**﻿﻿Módulo SQLite3**

**Importancia de CRUD y AMB**

Tanto **CRUD** como **AMB** son acrónimos que describen las operaciones básicas que se pueden realizar en un sistema de gestión de bases de datos (SGBD). Estas operaciones son esenciales para cualquier aplicación que maneje datos, ya que permiten la creación, recuperación, actualización y eliminación de información. Comprender estos conceptos es fundamental para diseñar, implementar y mantener bases de datos efectivas.

CRUD (Create, Read, Update, Delete)

1. **Uso en el Desarrollo de Software**:
   * **CRUD** es ampliamente utilizado en el desarrollo de aplicaciones de software, especialmente en el contexto de aplicaciones web y móviles. Los desarrolladores utilizan estas operaciones para definir cómo los usuarios interactúan con los datos de la aplicación.
   * Por ejemplo, en una aplicación de gestión de usuarios, un desarrollador implementará formularios para crear nuevos usuarios (Create), visualizar la lista de usuarios (Read), actualizar información de usuarios existentes (Update) y eliminar usuarios (Delete).
2. **Diseño de Bases de Datos**:
   * Al diseñar una base de datos, los desarrolladores consideran cómo las operaciones CRUD se mapearán a las tablas y relaciones dentro de la base de datos. Cada tabla suele estar diseñada para soportar estas operaciones de manera eficiente.
   * Por ejemplo, se pueden crear índices en las columnas más consultadas para mejorar el rendimiento de las operaciones de lectura (Read).

AMB (Alta, Baja y Modificación)

1. **Uso en la Administración de Bases de Datos**:
   * **AMB** es un término que se usa con frecuencia en la administración y gestión de bases de datos, donde se hace referencia a las operaciones básicas de mantenimiento de datos.
   * En un contexto administrativo, los términos "alta" y "baja" son comunes, especialmente en sistemas donde la gestión de datos implica la adición y eliminación de registros, como en sistemas de recursos humanos o gestión de inventarios.
2. **Sistemas de Gestión de Información**:
   * AMB se relaciona más con la perspectiva de un usuario que gestiona datos. Por ejemplo, un administrador de base de datos (DBA) podría centrarse en realizar altas y bajas de registros de forma continua, manteniendo la integridad y consistencia de la base de datos.
   * A menudo, el enfoque de AMB puede incluir aspectos de control de acceso y validación, asegurando que solo los usuarios autorizados puedan realizar estas operaciones.

Comparación y Contexto

* **Contexto de Uso**:
  + CRUD se asocia más con el desarrollo de software y se encuentra en la jerga técnica de los programadores. En cambio, AMB es un término más práctico y administrativo que se usa en la gestión y operación de bases de datos.
* **Alcance**:
  + CRUD puede extenderse a incluir operaciones adicionales como **search** o **filter**, que son vitales en aplicaciones más complejas, mientras que AMB tiende a centrarse en las operaciones de mantenimiento directo de los registros.

Tanto CRUD como AMB son esenciales para la manipulación efectiva de datos en bases de datos. Aunque ambos términos describen operaciones similares, su uso varía según el contexto. CRUD es más prevalente en el desarrollo y diseño de software, mientras que AMB se utiliza comúnmente en la administración de bases de datos. Comprender ambos conceptos es crucial para cualquier profesional que trabaje con bases de datos, ya que permite abordar tanto el desarrollo como la gestión de datos de manera efectiva.



|  |
| --- |
| SQLite3 |

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.

|  |
| --- |
| CRUD en SQLite con Python para Análisis de Datos |

Introducción

En el mundo del análisis de datos, el manejo eficiente de bases de datos es crucial para almacenar, recuperar y manipular grandes volúmenes de información. Un sistema de gestión de bases de datos relacional (RDBMS) como SQLite es una herramienta versátil y ligera que se adapta bien a este tipo de tareas. SQLite es una base de datos integrada, lo que significa que no necesita configurarse un servidor ni una conexión de red para usarla, lo cual la hace ideal para desarrolladores y analistas que trabajan en sus equipos locales.

Uno de los aspectos más esenciales en el manejo de bases de datos es el conjunto de operaciones CRUD: **Create**, **Read**, **Update**, y **Delete**. Estas operaciones forman el núcleo de cualquier interacción con una base de datos, permitiendo la gestión de los datos de manera eficiente y estructurada. En este contexto, aprenderemos cómo implementar CRUD usando SQLite3 en Python, profundizando en los detalles técnicos y proporcionando ejemplos claros y aplicables.

Primeros Pasos

SQLite3 viene preinstalado con la mayoría de las distribuciones de Python, por lo que no es necesario instalar paquetes adicionales para comenzar a trabajar. Además, recomendamos usar herramientas como **DB Browser for SQLite** para visualizar y manipular bases de datos gráficamente. Esto es especialmente útil para ver el contenido de las tablas mientras se trabaja con consultas SQL desde Python.

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

|  |
| --- |
| import sqlite3 |

El Objeto Conexión

La función sqlite3.connect() se encarga de crear una conexión a la base de datos. Si la base de datos no existe, SQLite automáticamente creará un archivo para almacenarla. Esto significa que, a diferencia de otros RDBMS, no es necesario ejecutar un comando explícito como CREATE DATABASE. Además, SQLite no necesita configuraciones de usuario, contraseñas o puertos, ya que todo se maneja en el archivo local.

**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') |

|  |
| --- |
| El Objeto Cursor |

Un cursor es un intermediario entre Python y la base de datos que permite ejecutar comandos SQL. Una vez que se crea el objeto cursor, se pueden ejecutar consultas SQL utilizando el método execute.

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.

Es importante notar que las consultas en SQLite se ejecutan de manera secuencial, y Python solo trata estas consultas como cadenas de texto hasta que se ejecutan.

|  |
| --- |
| 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 el 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.

Relaciones y Consultas JOIN

Una de las características más poderosas de SQL es la capacidad de combinar datos de varias tablas usando **joins**. SQLite admite todos los tipos de joins comunes: INNER JOIN, LEFT JOIN, y CROSS

El dominio de las operaciones CRUD en bases de datos es esencial para cualquier analista de datos que desee manipular y gestionar información de manera eficiente. SQLite, junto con Python, ofrece una forma poderosa y sencilla de realizar estas operaciones, permitiendo el manejo de bases de datos relacionales sin necesidad de configuraciones complejas. A lo largo de este documento, hemos explorado cómo crear, leer, actualizar y eliminar registros en una base de datos SQLite, brindando una base sólida para comenzar a trabajar con bases de datos en proyectos de análisis de datos.

|  |
| --- |
| 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. | |
| 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. | |
| SELECT clientes.nombre, pedidos.numero  FROM clientes  LEFT JOIN pedidos ON clientes.id = pedidos.cliente\_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. | |
| 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 ". |

.

|  |
| --- |
| **Operaciones CRUD**  Veamos en detalle cómo realizar las operaciones CRUD en una base de datos SQLite utilizando Python: |
| Create (Insertar Datos)  La operación de **inserción** nos permite agregar nuevos registros a una tabla. Para ello, usamos el comando INSERT INTO. |
| # Insertar datos en la tabla 'clientes'  query = "INSERT INTO clientes (nombre, email) VALUES (?, ?)"  cursor.execute(query, ('Juan', 'juan@mail.com'))  # Confirmar los cambios en la base de datos  conexion.commit() |
| El uso de placeholders (?) en la consulta es una buena práctica para evitar inyecciones SQL y garantizar que los datos se formateen correctamente. Después de ejecutar el INSERT, es necesario llamar a conexion.commit() para guardar los cambios en la base de datos. |
| **Read (Leer Datos)**  La operación **Read** permite consultar y obtener información almacenada en la base de datos. SQLite ofrece varios métodos para recuperar los resultados de las consultas: fetchall(), fetchone(), y fetchmany().   * fetchall(): Recupera todos los registros de la consulta. * fetchone(): Recupera solo una fila a la vez. * fetchmany(n): Recupera un lote de n registros. |
| # Recuperar todos los registros de la tabla 'clientes'  query = 'SELECT \* FROM clientes'  cursor.execute(query)  resultados = cursor.fetchall()  # Mostrar los resultados  for fila in resultados:  print(f'ID: {fila[0]}, Nombre: {fila[1]}, Email: {fila[2]}') |
| **Transacciones y Manejo de Errores**  En bases de datos relacionales, las transacciones permiten agrupar varias operaciones SQL para que se ejecuten de forma atómica. Esto significa que, si una operación en una transacción falla, se pueden revertir todas las operaciones anteriores de esa transacción, garantizando que la base de datos se mantenga en un estado coherente.  Las funciones commit() y rollback() son cruciales en este proceso. Mientras que commit() guarda los cambios realizados en una transacción, rollback() deshace todos los cambios si se encuentra algún error. |
| try:  # Iniciar transacción  conexion.execute('BEGIN')  # Realizar varias operaciones  cursor.execute('INSERT INTO clientes (nombre, email) VALUES (?, ?)', ('Ana', 'ana@mail.com'))  cursor.execute('UPDATE productos SET stock = stock - 1 WHERE id = ?', (101,))  # Confirmar cambios si todo va bien  conexion.commit()  except Exception as e:  # Deshacer los cambios si ocurre un error  print(f"Error: {str(e)}")  conexion.rollback()  finally:  conexion.close()  Relaciones y Consultas JOIN |
| Una de las características más poderosas de SQL es la capacidad de combinar datos de varias tablas usando **joins**. SQLite admite todos los tipos de joins comunes: INNER JOIN, LEFT JOIN, y CROSS JOIN.  **INNER JOIN** devuelve solo las filas que tienen coincidencias en ambas tablas. |
| SELECT empleados.nombre, departamentos.nombre  FROM empleados  INNER JOIN departamentos ON empleados.departamento\_id = departamentos.id;  **LEFT JOIN** devuelve todas las filas de la tabla de la izquierda, junto con las coincidencias de la tabla de la derecha. Si no hay coincidencias, los campos de la tabla de la derecha se rellenan con NULL. |
| SELECT clientes.nombre, pedidos.numero  FROM clientes  LEFT JOIN pedidos ON clientes.id = pedidos.cliente\_id; |
| En Python, las consultas JOIN se ejecutan de la misma manera que cualquier otra consulta SQL:  query = """SELECT clientes.nombre, pedidos.numero  FROM clientes  LEFT JOIN pedidos ON clientes.id = pedidos.cliente\_id"""  cursor.execute(query)  resultados = cursor.fetchall()  for fila in resultados:  print(f'Cliente: {fila[0]}, Pedido: {fila[1]}') |
| Actualización de Datos (Update)  La operación **Update** se utiliza para modificar los registros existentes en una tabla. Para ello, se usa el comando UPDATE, seguido de la cláusula SET para especificar qué columnas deben actualizarse. |
| query = "UPDATE clientes SET email = ? WHERE id = ?"  cursor.execute(query, ('nuevo\_email@mail.com', 1))  conexion.commit() |
| Es fundamental usar WHERE en la instrucción UPDATE para evitar actualizar todos los registros de la tabla por error. |
| Eliminación de Datos (Delete)  La operación **Delete** permite eliminar registros de una tabla. Similar al UPDATE, se debe tener mucho cuidado con la cláusula WHERE para evitar eliminar todos los registros. |
| query = "DELETE FROM clientes WHERE id = ?"  cursor.execute(query, (1,))  conexion.commit() |

.

|  |
| --- |
| import sqlite3  # Establecer conexión  conexion = sqlite3.connect('ddbb\_sql3.db')  cursor = conexion.cursor()  # Crear tabla  cursor.execute('''CREATE TABLE IF NOT EXISTS clientes  (id INTEGER PRIMARY KEY, nombre TEXT, email TEXT)''')  # Insertar datos  cursor.execute('INSERT INTO clientes (nombre, email) VALUES (?, ?)', ('Carlos', 'carlos@mail.com'))  conexion.commit()  # Leer datos  cursor.execute('SELECT \* FROM clientes')  clientes = cursor.fetchall()  for cliente in clientes:  print(cliente)  # Actualizar datos  cursor.execute('UPDATE clientes SET email = ? WHERE id = ?', ('nuevo@mail.com', 1))  conexion.commit()  # Eliminar datos  cursor.execute('DELETE FROM clientes WHERE id = ?', (1,))  conexion.commit()  # Cerrar conexión  conexion.close() |
| Conclusión  El dominio de las operaciones CRUD en bases de datos es esencial para cualquier analista de datos que desee manipular y gestionar información de manera eficiente. SQLite, junto con Python, ofrece una forma poderosa y sencilla de realizar estas operaciones, permitiendo el manejo de bases de datos relacionales sin necesidad de configuraciones complejas. A lo largo de este documento, hemos explorado cómo crear, leer, actualizar y eliminar registros en una base de datos SQLite, brindando una base sólida para comenzar a trabajar con bases de datos en proyectos de análisis de datos. |

.

Una **vista** es una tabla virtual basada en el resultado de una consulta SQL. A diferencia de las tablas tradicionales, una vista no almacena datos de forma física; en su lugar, obtiene los datos de otras tablas según la consulta SQL que se haya definido. Las vistas pueden ser consultadas como si fueran tablas normales, lo que permite a los usuarios interactuar con la vista de manera similar a como lo harían con una tabla real.

|  |
| --- |
| Sintaxis para crear una vista |
| CREATE VIEW nombre\_de\_la\_vista AS  SELECT columnas  FROM tablas  WHERE condiciones; |
| Ventajas del uso de vistas   * **Simplificación de consultas complejas:** Una vista permite encapsular una consulta compleja, simplificando el acceso a los datos sin tener que escribir consultas largas repetidamente. * **Seguridad:** Las vistas pueden restringir el acceso a determinadas columnas o filas en una tabla, proporcionando una capa adicional de seguridad. * **Mantenimiento:** Si la lógica de los datos cambia, solo se necesita actualizar la consulta que define la vista, sin tener que modificar todas las consultas que utilizan esos datos. * **Reutilización:** Las vistas permiten reutilizar consultas SQL sin necesidad de repetir la lógica en múltiples lugares del código. |
| Ejemplo de creación de una vista  Supongamos que tienes una base de datos con dos tablas, clientes y pedidos, y deseas crear una vista que muestre solo los clientes que han realizado al menos un pedido.  Para ello, podrías crear una vista de la siguiente manera:  CREATE VIEW clientes\_con\_pedidos AS  SELECT clientes.id, clientes.nombre, clientes.email  FROM clientes  INNER JOIN pedidos ON clientes.id = pedidos.cliente\_id  WHERE pedidos.cliente\_id IS NOT NULL;  Aquí, la vista clientes\_con\_pedidos combina las tablas clientes y pedidos, y devuelve solo los registros donde existe al menos un pedido asociado a un cliente. |
| Una vez creada la vista, puedes consultarla como si fuera una tabla normal. Por ejemplo:  SELECT \* FROM clientes\_con\_pedidos;  Este comando mostrará todos los clientes que han realizado un pedido, simplificando el proceso de consulta. |
| **Actualización de datos a través de vistas**  En algunos sistemas de gestión de bases de datos, las vistas pueden ser actualizables, lo que significa que puedes modificar los datos subyacentes a través de la vista. Sin embargo, en SQLite, las vistas por defecto no son actualizables. Esto significa que no puedes hacer INSERT, UPDATE o DELETE directamente en una vista. Si deseas modificar los datos subyacentes, tendrás que hacerlo directamente en las tablas originales. |
| **Tipos de vistas**  Existen diferentes tipos de vistas que pueden crearse según las necesidades del proyecto:   * **Vistas simples:** Son aquellas basadas en una única tabla y que no contienen funciones agregadas, como SUM o AVG. Las vistas simples pueden ser más fácilmente modificables. * **Vistas complejas:** Son aquellas basadas en múltiples tablas o que incluyen funciones agregadas, subconsultas, GROUP BY, HAVING, etc. Estas vistas proporcionan una gran flexibilidad y son útiles para reportes y análisis de datos. |

.

|  |
| --- |
| **Ejemplo práctico con Python y SQLite** |
| Supongamos que estamos trabajando con una base de datos en SQLite en un entorno de Python, y queremos utilizar una vista para simplificar nuestras consultas. El siguiente código crea una vista en SQLite y la consulta desde Python: |
| import sqlite3  # Conectarse a la base de datos  conexion = sqlite3.connect('ddbb\_sql3.db')  cursor = conexion.cursor()  # Crear una tabla de clientes  cursor.execute('''CREATE TABLE IF NOT EXISTS clientes (  id INTEGER PRIMARY KEY,  nombre TEXT NOT NULL,  email TEXT NOT NULL)''')  # Crear una tabla de pedidos  cursor.execute('''CREATE TABLE IF NOT EXISTS pedidos (  id INTEGER PRIMARY KEY,  cliente\_id INTEGER,  producto TEXT,  FOREIGN KEY (cliente\_id) REFERENCES clientes(id))''')  # Insertar algunos datos en ambas tablas  cursor.execute("INSERT INTO clientes (nombre, email) VALUES ('Ariel', 'ariel@mail.com')")  cursor.execute("INSERT INTO pedidos (cliente\_id, producto) VALUES (1, 'Ordenador')")  conexion.commit()  # Crear una vista de clientes con pedidos  cursor.execute('''CREATE VIEW IF NOT EXISTS clientes\_con\_pedidos AS  SELECT clientes.nombre, clientes.email, pedidos.producto  FROM clientes  INNER JOIN pedidos ON clientes.id = pedidos.cliente\_id''')  # Consultar la vista  cursor.execute("SELECT \* FROM clientes\_con\_pedidos")  resultados = cursor.fetchall()  # Mostrar los resultados  for fila in resultados:  print(f'Nombre: {fila[0]}, Email: {fila[1]}, Producto: {fila[2]}')  # Cerrar la conexión  conexion.close() |
| En este ejemplo, hemos creado una vista llamada clientes\_con\_pedidos que combina la información de los clientes y los pedidos. Luego, consultamos la vista desde Python y mostramos los resultados.  Limitaciones de las vistas en SQLite  SQLite no soporta vistas actualizables de manera directa. Aunque puedes crear vistas basadas en tablas, no podrás insertar, actualizar o eliminar datos a través de esas vistas. Para realizar esas operaciones, deberás trabajar con las tablas subyacentes directamente.  Además, en SQLite no existen las "vistas materializadas", es decir, las vistas que almacenan los datos físicos. Cada vez que se consulta una vista en SQLite, se ejecuta la consulta subyacente, lo que puede ser un factor a tener en cuenta para el rendimiento en bases de datos muy grandes.  Casos de uso de las vistas en análisis de datos  En el contexto de análisis de datos, las vistas pueden ser extremadamente útiles para:   * **Filtrar datos**: Puedes crear vistas que solo contengan datos relevantes para tu análisis, como vistas que filtren registros de clientes con compras recientes o productos con bajas existencias. * **Agregación de datos**: Crear vistas que calculen valores agregados como promedios o sumas, y luego usar esas vistas en consultas posteriores para un análisis más profundo. * **Separar responsabilidades**: En un equipo, una vista puede ser utilizada para mostrar solo la información relevante a un rol específico sin exponer datos sensibles o innecesarios.   Conclusión  Las vistas son una herramienta poderosa en bases de datos relacionales que permite simplificar la consulta de datos, mejorar la seguridad y hacer que el código sea más fácil de mantener. Aunque SQLite tiene ciertas limitaciones en cuanto a la actualización de vistas, su uso sigue siendo fundamental en proyectos que requieren estructuración y simplificación de consultas SQL complejas, especialmente en el campo del análisis de datos. |