Лабораторная работа №4. "Введение в базы данных"

- как подключиться к различным СУБД с помощью библиотек Python для работы с SQL базами данных;
- как управлять базами данных SQLite, MySQL и PostgreSQL;
- как выполнять запросы к базе данных внутри приложения Python.

Contents

Лабораторная работа №4. "Введение в базы данных" 1

Разделы: 2

Порядок выполнения работы 2

Требования и состав отчёта 2

Материалы для подготовки: 2

1. Схема базы данных для обучения 3

2. Подключение к базам данных 4

SQLite 4

MySQL 5

3. Создание таблиц 7

SQLite 7

MySQL 9

4. Вставка записей 10

SQLite 11

MySQL 12

5. Извлечение данных из записей 13

SQLite 13

SELECT 14

JOIN 15

WHERE 16

MySQL 17

6. Обновление записей таблицы 18

SQLite 18

MySQL 19

7. Удаление записей таблицы 19

SQLite 19

Разделы:

1. Схема базы данных
2. Подключение к базам данных
3. Создание таблиц
4. Вставка записей
5. Извлечение записей
6. Обновление содержания
7. Удаление записей таблицы

Порядок выполнения работы

1. Изучить материалы для подготовки.
2. Выполнить задания-примеры для того, чтобы разобраться с принципом взаимодействия с базами данных с помощью языка программирования Python.
3. Выполнить задания для самостоятельной работы: SQLLite (обязательное задание – 60% от максимальной оценки) и MySQL (необязательное задание – 40% от максимальной оценки).
4. Составить отчет.

Требования и состав отчёта

1. Отчёт должен быть выполнен на листе размером А4 с использованием Microsoft Word, Libre Office и т.п.
2. Отчёт должен начинаться с титульного листа с названием вуза и факультета, номером и названием лабораторной работы, вариантом, ФИО студента, № группы, ФИО преподавателя, городом и годом.
3. Отчет должен содержать автособираемое оглавление (обязательные разделы – Задание, Основные этапы вычисления, Вывод).
4. В отчёте нужно кратко описать задание, показать основные этапы вычисления при выполнении всех операций, сформулировать выводы.
5. Отчёт предоставить в электронном виде и «защитить».

Материалы для подготовки:

SQL за 20 минут https://proglib.io/p/sql-for-20-minutes

6 бесплатных ресурсов для практики в SQL https://robotdreams.cc/blog/178-6-besplatnyh-resursov-dlya-praktiki-v-sql

20 вопросов и задач по SQL на собеседовании с ответами - https://vc.ru/life/443626-20-voprosov-i-zadach-po-sql-na-sobesedovanii-s-otvetami

SQL. Занимательные задачки - https://habr.com/ru/articles/461567/

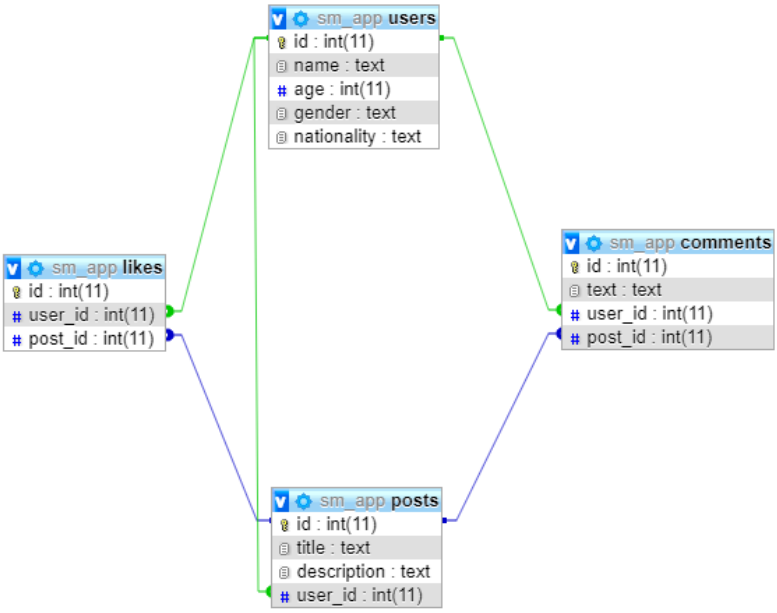
SQL. Задачки для тренировки - https://sql-academy.org/ru/trainer

1. Схема базы данных для обучения

Рассмотрение заданий буде идти на примере базы данных приложения для социальных сетей. База данных будет состоять из четырех таблиц:

1. users
2. posts
3. comments
4. likes

Схема базы данных показана на рисунке ниже.



Пользователи (users) и публикации (posts) будут находиться иметь тип связи один-ко-многим: одному читателю может понравиться несколько постов. Точно так же один и тот же юзер может оставлять много комментариев (comments), а один пост может иметь несколько комментариев. Таким образом, и users, и posts по отношению к comments имеют тот же тип связи. А лайки (likes) в этом плане идентичны комментариям.

2. Подключение к базам данных

Прежде чем взаимодействовать с любой базой данных через SQL-библиотеку, с ней необходимо связаться.

SQLite

SQLite, вероятно, является самой простой базой данных, к которой можно подключиться с помощью Python, поскольку для этого не требуется устанавливать какие-либо внешние модули. По умолчанию стандартная библиотека Python уже содержит модуль sqlite3.

Более того, SQLite база данных не требует сервера и самодостаточна, то есть просто читает и записывает данные в файл. Подключимся с помощью sqlite3 к базе данных:

```
import sqlite3
from sqlite3 import Error

def create_connection(path):
    connection = None
    try:
        connection = sqlite3.connect(path)
        print("Connection to SQLite DB successful")
    except Error as e:
        print(f"The error '{e}' occurred")

    return connection
```

Вот как работает этот код:

- Строки 1 и 2 – импорт sqlite3 и класса Error.
- Строка 4 определяет функцию create_connection(), которая принимает путь к базе данных SQLite.
- Строка 7 использует метод connect() и принимает в качестве параметра путь к базе данных SQLite. Если база данных в указанном месте существует, будет установлено соединение. В противном случае по указанному пути будет создана новая база данных и так же установлено соединение.
- В строке 8 выводится состояние успешного подключения к базе данных.
- Строка 9 перехватывает любое исключение, которое может быть получено, если методу .connect() не удастся установить соединение.
- В строке 10 отображается сообщение об ошибке в консоли.

sqlite3.connect(path) возвращает объект connection. Этот объект может использоваться для выполнения запросов к базе данных SQLite. Следующий скрипт формирует соединение с базой данных SQLite:

```
connection = create_connection("E:\\sm_app.sqlite")
```

Выполнив вышеуказанный скрипт, вы увидите, как в корневом каталоге диска E появится файл базы данных sm_app.sqlite. Конечно, вы можете изменить местоположение в соответствии с вашими интересами.

MySQL

В отличие от SQLite, в Python по умолчанию нет модуля, который можно использовать для подключения к базе данных MySQL. Для этого вам нужно установить драйвер Python для MySQL. Одним из таких драйверов является mysql-connector-python. Вы можете скачать этот модуль Python SQL с помощью pip:

```
pip install mysql-connector-python
```

Обратите внимание, что MySQL – это серверная система управления базами данных. Один сервер MySQL может хранить несколько баз данных. В отличие от SQLite, где соединение равносильно порождению БД, формирование базы данных MySQL состоит из двух этапов:

1. Установка соединения с сервером MySQL.
2. Выполнение запроса для создания БД.

Определим функцию, которая будет подключаться к серверу MySQL и возвращать объект подключения:

```
import mysql.connector
from mysql.connector import Error

def create_connection(host_name, user_name, user_password):
    connection = None
    try:
        connection = mysql.connector.connect(
            host=host_name,
            user=user_name,
            passwd=user_password
        )
        print("Connection to MySQL DB successful")
    except Error as e:
        print(f"The error '{e}' occurred")

    return connection

connection = create_connection("localhost", "root", "")
```

В приведенном выше коде мы определили новую функцию create_connection(), которая принимает три параметра:

1. host_name
2. user_name
3. user_password

Модуль mysql.connector определяет метод connect(), используемый в седьмой строке для подключения к серверу MySQL. Как только соединение установлено, объект connection возвращается вызывающей функции. В последней строке функция create_connection() вызывается с именем хоста, именем пользователя и паролем.

Пока мы только установили соединение. Самой базы ещё нет. Для этого мы определим другую функцию – create_database(), которая принимает два параметра:

1. Объект connection;
2. query – строковый запрос о создании базу данных.

Вот как выглядит эта функция:

```
    def create_database(connection, query):
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        print("Database created successfully")
    except Error as e:
        print(f"The error '{e}' occurred")
```

Для выполнения запросов используется объект cursor.

Создадим базу данных sm_app для нашего приложения на сервере MySQL:

```
create_database_query = "CREATE DATABASE sm_app"
create_database(connection, create_database_query)
```

Теперь у нас есть база данных на сервере. Однако объект connection, возвращаемый функцией create_connection() подключен к серверу MySQL. А нам необходимо подключиться к базе данных sm_app. Для этого нужно изменить create_connection() следующим образом:

```
 def create_connection(host_name, user_name, user_password, db_name):
    connection = None
    try:
        connection = mysql.connector.connect(
            host=host_name,
            user=user_name,
            passwd=user_password,
            database=db_name
        )
        print("Connection to MySQL DB successful")
    except Error as e:
```

```
        print(f"The error '{e}' occurred")

    return connection
```

Функция create_connection() теперь принимает дополнительный параметр с именем db_name. Этот параметр указывает имя БД, к которой мы хотим подключиться. Имя теперь можно передать при вызове функции:

```
connection = create_connection("localhost", "root", "", "sm_app")
```

Скрипт успешно вызывает create_connection() и подключается к базе данных sm_app

3. Создание таблиц

В предыдущем разделе мы увидели, как подключаться к серверам баз данных SQLite, MySQL и PostgreSQL, используя разные библиотеки Python. Мы создали базу данных sm_app на всех трех серверах БД. В данном разделе мы рассмотрим, как формировать таблицы внутри этих трех баз данных.

Как обсуждалось ранее, нам нужно получить и связать четыре таблицы:

1. users
2. posts
3. comments
4. likes

SQLite

Для выполнения запросов в SQLite используется метод cursor.execute(). В этом разделе мы определим функцию execute_query(), которая использует этот метод. Функция будет принимать объект connection и строку запроса. Далее строка запроса будет передаваться методу execute( ). В этом разделе он будет использоваться для формирования таблиц, а в следующих – мы применим его для выполнения запросов на обновление и удаление.

Примечание. Описываемый далее скрипт – часть того же файла, в котором мы описали соединение с базой данных SQLite.

Итак, начнем с определения функции execute_query():

```
def execute_query(connection, query):
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        connection.commit()
        print("Query executed successfully")
```

```
    except Error as e:
        print(f"The error '{e}' occurred")
```

Теперь напишем передаваемый запрос (query):

```
create_users_table = """
CREATE TABLE IF NOT EXISTS users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  age INTEGER,
  gender TEXT,
  nationality TEXT
);
"""
```

В запросе говорится, что нужно создать таблицу users со следующими пятью столбцами:

1. id
2. name
3. age
4. gender
5. nationality

Наконец, чтобы появилась таблица, вызываем execute_query(). Передаём объект connection, который мы описали в предыдущем разделе, вместе с только что подготовленной строкой запроса create_users_table:

```
execute_query(connection, create_users_table)
```

Следующий запрос используется для создания таблицы posts:

```
create_posts_table = """
CREATE TABLE IF NOT EXISTS posts(
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  title TEXT NOT NULL,
  description TEXT NOT NULL,
  user_id INTEGER NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users (id)
);
"""
```

Поскольку между users и posts имеет место отношение один-ко-многим, в таблице появляется ключ user_id, который ссылается на столбец id в таблице users. Выполняем следующий скрипт для построения таблицы posts:

```
execute_query(connection, create_posts_table)
```

Наконец, формируем следующим скриптом таблицы comments и likes:

```
create_comments_table = """
CREATE TABLE IF NOT EXISTS comments (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  text TEXT NOT NULL,
  user_id INTEGER NOT NULL,
  post_id INTEGER NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users (id) FOREIGN KEY (post_id)
REFERENCES posts (id)
);
"""

create_likes_table = """
CREATE TABLE IF NOT EXISTS likes (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  user_id INTEGER NOT NULL,
  post_id integer NOT NULL,
  FOREIGN KEY (user_id) REFERENCES users (id) FOREIGN KEY (post_id)
REFERENCES posts (id)
);
"""

execute_query(connection, create_comments_table)
execute_query(connection, create_likes_table)
```

Вы могли заметить, что создание таблиц в SQLite очень похоже на использование чистого SQL. Все, что вам нужно сделать, это сохранить запрос в строковой переменной и затем передать эту переменную cursor.execute().

MySQL

Так же, как с SQLite, чтобы создать таблицу в MySQL, нужно передать запрос в cursor.execute(). Создадим новый вариант функции execute_query():

```
def execute_query(connection, query):
     cursor = connection.cursor()
     try:
         cursor.execute(query)
         connection.commit()
```

```
        print("Query executed successfully")
    except Error as e:
        print(f"The error '{e}' occurred")
```

Описываем таблицу users:

```
create_users_table = """
CREATE TABLE IF NOT EXISTS users (
  id INT AUTO_INCREMENT,
  name TEXT NOT NULL,
  age INT,
  gender TEXT,
  nationality TEXT,
  PRIMARY KEY (id)
) ENGINE = InnoDB
"""

execute_query(connection, create_users_table)
```

Запрос для реализации отношения внешнего ключа в MySQL немного отличается от SQLite. Более того, MySQL использует ключевое слово AUTO_INCREMENT для указания столбцов, значения которых автоматически увеличиваются при вставке новых записей.

Следующий скрипт составит таблицу posts, содержащую внешний ключ user_id, который ссылается на id столбца таблицы users:

```
create_posts_table = """
CREATE TABLE IF NOT EXISTS posts (
  id INT AUTO_INCREMENT,
  title TEXT NOT NULL,
  description TEXT NOT NULL,
  user_id INTEGER NOT NULL,
  FOREIGN KEY fk_user_id (user_id) REFERENCES users(id),
  PRIMARY KEY (id)
) ENGINE = InnoDB
"""

execute_query(connection, create_posts_table)
```

Аналогично для создания таблиц comments и likes, передаём соответствующие CREATE-запросы функции execute_query().

4. Вставка записей

В предыдущем разделе мы разобрали, как развертывать таблицы в базах данных SQLite, MySQL и PostgreSQL с использованием различных модулей Python. В этом разделе узнаем, как вставлять записи.

SQLite

Чтобы вставить записи в базу данных SQLite, мы можем использовать ту же execute_query() функцию, что и для создания таблиц. Для этого сначала нужно сохранить в виде строки запрос INSERT INTO. Затем нужно передать объект connection и строковый запрос в execute_query(). Вставим для примера пять записей в таблицу users:

```
create_users = """
INSERT INTO
  users (name, age, gender, nationality)
VALUES
  ('James', 25, 'male', 'USA'),
  ('Leila', 32, 'female', 'France'),
  ('Brigitte', 35, 'female', 'England'),
  ('Mike', 40, 'male', 'Denmark'),
  ('Elizabeth', 21, 'female', 'Canada');
"""

execute_query(connection, create_users)
```

Поскольку мы установили автоинкремент для столбца id, нам не нужно указывать его дополнительно. Таблица users будет автоматически заполнена пятью записями со значениями id от 1 до 5.

Вставим в таблицу posts шесть записей:

```
create_posts = """
INSERT INTO
  posts (title, description, user_id)
VALUES
  ("Happy", "I am feeling very happy today", 1),
  ("Hot Weather", "The weather is very hot today", 2),
  ("Help", "I need some help with my work", 2),
  ("Great News", "I am getting married", 1),
  ("Interesting Game", "It was a fantastic game of tennis", 5),
  ("Party", "Anyone up for a late-night party today?", 3);
"""

execute_query(connection, create_posts)
```

Важно отметить, что столбец user_id таблицы posts является внешним ключом, который ссылается на столбец таблицы users. Это означает, что столбец user_id должен содержать значение, которое уже существует в столбце id таблицы users. Если его не существует, мы получим сообщение об ошибке.

Следующий скрипт вставляет записи в таблицы comments и likes:

```
create_comments = """
INSERT INTO
  comments (text, user_id, post_id)
VALUES
  ('Count me in', 1, 6),
  ('What sort of help?', 5, 3),
  ('Congrats buddy', 2, 4),
  ('I was rooting for Nadal though', 4, 5),
  ('Help with your thesis?', 2, 3),
  ('Many congratulations', 5, 4);
"""

create_likes = """
INSERT INTO
  likes (user_id, post_id)
VALUES
  (1, 6),
  (2, 3),
  (1, 5),
  (5, 4),
  (2, 4),
  (4, 2),
  (3, 6);
"""

execute_query(connection, create_comments)
execute_query(connection, create_likes)
```

MySQL

Есть два способа вставить записи в базы данных MySQL из приложения Python. Первый подход похож на SQLite. Можно сохранить запрос INSERT INTO в строке, а затем использовать для вставки записей cursor.execute().

Ранее мы определили функцию-оболочку execute_query(), которую использовали для вставки записей. Мы можем использовать ту же функцию:

```
create_users = """
INSERT INTO
  `users` (`name`, `age`, `gender`, `nationality`)
VALUES
  ('James', 25, 'male', 'USA'),
```

```
  ('Leila', 32, 'female', 'France'),
  ('Brigitte', 35, 'female', 'England'),
  ('Mike', 40, 'male', 'Denmark'),
  ('Elizabeth', 21, 'female', 'Canada');
"""

execute_query(connection, create_users)
```

Второй подход использует метод cursor.executemany(), который принимает два параметра:

1. Строка query, содержащая заполнители для вставляемых записей.
2. Список записей, которые мы хотим вставить.

Посмотрите на следующий пример, который вставляет две записи в таблицу likes:

```
sql = "INSERT INTO likes ( user_id, post_id ) VALUES ( %s, %s )"
val = [(4, 5), (3, 4)]

cursor = connection.cursor()
cursor.executemany(sql, val)
connection.commit()
```

Какой подход выбрать – зависит от вас. Если вы не очень хорошо знакомы с SQL, проще использовать метод курсора executemany().

5. Извлечение данных из записей

SQLite

Чтобы выбрать записи в SQLite, можно снова использовать cursor.execute(). Однако после этого потребуется вызвать метод курсора fetchall(). Этот метод возвращает список кортежей, где каждый кортеж сопоставлен с соответствующей строкой в извлеченных записях. Чтобы упростить процесс, напишем функцию execute_read_query():

```
def execute_read_query(connection, query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Error as e:
        print(f"The error '{e}' occurred")
```

Эта функция принимает объект connection и SELECT-запрос, а возвращает выбранную запись.

SELECT

Давайте выберем все записи из таблицы users:

```
select_users = "SELECT * from users"
users = execute_read_query(connection, select_users)

for user in users:
    print(user)
```

В приведенном выше скрипте запрос SELECT забирает всех пользователей из таблицы users. Результат передается в написанную нами функцию execute_read_query(), возвращающую все записи из таблицы users.

Примечание. Не рекомендуется использовать SELECT * для больших таблиц, так как это может привести к большому числу операций ввода-вывода, которые увеличивают сетевой трафик.

Результат вышеприведенного запроса выглядит следующим образом:

```
(1, 'James', 25, 'male', 'USA')
(2, 'Leila', 32, 'female', 'France')
(3, 'Brigitte', 35, 'female', 'England')
(4, 'Mike', 40, 'male', 'Denmark')
(5, 'Elizabeth', 21, 'female', 'Canada')
```

Таким же образом вы можете извлечь все записи из таблицы posts:

```
select_posts = "SELECT * FROM posts"
posts = execute_read_query(connection, select_posts)

for post in posts:
    print(post)
```

Вывод выглядит так:

```
(1, 'Happy', 'I am feeling very happy today', 1)
(2, 'Hot Weather', 'The weather is very hot today', 2)
```

```
(3, 'Help', 'I need some help with my work', 2)
(4, 'Great News', 'I am getting married', 1)
(5, 'Interesting Game', 'It was a fantastic game of tennis', 5)
(6, 'Party', 'Anyone up for a late-night party today?', 3)
```

JOIN

Вы также можете выполнять более сложные запросы, включающие операции типа JOIN для извлечения данных из двух связанных таблиц. Например, следующий скрипт возвращает идентификаторы и имена пользователей, а также описание сообщений, опубликованных этими пользователями:

```
select_users_posts = """
SELECT
  users.id,
  users.name,
  posts.description
FROM
  posts
  INNER JOIN users ON users.id = posts.user_id
"""

users_posts = execute_read_query(connection, select_users_posts)

for users_post in users_posts:
    print(users_post)
```

Вывод данных:

```
(1, 'James', 'I am feeling very happy today')
(2, 'Leila', 'The weather is very hot today')
(2, 'Leila', 'I need some help with my work')
(1, 'James', 'I am getting married')
(5, 'Elizabeth', 'It was a fantastic game of tennis')
(3, 'Brigitte', 'Anyone up for a late night party today?')
```

Следующий скрипт возвращает все сообщения вместе с комментариями к сообщениям и именами пользователей, которые разместили комментарии:

```
select_posts_comments_users = """
SELECT
  posts.description as post,
  text as comment,
  name
FROM
```

```
  posts
  INNER JOIN comments ON posts.id = comments.post_id
  INNER JOIN users ON users.id = comments.user_id
"""

posts_comments_users = execute_read_query(
    connection, select_posts_comments_users
)

for posts_comments_user in posts_comments_users:
    print(posts_comments_user)
```

Вывод выглядит так:

```
 ('Anyone up for a late night party today?', 'Count me in', 'James')
('I need some help with my work', 'What sort of help?', 'Elizabeth')
('I am getting married', 'Congrats buddy', 'Leila')
('It was a fantastic game of tennis', 'I was rooting for Nadal
though', 'Mike')
('I need some help with my work', 'Help with your thesis?', 'Leila')
('I am getting married', 'Many congratulations', 'Elizabeth')
```

Из вывода понятно, что имена столбцов не были возвращены методом fetchall(). Чтобы вернуть имена столбцов, нужно забрать атрибут description объекта cursor. Например, следующий список возвращает все имена столбцов для вышеуказанного запроса:

```
cursor = connection.cursor()
cursor.execute(select_posts_comments_users)
cursor.fetchall()

column_names = [description[0] for description in cursor.description]
print(column_names)
```

Вывод выглядит так:

```
['post', 'comment', 'name']
```

WHERE

Теперь мы выполним SELECT-запрос, который возвращает текст поста и общее количество лайков, им полученных:

```
select_post_likes = """
```

```
SELECT
  description as Post,
  COUNT(likes.id) as Likes
FROM
  likes,
  posts
WHERE
  posts.id = likes.post_id
GROUP BY
  likes.post_id
"""

post_likes = execute_read_query(connection, select_post_likes)

for post_like in post_likes:
    print(post_like)
```

Вывод следующий:

```
('The weather is very hot today', 1)
('I need some help with my work', 1)
('I am getting married', 2)
('It was a fantastic game of tennis', 1)
('Anyone up for a late night party today?', 2)
```

То есть используя запрос WHERE, вы можете возвращать более конкретные результаты.

MySQL

Процесс выбора записей в MySQL абсолютно идентичен процессу выбора записей в SQLite:

```
def execute_read_query(connection, query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Error as e:
        print(f"The error '{e}' occurred")
```

Теперь выберем все записи из таблицы users:

```
select_users = "SELECT * FROM users"
users = execute_read_query(connection, select_users)

for user in users:
    print(user)
```

Вывод будет похож на то, что мы видели с SQLite.

6. Обновление записей таблицы

SQLite

Обновление записей в SQLite выглядит довольно просто. Снова можно применить execute_query(). В качестве примера обновим текст поста с id равным 2. Сначала создадим описание для SELECT:

```
select_post_description = "SELECT description FROM posts WHERE id = 2"

post_description = execute_read_query(connection,
select_post_description)

for description in post_description:
    print(description)
```

Увидим следующий вывод:

```
('The weather is very hot today',)
```

Следующий скрипт обновит описание:

```
update_post_description = """
UPDATE
  posts
SET
  description = "The weather has become pleasant now"
WHERE
  id = 2
"""

execute_query(connection, update_post_description)
```

Теперь, если мы выполним SELECT-запрос еще раз, увидим следующий результат:

```
('The weather has become pleasant now',)
```

То есть запись была обновлена.

MySQL

Процесс обновления записей в MySQL с помощью модуля mysql-connector-python является точной копией модуля sqlite3:

```
update_post_description = """
UPDATE
  posts
SET
  description = "The weather has become pleasant now"
WHERE
  id = 2
"""

execute_query(connection,  update_post_description)
```

7. Удаление записей таблицы

SQLite

В качестве примера удалим комментарий с id равным 5:

```
delete_comment = "DELETE FROM comments WHERE id = 5"
execute_query(connection, delete_comment)
```

Теперь, если мы извлечем все записи из таблицы comments, то увидим, что пятый комментарий был удален. Процесс удаления в MySQL идентичен SQLite.

ЗАДАНИЕ

1. Проделайте все шаги из примера выше. Удостоверьтесь, что Вы поняли, как это работает.
2. Самостоятельно выберите тематику для своей собственной базы данных. База данных должна содержать не менее четырех таблиц. Каждая таблица

должна иметь минимум одну связь с другими таблицами для формирования сложных запросов. Каждая таблица содержит не менее 5 записей.

3. Нарисуйте схему базы данных как в примере с указанием связей между таблицами.
4. Создайте таблицы с помощью Python для SQLite и MySQL.
5. Добавьте по одной новой записи в каждую из таблиц Вашей базы данных.
6. Продемонстрируйте работу запросов на извлечение данных:

- выбрать все записи из таблиц,

- составить запрос по извлечению данных с использованием JOIN

- составить запрос по извлечению данных с использованием WHERE и GROUP BY
Составить два запроса, в которых будет вложенный SELECT-запрос (вложение с помощью WHERE.
- составить 2 запроса с использованием UNION (объединение запросов).
Составить 1 запрос с использованием DISTINCT. Если для демонстрации работы этого ключевого слова недостаточно данных – предварительно дополните таблицу.

7. Обновить две записи в двух разных таблицах Вашей базы данных

8. Удалить по одной записи из каждой таблицы.

Удалите все записи в одной из таблиц.

Сформируйте отчет.