**Работа с базами данных и использованием SQLite3**

План лекции:

1. Введение в SQLite.
2. Установка и настройка SQLite.
3. Базовые SQL операции в SQLite.
4. Продвинутые SQL концепции в SQLite.
5. Практические примеры использования SQLite.
6. **Введение в SQLite**

SQLite (Structured Query Language) – это компактная встроенная реляционная база данных, которая предоставляет механизм хранения и доступа к данным без необходимости отдельного сервера базы данных. Она предоставляет полноценную реализацию языка SQL и поддерживает большинство его стандартных функций.

**Основные особенности SQLite**:

***Встроенная база данных***. SQLite не требует отдельного сервера базы данных, так как база данных хранится в виде обычного файла на компьютере или устройстве. Это делает ее легкой в использовании и подходящей для различных приложений.

***Независимость от платформы***. SQLite поддерживается на различных платформах, включая Windows, macOS, Linux, Android и другие. Это позволяет разработчикам создавать приложения, которые могут работать на разных операционных системах без изменений в коде.

***Компактность***. SQLite является одной из самых компактных баз данных, что делает ее идеальным выбором для ограниченных по ресурсам сред, таких как мобильные устройства или встроенные системы.

***Поддержка SQL*** SQLite полностью поддерживает язык SQL, что позволяет разработчикам выполнять запросы, создавать, изменять и удалять таблицы, индексы и другие элементы базы данных.

***Надежность***. SQLite обеспечивает атомарность, согласованность, изолированность и долговечность (ACID) для транзакций. Это означает, что данные остаются надежными и целостными даже при сбоях или ошибочных операциях.

**Ключевые особенности и преимущества использования SQLite:**

***Простота в использовании***. SQLite очень прост в интеграции и использовании. Он не требует сложной установки и настройки сервера базы данных, так как база данных хранится в одном файле. Это делает его идеальным выбором для небольших проектов или приложений.

***Компактность и эффективность***. SQLite является компактным и легким по размеру. Его библиотека имеет небольшой объем, что делает его идеальным для встраивания в различные приложения с ограниченными ресурсами. Он может работать с базами данных, состоящими из нескольких терабайт данных.

***Независимость от платформы***. SQLite поддерживается на множестве платформ, включая Windows, macOS, Linux, Android и другие. Это позволяет разработчикам создавать переносимые приложения, которые могут работать на различных операционных системах без изменений в коде.

***Полная поддержка SQL***. SQLite предоставляет полноценную реализацию языка SQL, включая поддержку сложных запросов, транзакций, представлений, триггеров и индексов. Это делает его мощным инструментом для работы с данными и обработки запросов.

***Высокая производительность***. SQLite обеспечивает высокую скорость выполнения запросов и обработки данных. Он имеет оптимизированный движок, который обеспечивает быстрый доступ к данным и эффективное выполнение запросов. SQLite также поддерживает индексы, что улучшает производительность при поиске и фильтрации данных.

***Надежность и целостность данных***. SQLite обеспечивает ACID-свойства для транзакций (атомарность, согласованность, изолированность и долговечность), что гарантирует надежность и целостность данных при обработке множества одновременных запросов или при возникновении сбоев.

***Расширяемость***. SQLite позволяет расширять его функциональность с помощью пользовательских функций и модулей расширения. Разработчики могут создавать свои собственные функции на языках C/C++ или использовать доступные модули расширения для решения специфических задач.

SQLite имеет свои уникальные преимущества и сценарии использования, которые делают его предпочтительным в некоторых случаях по сравнению с другими СУБД. Вот несколько сценариев, когда использование SQLite может быть особенно полезным:

***Мобильные приложения***. SQLite является популярным выбором для разработки мобильных приложений, так как он легко интегрируется и не требует отдельного сервера базы данных. Он обеспечивает хорошую производительность при работе с локальными данными на мобильных устройствах с ограниченными ресурсами.

***Встраиваемые системы***. SQLite отлично подходит для встраиваемых систем, таких как системы управления умным домом, системы навигации в автомобилях или встроенные устройства IoT. Его компактность, низкий потребляемый объем ресурсов и независимость от сервера делают его удобным выбором для таких систем.

***Прототипирование и разработка***. SQLite предоставляет быстрый и простой способ хранения и обработки данных при прототипировании приложений или во время разработки и тестирования. Он не требует сложной настройки и позволяет легко изменять схему базы данных на ходу.

***Небольшие проекты и локальные приложения***. Если вам нужна локальная база данных для небольшого проекта, например, для хранения настроек, кэширования данных или хранения локальных файлов, SQLite может быть простым и эффективным решением. Он не требует дополнительных ресурсов для установки и настройки сервера базы данных.

***Тестирование***. SQLite может быть полезным инструментом для создания тестовых сред, где данные могут быть сохранены и загружены с помощью базы данных. Это позволяет легко управлять тестовыми данными и выполнять различные сценарии тестирования.

1. **Установка и настройка SQLite**

Установка SQLite может немного отличаться в зависимости от операционной системы. Вот инструкции по установке SQLite на нескольких популярных операционных системах:

**Установка SQLite на Windows:**

Перейдите на официальный сайт SQLite (https://www.sqlite.org/) и перейдите на страницу загрузки.

Прокрутите вниз до раздела "Precompiled Binaries for Windows" и выберите ссылку для загрузки предварительно скомпилированного исполняемого файла для Windows.

Скачайте архив с исполняемым файлом SQLite.

Распакуйте скачанный архив в папку на вашем компьютере.

Добавьте путь к исполняемому файлу SQLite в переменную среды PATH Windows.

**Установка SQLite на macOS:**

SQLite уже предустановлен на большинстве macOS систем.

Откройте терминал и выполните команду "sqlite3" для запуска интерактивной оболочки SQLite.

Если у вас нет установленного SQLite или вы хотите обновить версию, можно установить его с помощью Homebrew или MacPorts.

**Установка SQLite на Linux:**

В большинстве дистрибутивов Linux SQLite доступен через официальные репозитории. Выполните команду в терминале, соответствующую вашему пакетному менеджеру:

Ubuntu/Debian: sudo apt-get install sqlite3

CentOS/Fedora: sudo yum install sqlite

После установки можно запустить SQLite, выполнив команду "sqlite3" в терминале.

1. **Базовые SQL операции в SQLite**

На языке Python для создания базы данных и таблицы в SQLite:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Создание таблицы "users"

cursor.execute('''CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY,

name TEXT,

age INTEGER

)''')

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы импортируем модуль sqlite3 и создаем подключение к базе данных с помощью sqlite3.connect(), указывая имя базы данных «mydatabase.db». Если база данных не существует, она будет создана автоматически.

Затем мы создаем объект-курсор с помощью conn.cursor(). Курсор позволяет выполнить SQL-запросы и манипуляции с базой данных.

Далее мы используем метод execute() курсора для выполнения команды SQL CREATE TABLE, создающей таблицу «users». В данном случае, мы создаем таблицу с тремя столбцами: «id» (целочисленный первичный ключ), «name» (текстовый столбец) и «age» (целочисленный столбец).

Обратите внимание на фразу IF NOT EXISTS в команде CREATE TABLE. Она гарантирует, что таблица будет создана только в том случае, если она еще не существует.

После выполнения команды cursor.execute() мы сохраняем изменения с помощью conn.commit(). Наконец, мы закрываем соединение с базой данных с помощью conn.close().

Теперь у вас будет база данных «mydatabase.db» с таблицей «users», которую можно использовать для добавления, выборки и обновления данных.

Примеры для вставки данных в таблицу SQLite с использованием библиотеки sqlite3:

Пример 1: Вставка одной строки данных

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Вставка одной строки данных в таблицу "users"

cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('John', 25))

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы используем метод execute() курсора для выполнения оператора INSERT INTO. Мы указываем имя таблицы «users» и столбцы «name» и «age», а затем передаем значения в виде параметров с помощью символов «?». Фактические значения передаются в виде кортежа ('John', 25).

Пример 2: Вставка нескольких строк данных

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Список данных для вставки

data = [

('Alice', 28),

('Bob', 32),

('Eve', 30)

]

# Вставка нескольких строк данных в таблицу "users"

cursor.executemany("INSERT INTO users (name, age) VALUES (?, ?)", data)

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы используем метод executemany() курсора, который позволяет вставить несколько строк данных за один раз. Мы передаем список данных data, где каждый элемент является кортежем с парами значений для столбцов «name» и «age».

После выполнения оператора execute() или executemany(), мы сохраняем изменения с помощью conn.commit(), а затем закрываем соединение с базой данных с помощью conn.close().

Примеры кода для выборки данных из таблицы SQLite с использованием библиотеки sqlite3:

Пример 1: Выборка всех строк данных

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выборка всех строк данных из таблицы "users"

cursor.execute("SELECT \* FROM users")

# Получение результатов выборки

rows = cursor.fetchall()

# Вывод результатов

for row in rows:

print(row)

# Закрытие подключения

conn.close()

В этом примере мы используем метод execute() курсора для выполнения оператора SELECT \* FROM users, который выбирает все строки данных из таблицы «users». Затем мы используем метод fetchall() для получения всех результатов выборки в виде списка кортежей.

Затем мы перебираем каждую строку данных и выводим ее содержимое.

Пример 2: Выборка с условием

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выборка строк данных из таблицы "users" с условием

cursor.execute("SELECT \* FROM users WHERE age > ?", (30,))

# Получение результатов выборки

rows = cursor.fetchall()

# Вывод результатов

for row in rows:

print(row)

# Закрытие подключения

conn.close()

В этом примере мы добавляем условие в оператор SELECT, чтобы выбрать строки данных из таблицы «users», где возраст (age) больше 30. Мы используем символ «?» в SQL-запросе и передаем значение 30 в виде параметра кортежа (30,). Обратите внимание, что параметры SQL передаются в виде кортежей, даже если у нас только один параметр.

Примеры для обновления данных в таблице SQLite:

Пример 1: Обновление одной строки данных

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Обновление одной строки данных в таблице "users"

cursor.execute("UPDATE users SET age = ? WHERE name = ?", (30, 'John'))

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы используем метод execute() курсора для выполнения оператора UPDATE. Мы обновляем значение столбца «age» на 30 для строки, где значение столбца «name» равно 'John'. Мы передаем значения в виде параметров с помощью символов «?».

Пример 2: Обновление нескольких строк данных

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Обновление нескольких строк данных в таблице "users"

cursor.execute("UPDATE users SET age = ? WHERE age < ?", (35, 30))

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы также используем метод execute() для выполнения оператора UPDATE. Мы обновляем значение столбца «age» на 35 для всех строк, где значение столбца «age» меньше 30.

После выполнения оператора execute(), мы сохраняем изменения с помощью conn.commit(). Затем мы закрываем соединение с базой данных с помощью conn.close().

Примеры кода для удаления данных из таблицы SQLite:

Пример 1: Удаление одной строки данных

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Удаление одной строки данных из таблицы "users"

cursor.execute("DELETE FROM users WHERE name = ?", ('John',))

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы используем метод execute() курсора для выполнения оператора DELETE FROM. Мы удаляем строку, где значение столбца «name» равно 'John'. Мы передаем значение в виде параметра с помощью символа «?».

Пример 2: Удаление с условием

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Удаление строк данных из таблицы "users" с условием

cursor.execute("DELETE FROM users WHERE age < ?", (30,))

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы также используем метод execute() для выполнения оператора DELETE FROM. Мы удаляем все строки, где значение столбца «age» меньше 30.

После выполнения оператора execute(), мы сохраняем изменения с помощью conn.commit(). Затем мы закрываем соединение с базой данных с помощью conn.close().

1. **Продвинутые SQL концепции в SQLite**

В Python с использованием библиотеки sqlite3 вы можете создавать индексы и оптимизировать запросы в SQLite. Вот примеры кода:

Создание индекса:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Создание индекса на столбце 'name' в таблице 'users'

cursor.execute("CREATE INDEX idx\_name ON users(name)")

# Сохранение изменений и закрытие подключения

conn.commit()

conn.close()

В этом примере мы используем метод execute() курсора для выполнения команды создания индекса CREATE INDEX. Индекс создается на столбце 'name' в таблице 'users'. Затем мы сохраняем изменения с помощью conn.commit() и закрываем соединение с базой данных.

Оптимизация запросов:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Включение режима оптимизации запросов

cursor.execute("PRAGMA optimize")

# Выполнение запроса

cursor.execute("SELECT \* FROM users WHERE age > ?", (30,))

# Получение результатов

rows = cursor.fetchall()

# Вывод результатов

for row in rows:

print(row)

# Закрытие подключения

conn.close()

В этом примере мы используем метод execute() курсора для выполнения команды PRAGMA optimize, которая включает режим оптимизации запросов. Затем мы выполняем запрос с использованием индексов или других оптимизаций. После получения результатов мы выводим их.

Помимо создания индексов и оптимизации запросов, также рекомендуется использовать правильные операторы SQL, написать эффективные условия WHERE, ограничивать объем выбираемых данных и использовать другие оптимизационные приемы для улучшения производительности запросов в SQLite.

В Python с использованием библиотеки sqlite3 вы можете работать с транзакциями и управлять конкурентным доступом в SQLite. Вот примеры кода:

Транзакции:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

try:

# Начало транзакции

conn.execute("BEGIN")

# Выполнение операций в рамках транзакции

cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ('John', 25))

cursor.execute("UPDATE users SET age = ? WHERE name = ?", (30, 'Alice'))

# Завершение транзакции

conn.commit()

except Exception as e:

# Откат транзакции в случае ошибки

conn.rollback()

print("Ошибка:", str(e))

# Закрытие подключения

conn.close()

В этом примере мы используем методы execute() и commit() для работы с транзакцией. Мы начинаем транзакцию с помощью conn.execute("BEGIN"), выполняем операции в рамках транзакции и, если все операции успешны, сохраняем изменения с помощью conn.commit(). В случае возникновения ошибки, мы откатываем транзакцию с помощью conn.rollback() и выводим сообщение об ошибке.

Управление конкурентным доступом:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Настройка режима блокировки для поддержки конкурентного доступа

conn.execute("PRAGMA locking\_mode = EXCLUSIVE")

# Создание объекта-курсора

cursor = conn.cursor()

# Выполнение запросов в рамках конкурентного доступа

cursor.execute("SELECT \* FROM users")

rows = cursor.fetchall()

# Закрытие подключения

conn.close()

В этом примере мы используем команду PRAGMA locking\_mode для настройки режима блокировки базы данных. В данном случае, мы устанавливаем режим блокировки EXCLUSIVE, который предотвращает одновременный доступ нескольких пользователей или процессов к базе данных. Затем мы выполняем запросы в рамках конкурентного доступа и закрываем подключение.

В Python с использованием библиотеки sqlite3 вы можете выполнять вложенные запросы и объединения (JOIN) в SQLite. Вот примеры кода:

Вложенные запросы:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выполнение вложенного запроса

cursor.execute("SELECT name FROM users WHERE age > (SELECT AVG(age) FROM users)")

# Получение результатов

rows = cursor.fetchall()

# Вывод результатов

for row in rows:

print(row[0])

# Закрытие подключения

conn.close()

В этом примере мы используем метод execute() курсора для выполнения вложенного запроса. Внешний запрос выбирает имена пользователей из таблицы «users», у которых возраст превышает средний возраст, полученный во вложенном запросе. Мы используем метод fetchall() для получения результатов выборки и выводим их.

Объединения (JOIN):

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выполнение объединения (JOIN)

cursor.execute("SELECT users.name, orders.order\_date FROM users JOIN orders ON users.id = orders.user\_id")

# Получение результатов

rows = cursor.fetchall()

# Вывод результатов

for row in rows:

print(row[0], row[1])

# Закрытие подключения

conn.close()

В этом примере мы используем метод execute() курсора для выполнения объединения (JOIN). Запрос выбирает имена пользователей и даты заказов из таблиц «users» и «orders» на основе условия сопоставления идентификатора пользователя (user\_id). Мы используем метод fetchall() для получения результатов выборки и выводим их.

В Python с использованием библиотеки sqlite3 вы можете использовать встроенные функции SQLite для выполнения различных операций над данными. Вот некоторые примеры использования встроенных функций SQLite:

**COUNT()**

Функция COUNT() используется для подсчета числа строк, удовлетворяющих заданному условию.

Пример использования:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выполнение запроса с использованием COUNT()

cursor.execute("SELECT COUNT(\*) FROM users")

# Получение результата

result = cursor.fetchone()

# Вывод результата

print("Количество пользователей:", result[0])

# Закрытие подключения

conn.close()

**SUM()**

Функция SUM() используется для получения суммы значений в столбце.

Пример использования:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выполнение запроса с использованием SUM()

cursor.execute("SELECT SUM(quantity) FROM products")

# Получение результата

result = cursor.fetchone()

# Вывод результата

print("Общее количество продуктов:", result[0])

# Закрытие подключения

conn.close()

**AVG()**

Функция AVG() используется для получения среднего значения столбца.

Пример использования:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выполнение запроса с использованием AVG()

cursor.execute("SELECT AVG(price) FROM products")

# Получение результата

result = cursor.fetchone()

# Вывод результата

print("Средняя цена продукта:", result[0])

# Закрытие подключения

conn.close()

**MAX() и MIN()**

Функции MAX() и MIN() используются для получения максимального и минимального значения в столбце соответственно.

Пример использования:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('mydatabase.db')

# Создание объекта-курсора

cursor = conn.cursor()

# Выполнение запроса с использованием MAX()

cursor.execute("SELECT MAX(price) FROM products")

# Получение результата

max\_price = cursor.fetchone()[0]

# Выполнение запроса с использованием MIN()

cursor.execute("SELECT MIN(price) FROM products")

# Получение результата

min\_price = cursor.fetchone()[0]

# Вывод результатов

print("Максимальная цена:", max\_price)

print("Минимальная цена:", min\_price)

# Закрытие подключения

conn.close()

1. **Практические примеры использования SQLite**

Рассмотрим пример цельного приложения на Python, использующего SQLite. Представим простую книжную библиотеку, в которой пользователи могут добавлять, просматривать, обновлять и удалять книги. Вот общая структура такого приложения:

import sqlite3

# Создание подключения к базе данных

conn = sqlite3.connect('library.db')

# Создание объекта-курсора

cursor = conn.cursor()

def create\_table():

# Создание таблицы books

cursor.execute('''CREATE TABLE IF NOT EXISTS books (

id INTEGER PRIMARY KEY AUTOINCREMENT,

title TEXT,

author TEXT,

year INTEGER

)''')

def add\_book(title, author, year):

# Вставка книги в таблицу

cursor.execute("INSERT INTO books (title, author, year) VALUES (?, ?, ?)", (title, author, year))

conn.commit()

print("Книга успешно добавлена.")

def view\_books():

# Получение всех книг из таблицы

cursor.execute("SELECT \* FROM books")

rows = cursor.fetchall()

for row in rows:

print(row)

def update\_book(id, title, author, year):

# Обновление книги в таблице

cursor.execute("UPDATE books SET title = ?, author = ?, year = ? WHERE id = ?", (title, author, year, id))

conn.commit()

print("Книга успешно обновлена.")

def delete\_book(id):

# Удаление книги из таблицы

cursor.execute("DELETE FROM books WHERE id = ?", (id,))

conn.commit()

print("Книга успешно удалена.")

def main():

create\_table()

while True:

print("\n------ Книжная библиотека ------")

print("1. Добавить книгу")

print("2. Просмотреть все книги")

print("3. Обновить книгу")

print("4. Удалить книгу")

print("5. Выйти")

choice = input("Выберите действие (1-5): ")

if choice == '1':

title = input("Введите название книги: ")

author = input("Введите автора книги: ")

year = int(input("Введите год издания книги: "))

add\_book(title, author, year)

elif choice == '2':

print("\n--- Список всех книг ---")

view\_books()

elif choice == '3':

id = int(input("Введите ID книги, которую хотите обновить: "))

title = input("Введите новое название книги: ")

author = input("Введите нового автора книги: ")

year = int(input("Введите новый год издания книги: "))

update\_book(id, title, author, year)

elif choice == '4':

id = int(input("Введите ID книги, которую хотите удалить: "))

delete\_book(id)

elif choice == '5':

break

else:

print("Неверный выбор. Попробуйте снова.")

# Закрытие подключения

conn.close()

main()

В этом примере мы создаем базу данных library.db, в которой хранятся книги. Функция create\_table() создает таблицу books, если она еще не существует. Далее, у нас есть функции для добавления (add\_book()), просмотра (view\_books()), обновления (update\_book()) и удаления (delete\_book()) книг из таблицы.

Функция main() представляет пользовательский интерфейс командной строки, где пользователь может выбрать опцию для выполнения соответствующей операции. Приложение продолжает работу до тех пор, пока пользователь не выберет опцию «Выйти» (5).

Это простой пример приложения, демонстрирующий использование SQLite для работы с базой данных книжной библиотеки. Вы можете расширить его функциональность, добавить дополнительные операции или использовать более сложные запросы SQLite в зависимости от требований вашего проекта.

SQLite3 – это легковесная и встроенная СУБД, которая предоставляет простой и эффективный способ работы с базами данных. Мы рассмотрели основные концепции SQLite3, такие как создание базы данных, создание таблиц, вставка данных, выборка данных, обновление и удаление данных.

SQLite3 предлагает ряд преимуществ, которые делают его привлекательным выбором для различных проектов. Он портативен, не требует отдельного сервера и может быть использован в различных сценариях, включая мобильные приложения, веб-приложения, десктопные приложения и даже встроенные системы. SQLite3 обладает надежностью, хорошей производительностью и поддерживает множество функций SQL для управления данными.

Мы изучили основные SQL операции, такие как создание таблиц, вставка данных, выборка данных, обновление и удаление данных. Кроме того, мы также рассмотрели продвинутые концепции, такие как использование индексов для оптимизации запросов, работу с транзакциями для обеспечения целостности данных и использование вложенных запросов и объединений (JOIN) для сложных операций с данными.