compatible en gran medida con MySQL, puede haber algunas diferencias y características adicionales que aun no están documentadas.  Menos Herramientas y Compatibilidad de Terceros(en relacion a MySQL).  Baja cuota de mercado en grandes empresas |
| **MySQL** | Fue lanzada en 1995, es una base de datos SQL de código abierto que ganó popularidad por su facilidad de uso y rendimiento. |
| ventajas | Es muy fácil de instalar y utilizar. Ofrece opciones de alta disponibilidad y escalabilidad. También cuenta con una gran comunidad de desarrolladores y una gran cantidad de recursos en línea. |
| desventajas | Las actualizaciones a veces pueden ser complicadas, y a pesar de ser de código abierto, algunas funciones avanzadas pueden requerir una licencia paga. |
| **Oracle** | Oracle Database es una base de datos SQL desarrollada por la empresa Oracle Corporation en 1977. Es especialista en grandes cantidades de datos y altas cargas de trabajo. |
| ventajas | Es muy escalable y ofrece grandes capacidades de seguridad, incluyendo opciones de copia de seguridad y recuperación ante desastres. También cuenta con soporte técnico de nivel empresarial. |
| desventajas | Es uno de los sistemas de base de datos más costosos del mercado. Muchas de las funciones avanzadas son difíciles de manejar para los principiantes. |
| **SQL Server** | Fue lanzado por Microsoft en 1989 y es el sistema de base de datos de la oferta de Microsoft. Está disponible en ediciones Express, Standard y Enterprise. |
| ventajas | Ofrece una integración nativa con otras herramientas de Microsoft como Excel, SharePoint, y Windows Server. También cuenta con opciones de alta disponibilidad y escalabilidad. |
| desventajas | La mayoría de las funciones avanzadas requieren una licencia de pago. No es nativa para otras plataformas que no sean Microsoft, lo que puede ser una limitación si se desea portabilidad. |
| **PostgreSQL** | Fue lanzado en 1989 como un proyecto universitario y se ha ido desarrollando desde entonces. Es una base de datos SQL de código abierto.  Es una ORDBMS orientado a objetos complejos |
| ventajas | Es muy segura y cuenta con opciones de escalabilidad y alta disponibilidad. También es conocida por su alta compatibilidad con los estándares SQL. |
| desventajas | La instalación y configuración inicial pueden ser complicadas. Algunas funciones avanzadas pueden requerir de trabajo adicional por parte del desarrollador. |
| **SQLite** | Fue lanzado en 2000 como una alternativa a MySQL. Es una base de datos SQL ligera y fácil de utilizar. Es muy popular en el desarrollo de aplicaciones móviles y para navegadores web. |
| ventajas | Es muy ligero y fácil de utilizar. No requiere instalación. También cuenta con soporte para un gran número de lenguajes de programación. |
| desventajas | SQLite no está diseñado para manejar grandes cantidades de datos. Puede ser limitado en cuanto a capacidades de escalabilidad y alta disponibilidad. |

·

Esta tabla proporciona una descripción más clara de los tipos de datos comunes y su compatibilidad con tres sistemas de gestión de bases de datos populares: SQLite3, MySQL y PostgreSQL. Los marcadores "Sí" indican que el tipo de dato es compatible con el sistema de gestión de bases de datos correspondiente.

**Grupo de comandos SQL**

### **Tipos de datos mas comunes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tipo de dato** | **Descripción** | **SQLite3** | **MySQL** | **Postgre**  **SQL** |
| VARCHAR(n) | Cadena de texto de longitud variable | si | si | si |
| TEXT | Cadena de texto de longitud variable | si | si | si |
| INTEGER | Número entero | si | si | si |
| FLOAT | Número decimal de precisión simple | si | si | si |
| DOUBLE | Número decimal de doble precisión | no | si | si |
| DECIMAL(p,s) | Número decimal de precisión variable | no | si | si |
| BOOLEAN | Valor booleano verdadero o falso | si | si | si |
| DATE | Fecha en formato 'AAAA-MM-DD' | si | si | si |
| TIME | Hora en formato 'HH:MM:SS' | si | si | si |
| DATETIME | Fecha y hora en formato 'AAAA-MM-DD HH:MM:SS | si | si | si |
| BLOB | Datos binarios de longitud variable | si | si | si |
| JSON | Almacena datos estructurados en formato JSON | si | si | si |
| ARRAY | Almacena datos de una columna en formato de arreglo | no | si(1) | si |
| ENUM | Almacena un conjunto de valores predeterminados | no | si(2) | si |

Notas:

1. En MySQL, el almacenamiento de datos tipo ARRAY se logra utilizando las columnas de tipo TEXT o VARCHAR y separando los valores por comas o algún otro carácter.

2. En MySQL, el almacenamiento de datos tipo ENUM se logra utilizando una lista de valores predeterminados separados por comas y utilizando una columna de tipo VARCHAR.

Es importante destacar que, aunque los tipos de datos básicos son similares en la mayoría de las bases de datos, pueden haber diferencias en la precisión, el almacenamiento, la longitud y otras propiedades. Además, cada motor de búsqueda puede tener diferentes tipos de datos avanzados y personalizados.

Comando SQL

|  |  |  |
| --- | --- | --- |
| **Comando SQL** | **Descripción** | **Ejemplo** |
| CREATE DATABASE | Crea una nueva base de datos. | CREATE DATABASE mi\_bd; |
| CREATE TABLE | Crea una nueva tabla en una base de datos. | CREATE TABLE clientes (id INT, nombre VARCHAR(50), email VARCHAR(50)); |
| ALTER TABLE | Modifica la estructura de una tabla existente. | ALTER TABLE clientes ADD telefono VARCHAR(20); |
| DROP TABLE | Elimina una tabla. | DROP TABLE clientes; |
| INSERT INTO | Agrega un nuevo registro a una tabla. | INSERT INTO clientes (id, nombre, email, telefono) VALUES (1, 'Ariel García', 'Ariel@example.com', '12345678'); |
| SELECT | Consulta información de una base de datos. | SELECT id, nombre, email FROM clientes WHERE id = 1; |
| FROM | Indica la tabla de la cual se van a seleccionar los datos. | SELECT id, nombre, apellido FROM clientes; |
| WHERE | Filtra los datos que cumplen cierta condición. | SELECT \* FROM clientes WHERE edad >= 18; |
| ORDER BY | Ordena los resultados según una determinada columna. | SELECT \* FROM clientes ORDER BY apellido ASC; |
| GROUP BY | Agrupa los resultados según una determinada columna. | SELECT departamento, COUNT(\*) FROM empleados GROUP BY departamento; |
| INSERT INTO | Inserta nuevos registros en una tabla. | INSERT INTO clientes (nombre, apellido, edad) VALUES ('Ariel', 'García', 25); |
| UPDATE | Actualiza uno o varios registros de una tabla. | UPDATE clientes SET telefono = '98765432' WHERE id = 1; |
| DELETE | Elimina uno o varios registros de una tabla. | DELETE FROM clientes WHERE id = 1; |
| DELETE FROM | Elimina registros existentes en una tabla. | DELETE FROM clientes WHERE id = 1; |
| JOIN | Combina información de múltiples tablas en una sola consulta. | SELECT c.id, c.nombre, p.nombre AS producto FROM clientes c JOIN pedidos p ON c.id = p.id\_cliente; |
| CREATE INDEX | Crea un índice en una tabla para acelerar la búsqueda. | CREATE INDEX idx\_nombre ON clientes (nombre); |
| DROP INDEX | Elimina un índice de una tabla. | DROP INDEX idx\_nombre; |
| GRANT | Otorga permisos a usuarios o roles. | GRANT SELECT ON clientes TO usuario; |
| REVOKE | Revoca permisos previamente otorgados. | REVOKE SELECT ON clientes FROM usuario; |
| COMMIT | Confirma una transacción en curso. | COMMIT; |
| ROLLBACK | Deshace los cambios en una transacción en curso. | ROLLBACK; |

La tabla siguiente proporciona una descripción más amplia de varios comandos SQL, sus funciones y ejemplos de cómo se utilizan. Puedes utilizar estos comandos para crear, modificar, consultar y administrar bases de datos y tablas en sistemas de gestión de bases de datos relacionales. Además de los comandos SQL básicos, se incluyen comandos para gestionar permisos, transacciones y crear índices para optimizar el rendimiento de las consultas.

Comando SQL

|  |  |
| --- | --- |
| **Comando SQL** | **Descripción** |
| CREATE DATABASE | Crea una nueva base de datos. |
| CREATE DATABASE mi\_bd; | |
| CREATE TABLE | Crea una nueva tabla en una base de datos. |
| CREATE TABLE clientes (id INT, name VARCHAR(50), email VARCHAR(50)); | |
| ALTER TABLE | Modifica la estructura de una tabla existente. |
| ALTER TABLE clientes ADD telefono VARCHAR(20); | |
| DROP TABLE | Elimina una tabla. |
| DROP TABLE clientes; | |
| INSERT INTO | Agrega un nuevo registro a una tabla. |
| INSERT INTO clientes (id, name, email, telefono) VALUES (1, 'Ariel', 'AGT@mail.com','888'); | |
| SELECT | Consulta información de una base de datos. |
| SELECT id, name, email FROM clientes WHERE id = 1; | |
| FROM | Indica la tabla de la cual se van a seleccionar los datos. |
| SELECT id, name, apellido FROM clientes; | |
| WHERE | Filtra los datos que cumplen cierta condición. |
| SELECT \* FROM clientes WHERE edad >= 18; | |
| ORDER BY | Ordena los resultados según una determinada columna. |
| SELECT \* FROM clientes ORDER BY apellido ASC; | |
| GROUP BY | Agrupa los resultados según una determinada columna. |
| SELECT departamento, COUNT(\*) FROM empleados GROUP BY departamento; | |
| INSERT INTO | Inserta nuevos registros en una tabla. |
| INSERT INTO clientes (name, apellido, edad) VALUES ('Ariel', 'García', 25); | |
| UPDATE | Actualiza uno o varios registros de una tabla. |
| UPDATE clientes SET telefono = '15 4475 4637' WHERE id = 1; | |
| DELETE | Elimina uno o varios registros de una tabla. |
| DELETE FROM clientes WHERE id = 1; | |
| DELETE FROM | Elimina registros existentes en una tabla. |
| DELETE FROM clientes WHERE id = 1; | |
| JOIN | Combina información de múltiples tablas en una sola consulta. |
| SELECT c.id, c.name, p.name AS producto FROM clientes c JOIN pedidos p ON c.id = p.id\_cliente; | |
| CREATE INDEX | Crea un índice en una tabla para acelerar la búsqueda. |
| CREATE INDEX idx\_name ON clientes (name); | |
| DROP INDEX | Elimina un índice de una tabla. |
| DROP INDEX idx\_name; | |
| GRANT | Otorga permisos a usuarios o roles. |
| GRANT SELECT ON clientes TO usuario; | |
| REVOKE | Revoca permisos previamente otorgados. |
| REVOKE SELECT ON clientes FROM usuario; | |
| COMMIT | Confirma una transacción en curso. |
| COMMIT; | |
| ROLLBACK | Deshace los cambios en una transacción en curso. |
| ROLLBACK; | |

·

**WHERE**

La siguiente tabla proporciona una referencia completa de las opciones de WHERE y los operadores lógicos en SQL, junto con ejemplos que ilustran su uso en consultas SQL. Estos comandos y operadores son fundamentales para filtrar y seleccionar datos de bases de datos de manera específica y efectiva.

|  |  |
| --- | --- |
| **WHERE** | **Descripción** |
| **=** | Compara la igualdad entre dos valores. |
| SELECT \* FROM clientes WHERE apellido = 'González'; | |
| <> o != | Compara la desigualdad entre dos valores. |
| SELECT \* FROM clientes WHERE edad <> 18; | |
| < | Compara si un valor es menor que otro. |
| SELECT \* FROM clientes WHERE edad < 25; | |
| > | Compara si un valor es mayor que otro. |
| SELECT \* FROM clientes WHERE edad > 25; | |
| <= | Compara si un valor es menor o igual que otro. |
| SELECT \* FROM clientes WHERE edad <= 25; | |
| >= | Compara si un valor es mayor o igual que otro. |
| SELECT \* FROM clientes WHERE edad >= 25; | |
| BETWEEN | Selecciona valores dentro de un rango dado. |
| SELECT \* FROM clientes WHERE edad BETWEEN 18 AND 30; | |
| LIKE | Selecciona valores que coinciden con un patrón de texto. |
| SELECT \* FROM clientes WHERE apellido LIKE 'G%'; | |
| IN | Selecciona valores que se encuentran en un conjunto de opciones. |
| SELECT \* FROM clientes WHERE apellido IN ('González', 'García'); | |
| IS NULL / | Compara si un valor es nulo. |
| SELECT \* FROM clientes WHERE telefono IS NULL; | |
| IS NOT NULL | Compara si un valor es no nulo. |
| SELECT \* FROM clientes WHERE telefono IS NOT NULL; | |

**Tipos de Operadores Lógicos:**

Es importante utilizar correctamente los operadores lógicos para obtener los resultados esperados en las consultas, ya que un uso inadecuado de ellos puede generar resultados inesperados. Cada opción de operadores lógicos tiene su propia sintaxis y, dependiendo de la base de datos utilizada, pueden haber diferencias en la forma en que se usan

|  |  |
| --- | --- |
| **Operadores**  **lógicos** | **Descripción** |
| **AND** | Devuelve verdadero si ambos operadores son verdaderos. |
| SELECT \* FROM clientes WHERE edad > 18 AND edad < 30; | |
| **OR** | Devuelve verdadero si alguno de los operadores es verdadero. |
| SELECT \* FROM clientes WHERE apellido = 'González' OR apellido = 'García'; | |
| **NOT** | Invierte el resultado de la expresión que le sigue. |
| SELECT \* FROM clientes WHERE NOT edad > 18; | |
| **IN** | Devuelve verdadero si una expresión coincide con cualquiera de los valores de una lista. |
| SELECT \* FROM clientes WHERE apellido IN ('González', 'García'); | |
| **BETWEEN** | Devuelve verdadero si una expresión se encuentra dentro de un rango determinado. |
| SELECT \* FROM clientes WHERE edad BETWEEN 18 AND 30; | |
| **LIKE** | Devuelve verdadero si una expresión coincide con una cadena de caracteres determinada. |
| SELECT \* FROM clientes WHERE apellido LIKE 'G%'; | |

·

Tipo de JOIN

|  |  |
| --- | --- |
| **Tipo de JOIN** | **Descripción** |
| INNER JOIN | Combina filas de dos tablas en función de una |
| condición de igualdad y muestra solo las filas |
| que tienen coincidencias en ambas tablas. |
| SELECT clientes.name, pedidos.prod FROM clientes INNER JOIN pedidos ON clientes.id = pedidos.id\_cliente; | |
| LEFT JOIN | Combina todas las filas de la tabla izquierda |
| con las filas coincidentes de la tabla derecha. |
| Las filas sin coincidencias tendrán valores NULL |
| en las columnas de la tabla derecha. |
| SELECT clientes.name, pedidos.prod FROM clientes LEFT JOIN pedidos ON clientes.id = pedidos.id\_cliente; | |
| RIGHT JOIN | Combina todas las filas de la tabla derecha con |
| las filas coincidentes de la tabla izquierda. |
| Las filas sin coincidencias tendrán valores NULL |
| en las columnas de la tabla izquierda. |
| SELECT clientes.name, pedidos.prod FROM clientes RIGHT JOIN pedidos ON clientes.id = pedidos.id\_cliente; | |
| FULL OUTER  JOIN | Combina todas las filas de ambas tablas y muestra |
| coincidencias donde existan. Las filas sin |
| coincidencias tendrán valores NULL en ambas |
| tablas. |
| SELECT clientes.name, pedidos.prod FROM clientes FULL OUTER JOIN pedidos ON clientes.id = pedidos.id\_cliente; | |
| SELF JOIN | Une una tabla consigo misma, creando una especie |
| de duplicado de la tabla original. Se utiliza |
| para realizar consultas en jerarquías o estructuras |
| de datos jerárquicas. |
| SELECT e1.name, e2.name AS supervisor FROM empleados e1 INNER JOIN empleados e2 ON e1.supervisor\_id = e2.id; | |

CRUD ﻿﻿

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Curso Python** | **Base de  datos** |  | | |
|  | **tablas** |  | |
|  | | **columnas** |  |
|  | | | **registros** |
| **C**reate | CREATE DATABASE | CREATE TABLE | ADD COLUMN | INSERT |
| **R**ead | SHOW DATABASES | SHOW TABLES | SHOW COLUMNS FROM | SELECT--fetchall order |
| **U**pdate |  | ALTER TABLE | ALTER COLUMNS | UPDATE |
| **D**elete | DROP DATABASE | DROP TABLES | DROP COLUMN | DELETE |

## 

SQLite3 es una biblioteca de bases de datos relacional de código abierto y autónoma que se utiliza para gestionar bases de datos ligeras y portátiles. Cuando se trabaja con SQLite3 en Python, generalmente se utiliza el módulo sqlite3 incorporado en Python para interactuar con bases de datos SQLite3 desde tus programas Python.

|  |
| --- |
| **Características de SQLite3** |
| **Sin servidor:** SQLite3 es un motor de bases de datos sin servidor, lo que significa que no requiere un proceso de servidor independiente para gestionar la base de datos. Los datos se almacenan en un solo archivo, lo que lo hace altamente portátil y adecuado para aplicaciones integradas y de escritorio. |
| **Multiplataforma:** SQLite se puede utilizar en sistemas operativos como Windows, macOS, Linux y muchos sistemas operativos de dispositivos móviles. |
| **Portátil:** SQLite es una biblioteca escrita completamente en C, que hace que sea fácilmente portátil a diferentes plataformas de sistema operativo. Sus archivos permiten ser llevados a otros equipos. |
| **Ligero:** SQLite3 es una base de datos ligera y de bajo consumo de recursos. Esto lo hace ideal para dispositivos con recursos limitados como móviles, sistemas embebidos, servidores de bajo costo y otros entornos en los que la eficiencia y la eficacia son importantes para un rendimiento eficiente |
| **Transacciones ACID:** Es liviano, multi-usuario, admite transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), lo que garantiza la integridad de los datos incluso en situaciones de fallo y alto rendimiento. |
| **Compatible con SQL estándar:** SQLite admite el lenguaje de consulta estructurado (SQL) y cumple con los estándares SQL-92 y SQL-99. |
| **SQL Completo:** SQLite3 admite una gran parte del estándar SQL, lo que facilita la escritura de consultas SQL para administrar los datos en la base de datos. |
| **Índices y restricciones:** Puedes crear índices en columnas para mejorar el rendimiento de las consultas y aplicar restricciones de integridad referencial para mantener la coherencia de los datos. |
| **Sin configuración:** No es necesario configurar servidores o ajustar configuraciones para comenzar a usar SQLite3. Puedes crear y utilizar bases de datos SQLite3 directamente en tu aplicación. |
| **Patentes:** al ser de código abierto y no tener cargos de licencia, SQLite es muy económico de usar y mantener. |

·

|  |
| --- |
| **Características de la Integración de SQLite3 con Python** |
| **Módulo sqlite3:** Python incluye el módulo sqlite3 estándar que permite interactuar con bases de datos SQLite3. Este módulo proporciona una API fácil de usar para crear, consultar y administrar bases de datos SQLite3 desde Python. |
| **Facilidad de uso:** El módulo sqlite3 simplifica la ejecución de comandos SQL y la recuperación de resultados. Puedes crear y administrar bases de datos SQLite3 con unas pocas líneas de código. |
| **Preparación de consultas:** SQLite3 admite consultas preparadas, lo que permite ejecutar consultas SQL de manera eficiente con parámetros variables. |
| **Soporte para transacciones:** Puedes utilizar transacciones en SQLite3 mediante métodos como commit() y rollback(), lo que garantiza la atomicidad de las operaciones. |
| **Seguridad incorporada:** SQLite3 incluye características de seguridad, como el mecanismo de vinculación de parámetros, para proteger contra inyecciones SQL. |
| **Soporte de tipos de datos:** SQLite3 admite tipos de datos comunes como INTEGER, REAL, TEXT y BLOB, lo que facilita la gestión de diversos tipos de información. |
| **Amplia disponibilidad:** SQLite3 se incluye de forma predeterminada en la mayoría de las instalaciones de Python, lo que garantiza su disponibilidad en una amplia variedad de entornos. |

SQLite3 es una base de datos ligera y portátil que se integra de manera efectiva con Python a través del módulo sqlite3. Es adecuada para aplicaciones que requieren una base de datos local y no necesitan un servidor de base de datos completo. La combinación de SQLite3 y Python proporciona una solución eficiente y fácil de usar para el almacenamiento y la manipulación de datos en aplicaciones Python.

**Primeros pasos**

Damos por seguro que ya tenes sqlite3 en tu pc. Te recomendamos instalar dbbrowser para manipularlas por afuera de Python

La librería estándar viene con la instalación básica de Python, no hay que instalarla con pip

|  |
| --- |
| **import sqlite3** |

**connect:** Este comando se utiliza para establecer una conexión con una base de datos SQLite. Toma como argumento la ruta o el nombre del archivo de la base de datos y devuelve un objeto de conexión.

Como SQLite3 no es un servidor, no requiere apuntar a un puerto, usuario, password o host.

|  |
| --- |
| import sqlite3  # Establecer una conexión a una base de datos SQLite (creará el archivo si no existe)  # Por lo que no se usa create database  **conexion = sqlite3.connect('ddbb\_sql3.db')** |

**Cursor:**Una vez que tienes una conexión, necesitas un objeto **cursor** para ejecutar consultas SQL en la base de datos.

El cursor actúa como un puntero a una ubicación en la base de datos. Es un carrier, un objeto que lleva y trae información y comandos o funciones entre Python y SQLite3.

Si la base de datos no proporciona un concepto de cursor directo, el módulo tendrá que emular los cursores utilizando otros medios en la medida que lo requiera esta especificación.

|  |
| --- |
| import sqlite3  conexion = sqlite3.connect('ddbb\_sql3.db') **cursor = conexion.cursor()** |

**·**

|  |
| --- |
| Cualquier consulta u operación que se envie desde Python es tomado por SQLite como orden (si esta bien hecha)  Esta solo es un string para Python, que se puede sacar por consola con print o mandar a SQLite3 por execute. |

**execute:** Este comando se utiliza para ejecutar consultas SQL en la base de datos. Toma un string que contiene la consulta SQL como argumento y la ejecuta a través del cursor.

|  |
| --- |
| import sqlite3  conexion = sqlite3.connect('ddbb\_sql3.db') cursor = conexion.cursor()  query = **'CREATE TABLE IF NOT EXISTS clientes (id INT, nombre TEXT, email TEXT)'**  **cursor.execute(**query**)** |

**conexión.commit:** Después de ejecutar una o varias consultas SQL, es importante confirmar los cambios en la base de datos utilizando el comando commit. Esto guarda permanentemente los cambios realizados en la base de datos que quedan pendientes de la confirmación. Sin commit, los cambios se pueden perder cuando se cierre la conexión. Tenga en cuenta que el commit es a la conexión no al cursor

Hay APIs que permiten una función de confirmación automática, el autocommit, donde se confirma cualquier transacción en la base de datos. Esta confirmación hay que activarla ya que por default es False

Se puede proporcionar un método de interfaz para volver a encenderlo.

Los módulos de base de datos que no admiten transacciones deben implementar este método con funcionalidad nula.

|  |
| --- |
| import sqlite3  conexion = sqlite3.connect('ddbb\_sql3.db') cursor = conexion.cursor()  query = "INSERT INTO clientes (id, nombre, email ) VALUES (99,'Ariel', 'ariel@mail.com'); "  cursor.execute(query)  **conexión.commit()** |

**conexion.close()**

Cierre la conexión ahora (en lugar de cada vez que se llame a.\_\_del\_\_()).

La conexión será inutilizable a partir de este momento; se generará una excepción de error (o subclase) si se intenta alguna operación con la conexión . Lo mismo se aplica a todos los objetos de cursor que intentan usar la conexión. Tenga en cuenta que cerrar una conexión sin confirmar los cambios primero provocará que se realice una reversión implícita.

|  |
| --- |
| import sqlite3  conexion = sqlite3.connect('ddbb\_sql3.db') cursor = conexion.cursor()  query = "INSERT INTO clientes (id, nombre, email ) VALUES (99,'Ariel', 'ariel@mail.com'); "  cursor.execute(query)  conexión.commit()  **conexion.close()** |

**conexion .rollback()**

Este método es opcional ya que no todas las bases de datos brindan soporte para transacciones.

En caso de que una base de datos proporcione transacciones, este método hace que la base de datos retroceda al inicio de cualquier transacción pendiente. Cerrar una conexión sin confirmar los cambios primero provocará que se realice una reversión implícita.

El comando rollback se utiliza en el contexto de transacciones en bases de datos, incluyendo SQLite3, para deshacer cambios no confirmados en la base de datos y restaurar el estado anterior de la misma. La función principal de rollback es cancelar o revertir todas las operaciones realizadas desde el último commit o savepoint en una transacción, lo que significa que la base de datos vuelve a su estado anterior antes de que se ejecutaran las operaciones no confirmadas.

Ejemplo de uso en SQLite3 en Python:

|  |
| --- |
| import sqlite3  conexion = sqlite3.connect('ddbb\_sql3.db') cursor = conexion.cursor()  query = "INSERT INTO clientes (id, nombre, email ) VALUES (99,'Ariel', 'ariel@mail.com'); "  cursor.execute(query)  conexión.commit()  query = "INSERT INTO clientes (id, nombre, email ) VALUES (100,'Pedro', 'pedro@mail.com'); "  cursor.execute(query)  **conexión.rollback ()**  conexion.close() |

·

|  |
| --- |
| import sqlite3  # Establecer una conexión a la base de datos  conexion = sqlite3.connect('ddbb\_sql3.db')  cursor = conexion.cursor()  try:  # Iniciar una transacción  conexion.execute('BEGIN')  # Realizar algunas operaciones en la base de datos  cursor.execute('INSERT INTO clientes (nombre, email) VALUES (?, ?)', ('Ariel', 'juan@mail.com'))  cursor.execute('UPDATE productos SET stock = stock - 1 WHERE id = ?', (101,))  # si esta todo bien. commit  conexion.commit()  except Exception as e:  # si hay un error. rollback  print(f"Error: {str(e)}")  # Se deshacen los cambios no confirmados  conexion.rollback()  finally:  conexion.close() |

En este ejemplo, si algo sale mal durante la transacción (como una excepción), se ejecuta un rollback para deshacer los cambios no confirmados y restaurar la base de datos a su estado anterior. Luego, se cierra la conexión después de la confirmación o el rollback, según corresponda.

|  |
| --- |
| **Cuadro sinóptico**  En una conexión de base de datos SQLite con la librería sqlite3 se encuentran los siguientes atributos y métodos. |
| **isolation\_level**  Obtenga o establezca el nivel de aislamiento predeterminado actual. Ninguno para el modo de confirmación automática o uno de “DIFERIDO”, “INMEDIATO” o “EXCLUSIVO”. Consulte la sección Control de transacciones para obtener una explicación más detallada.  **in\_transaction**  Verdadero si una transacción está activa (hay cambios no confirmados), Falso en caso contrario. Atributo de solo lectura.  **cursor**(factory=Cursor)  El método del cursor acepta una única fábrica de parámetros opcional. Si se proporciona, debe ser un invocable que devuelva una instancia de Cursor o sus subclases.  **commit**().  Este método confirma la transacción actual. Si no llama a este método, todo lo que haya hecho desde la última llamada a commit() no será visible desde otras conexiones de bases de datos. Si se pregunta por qué no ve los datos que escribió en la base de datos, verifique que no olvidó llamar a este método.  **rollback**()  Este método revierte cualquier cambio en la base de datos desde la última llamada a commit().  **close()**  Esto cierra la conexión de la base de datos. Tenga en cuenta que esto no llama automáticamente a commit(). Si simplemente cierra la conexión de su base de datos sin llamar primero a commit(), ¡sus cambios se perderán!  **execute(sql[, parameters])**  Este es un atajo no estándar que crea un objeto de cursor llamando al método cursor(), llama al método ejecutar() del cursor con los parámetros proporcionados y devuelve el cursor.  **executemany(sql[, parameters])**  Este es un atajo no estándar que crea un objeto de cursor llamando al método cursor(), llama al método ejecutarmany() del cursor con los parámetros proporcionados y devuelve el curso  **executescript(sql\_script)**  Este es un atajo no estándar que crea un objeto de cursor llamando al método cursor(), llama al método ejecutarscript() del cursor con el sql\_script dado y devuelve el cursor.  **create\_function(name, num\_params, func, \*, deterministic=False)**  Crea una función definida por el usuario que puede usar más adelante desde sentencias SQL bajo el nombre de la función. num\_params es el número de parámetros que acepta la función (si num\_params es -1, la función puede tomar cualquier número de argumentos) y func es un elemento invocable de Python que se llama como función SQL. Si determinista es verdadero, la función creada se marca como determinista, lo que permite a SQLite realizar optimizaciones adicionales. Este indicador es compatible con SQLite 3.8.3 o superior; se generará NotSupportedError si se usa con versiones anteriores. |

·

|  |
| --- |
| **Manejo de excepciones** |
| **exception sqlite3.Warning**  # Una subclase de excepción.  **exception sqlite3.Error**  # La clase base de las otras excepciones en este módulo. Es una subclase de Excepción.  **exception sqlite3.DatabaseError**  # Excepción generada por errores relacionados con la base de datos.  **exception sqlite3.IntegrityError**  # Se produce una excepción cuando la integridad relacional de la base de datos se ve afectada, p. falla la verificación de clave externa. Es una subclase de DatabaseError.  **exception sqlite3.ProgrammingError**  # Excepción planteada por errores de programación, p. tabla no encontrada o ya existe, error de sintaxis en la declaración SQL, número incorrecto de parámetros especificados, etc. Es una subclase de DatabaseError.  **exception sqlite3.OperationalError**  # Excepción planteada para errores relacionados con el funcionamiento de la base de datos y no necesariamente bajo el control del programador, p. ocurre una desconexión inesperada, no se encuentra el nombre de la fuente de datos, no se pudo procesar una transacción, etc. Es una subclase de DatabaseError.  **exception sqlite3.NotSupportedError**  # Se genera una excepción en caso de que se haya utilizado un método o API de base de datos que no es compatible con la base de datos, p. llamar al método rollback() en una conexión que no admite transacciones o tiene transacciones desactivadas. Es una subclase de DatabaseError. |

### Métodos para volcar datos desde DDBB a Python

Supongamos que tienes una base de datos SQLite3 llamada "ddbb\_sql3.db" con una tabla llamada "clientes" que contiene información sobre clientes con columnas "id," "nombre," y "email."

|  |
| --- |
| import sqlite3  # Establecer una conexión a la base de datos  conexion = sqlite3.connect('ddbb\_sql3.db')  # Crear un cursor  cursor = conexion.cursor()  # Ejecutar una consulta SQL para recuperar datos  query = 'SELECT \* FROM clientes'  cursor.execute(query)  # Recuperar todos los resultados como una lista anidadas de tuplas  resultados = **cursor.fetchall()**  # Iterar a través de los resultados y mostrarlos  for fila in resultados:  print(f'ID: {fila[0]}, Nombre: {fila[1]}, Email: {fila[2]}')  # Cerrar el cursor y la conexión  cursor.close()  conexion.close() |

En el próximo cuadro se describen tres de los métodos más comunes disponibles en la API de Python para trabajar con bases de datos, como SQLite3, PostgreSQL o MySQL:

|  |  |
| --- | --- |
| **fetchall()** | Este método se utiliza para recuperar todos los registros que resultaron de una consulta. Retorna una lista que contiene todas las filas de resultados. Cada fila se representa como una tupla, y todas estas tuplas se agrupan en una lista. Es útil cuando deseas recuperar todos los datos de una consulta. |
| … entrada = cursor.fetchall()  for orden,linea in enumerate(entrada):  print (f"""item {orden}:  id : {linea[0]}  Nombre: {linea[1]}  Email : {linea[2]}""")…. |
| **fetchone()** | Este método devuelve una sola tupla que contiene el siguiente registro que resultó de la consulta. Cada vez que se llama a este método, se obtiene un registro diferente hasta que se hayan alcanzado todos los registros. Es útil para procesar los resultados uno por uno de manera eficiente, especialmente cuando se trabaja con grandes conjuntos de datos. |
| ...fila = cursor.fetchone()  while fila is not None:  print(fila)  fila = cursor.fetchone() while True:  registro = cursor.fetchone()  if registro is None:  print ("no hay mas datos en la selección")... |
| **fetchmany (cant\_lote)** | Este método se utiliza para recuperar una lista que contiene los siguientes "n" registros que resultaron de la consulta, donde "n" se especifica como argumento. Permite recuperar lotes de registros en lugar de todos a la vez, lo que puede ser útil para administrar la memoria de manera eficiente en consultas con un gran número de resultados. |
| lotes = cursor.fetchmany(5) # Recupera de a 5 registros  for fila in lotes:  print(fila) |

Estos métodos son esenciales para procesar y recuperar datos de una base de datos de manera efectiva en Python. La elección del método adecuado depende de tus necesidades específicas y del tamaño de los resultados que esperas de tu consulta.

**Join**

En SQL, los joins se utilizan para combinar datos de dos o más tablas relacionadas en función de una condición específica. Los tipos de join más comunes en SQL son:

|  |  |  |
| --- | --- | --- |
| **Join** | **Descripción** | **SQLite3** |
| INNER JOIN | Devuelve solo las filas que tienen coincidencias en ambas tablas basadas en la condición de unión especificada.  Si no hay coincidencias, esas filas se excluyen del resultado. | **Disponible en SQLite3** |
| SELECT empleados.nombre, departamentos.nombre  FROM empleados  INNER JOIN departamentos ON empleados.departamento\_id = departamentos.id; | | |
| LEFT JOIN  LEFT OUTER JOIN | Devuelve todas las filas de la tabla de la izquierda (tabla principal) y las filas coincidentes de la tabla de la derecha (tabla secundaria).  Si no hay coincidencias en la tabla de la derecha, se rellenarán con valores NULL en el resultado. | **Disponible en SQLite3** |
| SELECT clientes.nombre, pedidos.numero  FROM clientes  LEFT JOIN pedidos ON clientes.id = pedidos.cliente\_id; | | |
| RIGHT JOIN  RIGHT OUTER JOIN | El RIGHT JOIN es similar al LEFT JOIN, pero devuelve todas las filas de la tabla de la derecha (tabla secundaria) y las filas coincidentes de la tabla de la izquierda (tabla principal).  Si no hay coincidencias en la tabla de la izquierda, se rellenarán con valores NULL en el resultado. | **No**  **Disponible en SQLite3** |
| SELECT pedidos.numero, clientes.nombre  FROM pedidos  RIGHT JOIN clientes ON pedidos.cliente\_id = clientes.id; | | |
| FULL JOIN  FULL OUTER JOIN | El FULL JOIN devuelve todas las filas de ambas tablas y combina filas coincidentes basadas en la condición de unión.  Si no hay coincidencias en una de las tablas, se rellenarán con valores NULL en el resultado.  No es compatible con todos los sistemas de gestión de bases de datos, por lo que en algunos casos se puede simular utilizando UNION de dos LEFT JOIN. | **No**  **Disponible**  **en SQLite3** |
| SELECT empleados.nombre, departamentos.nombre  FROM empleados  FULL JOIN departamentos ON empleados.departamento\_id = departamentos.id; | | |
| CROSS JOIN | Devuelve el producto cartesiano de dos tablas, lo que significa que combina cada fila de la primera tabla con cada fila de la segunda tabla.  No se utiliza una condición de unión en este tipo de join.  Puede generar un gran número de filas, por lo que se debe usar con precaución. | **Disponible en SQLite3** |
| SELECT empleados.nombre, departamentos.nombre  FROM empleados  CROSS JOIN departamentos; | | |

Estos son los tipos de join más comunes en SQL. La elección del tipo de join adecuado depende de la relación entre las tablas y de los datos que se desean recuperar en la consulta. Cada tipo de join tiene un propósito específico y se utiliza en función de los resultados deseados.

|  |
| --- |
| **Join**  La siguiente consulta SQL combina datos de las tablas "personas" y "trabajo" mediante su "area" de trabajo o profesión, selecciona información específica de las personas con "edad" mayor igual a 18 años y ordena los resultados según el nombre en orden descendente. Esta consulta proporciona una visión detallada de cómo se pueden unir y filtrar datos de múltiples tablas en una base de datos relacional. |
| sql = """SELECT personas.id,  personas.nombre,  personas.email,  trabajo.horario  FROM personas INNER JOIN trabajo  ON trabajo.area = personas. area  WHERE personas.edad >= 18  ORDER BY personas.nombre DESC"""  Esta consulta SQL se puede desglosar en varias partes, cada una con un propósito específico. A continuación, se describen en detalle cada una de las partes de la consulta:  **SELECT:** Esta cláusula determina qué columnas se van a seleccionar de las tablas especificadas en la consulta. En este caso, estamos seleccionando las siguientes columnas: personas.id, personas.nombre, personas.email, y trabajo.horario.  **FROM:** Indica la(s) tabla(s) de las cuales se obtendrán los datos. En esta consulta, los datos provienen de la tabla "personas".  **INNER JOIN:** Esta es una operación que combina dos tablas en una sola, permitiendo obtener datos de ambas. En esta consulta, estamos uniendo la tabla "personas" con la tabla "trabajo" utilizando como punto de unión lo ingresado en la cláusula ON.  **ON:** La cláusula ON se utiliza para especificar las condiciones de unión entre las tablas. En este caso, estamos relacionando la tabla "personas" con la tabla "trabajo" utilizando la columna "area" en ambas tablas como criterio de unión.  **WHERE:** Esta cláusula se utiliza para filtrar los datos según ciertas condiciones. Aquí, estamos filtrando el resultado para incluir solo personas tengan en el campo .edad un valor mayor o igual (>=) a 18.  **ORDER BY:** Esta cláusula se utiliza para ordenar los resultados en función de una o más columnas. En esta consulta, los resultados se ordenan en orden descendente según la columna "personas.nombre ". |