Python и MySQL: практическое введение

Рассмотрим на практическом примере, как из кода Python делать SQL-запросы к MySQL-серверу: CREATE, SELECT, UPDATE, JOIN и т. д.

Публикация представляет собой незначительно сокращенный перевод статьи Чайтаньи Баведжи [Python and MySQL Database: A Practical Introduction](https://realpython.com/python-mysql/" \t "_blank). Материал также адаптирован в виде [блокнота Jupyter](https://github.com/matyushkin/lessons/blob/master/databases/mysql.ipynb).

\*\*\*

Большинство приложений в той или иной форме взаимодействует с данными. Поэтому языки программирования (Python не исключение), предоставляют инструменты хранения источников данных и доступа к ним. MySQL — одна из самых популярных систем управления базами данных ([СУБД](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B0%D0%BC%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)). В прошлом году она заняла второе место после СУБД Oracle в [рейтинге баз данных](https://db-engines.com/en/ranking).

Используя методы, описанные в этом руководстве, вы сможете эффективно интегрировать базу данных MySQL в приложение на Python. В ходе руководства мы разработаем небольшую базу данных MySQL для системы рейтинга фильмов и узнаем, как забирать из нее данные с помощью Python-кода.

Начало формы

Конец формы

[Хотите начать свой путь в анализе данных и давно изучали школьную математику?](https://proglib.io/w/ca860e5f" \t "_blank)

К концу этого урока вы сможете:

* Подключить ваше приложение к базе данных MySQL
* Сделать запрос к базе данных для получения необходимых данных
* Обработать исключения, возникающие при доступе к базе данных

Чтобы получить максимальную отдачу от этого руководства, желательно иметь практические знания о таких концепциях Python, как цикл for, функции, обработка исключений. Также необходимо иметь [базовые представления](https://proglib.io/p/sql-for-20-minutes) о SQL-запросах, таких как SELECT, DROP, CREATE и JOIN.

Сравнение MySQL с другими SQL-базами данных

SQL ([Structured Query Language](https://ru.wikipedia.org/wiki/SQL" \t "_blank)) — язык структурированных запросов. SQL является широко используемым языком программирования для управления реляционными базами данных. Возможно, вы слышали о различных СУБД на основе SQL: [MySQL](https://www.mysql.com/" \t "_blank), [PostgreSQL](https://www.postgresql.org/" \t "_blank), [SQLite](https://www.sqlite.org/index.html" \t "_blank) и [SQL Server](https://www.microsoft.com/en-us/sql-server/sql-server-2019). Все эти базы данных соответствуют [стандартам SQL](https://docs.oracle.com/cd/B28359_01/server.111/b28286/intro002.htm), но отличаются в деталях.

В силу открытости исходного кода MySQL быстро стал лидером рынка среди SQL-решений. В настоящее время MySQL используется всеми крупными техническими фирмами, включая Google, LinkedIn, Uber, Netflix, Twitter и другие.

Помимо поддержки со стороны open source-сообщества, есть и другие причины успеха MySQL:

1. Простота установки. MySQL разработан, чтобы быть удобным для пользователя. Базу данных легко создать и настроить. MySQL доступен для основных операционных систем, включая Windows, macOS, Linux и Solaris.
2. Скорость. MySQL имеет репутацию быстрого решения для баз данных. Еще эта СУБД хорошо масштабируется.
3. Права пользователя и безопасность. MySQL позволяет устанавливать уровни безопасности паролей, добавлять и удалять привилегии учетным записям пользователей. Управление правами пользователей выглядит существенно проще, чем у многих других СУБД, таких как PostgreSQL, где управление [файлами конфигурации](https://www.postgresql.org/docs/9.3/config-setting.html), требует некоторой сноровки.

MySQL использует синтаксис, похожий на стандартный SQL, однако имеющий некоторые важные отличия, описанные в [официальной документации](https://dev.mysql.com/doc/refman/8.0/en/differences-from-ansi.html).

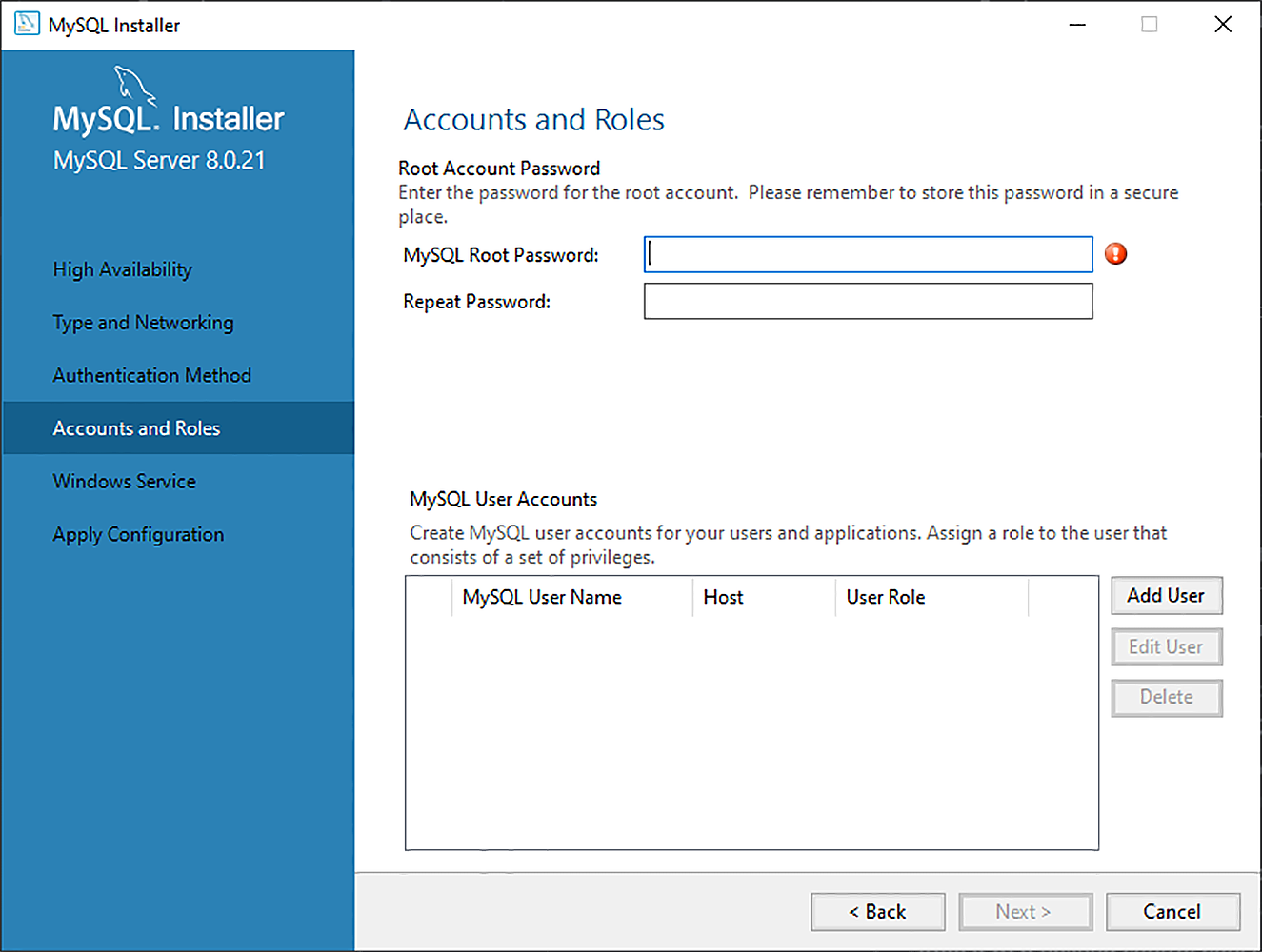
Установка MySQL Server и MySQL Connector

Чтобы начать работу с этим руководством, вам необходимо настроить две вещи: MySQL Server и MySQL Connector. MySQL Server предоставит ресурсы, необходимые для работы с базой данных. После запуска сервера вы сможете подключить к нему свое приложение Python с помощью MySQL Connector/Python.

Установка MySQL Server

Официальная документация описывает рекомендуемые способы загрузки и установки MySQL Server. Есть инструкции для всех популярных операционных систем, включая [Windows](https://dev.mysql.com/doc/refman/8.0/en/windows-installation.html" \t "_blank), [macOS](https://dev.mysql.com/doc/refman/8.0/en/osx-installation.html" \t "_blank), [Solaris](https://dev.mysql.com/doc/refman/8.0/en/solaris-installation.html" \t "_blank), [Linux](https://dev.mysql.com/doc/refman/8.0/en/linux-installation.html" \t "_blank) и многие другие.

Для Windows лучше всего загрузить [установщик MySQL](https://dev.mysql.com/downloads/installer/) и позволить ему позаботиться о процессе. Диспетчер установки также поможет настроить параметры безопасности сервера MySQL. На странице учетных записей будет необходимо ввести пароль для root-записи и при желании добавить других пользователей с различными привилегиями.

Настройка учетной записи MySQL

С помощью установщиков можно настроить и другие полезные инструменты, например, [MySQL Workbench](https://www.mysql.com/products/workbench/" \t "_blank). Удобная альтернатива установке в операционной системе — [развернуть MySQL с помощью Docker](https://dev.mysql.com/doc/refman/8.0/en/linux-installation-docker.html).

Установка MySQL Connector/Python

Драйвер базы данных — программное обеспечение, позволяющее приложению подключаться и взаимодействовать с СУБД. Такие драйверы обычно поставляются в виде отдельных модулей. Сандартный интерфейс, которому должны соответствовать все драйверы баз данных Python, описан в [PEP 249](https://www.python.org/dev/peps/pep-0249/). Драйверы баз данных Python, такие как [sqlite3](https://docs.python.org/3/library/sqlite3.html) для SQLite, [psycopg](https://www.psycopg.org/docs/" \t "_blank) для PostgreSQL и [MySQL Connector/Python](https://github.com/mysql/mysql-connector-python" \t "_blank) для MySQL, следуют этим правилам.

Для установки драйвера (коннектора) воспользуемся менеджером пакетов pip:

pip install mysql-connector-python

pip установит коннектор в текущую активную среду. Чтобы работать с проектом изолированным образом, мы рекомендуем [настроить виртуальную среду](https://proglib.io/p/kak-sozdat-virtualnoe-okruzhenie-v-python-i-perestat-dumat-o-zavisimostyah-2020-03-10).

Проверим результат установки, запустив в терминале Python следующую команду:

import mysql.connector

Если инструкция импорта выполняется без ошибок, значит mysql.connector успешно установлен и готов к использованию.

Установление соединения с сервером MySQL

MySQL ― это серверная система управления базами данных. Один сервер может содержать несколько баз данных. Чтобы взаимодействовать с базой данных, мы должны установить соединение с сервером. Рабочий процесс программы Python, которая взаимодействует с базой данных на основе MySQL, в общих чертах выглядит следующим образом:

1. Подключаемся к серверу MySQL.
2. Создаем новую базу данных (при необходимости).
3. Соединяемся с базой данных.
4. Выполняем SQL-запрос, собираем результаты.
5. Сообщаем базе данных, если в таблицу внесены изменения.
6. Закрываем соединением с сервером MySQL.

Каким бы ни было приложение, первый шаг ― связать между собой приложение и базу данных.

Подключаемся к серверу MySQL из Python

Чтобы установить соединение, используем connect() из модуля mysql.connector. Эта функция принимает параметры host, user и password, а возвращает объект MySQLConnection. Учетные данные можно получить в результате ввода от пользователя:

from getpass import getpass

from mysql.connector import connect, Error

try:

with connect(

host="localhost",

user=input("Имя пользователя: "),

password=getpass("Пароль: "),

) as connection:

print(connection)

except Error as e:

print(e)

Объект MySQLConnection хранится в переменной connection, которую мы будем использовать для доступа к серверу MySQL. Несколько важных моментов:

* Все соединения с базой данных оборачивайтев блоки try ... except. Так будет проще перехватить и изучить любые исключения.
* Не забывайте закрывать соединение после завершения доступа к базе данных. Неиспользуемые открытые соединения приводят к неожиданным ошибкам и проблемам с производительностью. В коде для этого используется диспетчер контекста (with ... as ...).
* Никогда не следует встраивать учетные данные (имя пользователя и пароль) в строковом виде в скрипт Python. Это плохая практика для развертывания, которая представляет серьезную угрозу безопасности. Приведенный код запрашивает для входа учетные данные. Для этого используется встроенный модуль getpass, чтобы скрыть вводимый пароль. Хотя это лучше, чем жесткое кодирование, но есть и другие, более безопасные способы хранения конфиденциальной информации, например, использование переменных среды.

Итак, мы установили соединение между нашей программой и сервером MySQL. Теперь нужно либо создать новую базу данных, либо подключиться к существующей.

Создаем новую базу данных

Чтобы создать новую базу данных, например, с именем online\_movie\_rating, нужно выполнить инструкцию SQL:

CREATE DATABASE online\_movie\_rating;

Примечание

MySQL обязывает ставить точку с запятой (;) в конце оператора. Однако MySQL Connector/Python автоматически добавляет точку с запятой в конце каждого запроса.

Чтобы выполнить SQL-запрос, нам понадобится курсор, который абстрагирует процесс доступа к записям базы данных. MySQL Connector/Python предоставляет соответствующий класс MySQLCursor, экземпляр которого также называется курсором.

Передадим наш запрос о создании базы данных online\_movie\_rating:

try:

with connect(

host="localhost",

user=input("Имя пользователя: "),

password=getpass("Пароль: "),

) as connection:

create\_db\_query = "CREATE DATABASE online\_movie\_rating"

with connection.cursor() as cursor:

cursor.execute(create\_db\_query)

except Error as e:

print(e)

Запрос CREATE DATABASE сохраняется в виде строки в переменной create\_db\_query, а затем передается на выполнение в cursor.execute().

Если база данных с таким именем уже существует на сервере, мы получим сообщение об ошибке. Используя тот же объект MySQLConnection, что и ранее, выполним запрос SHOW DATABASES, чтобы увидеть все таблицы, хранящиеся в базе данных:

try:

with connect(

host="localhost",

user=input("Введите имя пользователя: "),

password=getpass("Введите пароль: "),

) as connection:

show\_db\_query = "SHOW DATABASES"

with connection.cursor() as cursor:

cursor.execute(show\_db\_query)

for db in cursor:

print(db)

except Error as e:

print(e)

Введите имя пользователя: root

Введите пароль: ········

('information\_schema',)

('mysql',)

('online\_movie\_rating',)

('performance\_schema',)

('sys',)

Приведенный код выведет имена всех баз данных, находящихся на нашем сервере MySQL. Команда SHOW DATABASES в нашем примере также вывела базы данных, которые автоматически создаются сервером MySQL и предоставляют доступ к метаданным баз данных и настройкам сервера.

Подключение к существующей базе данных

Итак, мы создали базу данных под названием online\_movie\_rating. Чтобы к ней подключиться, просто дополняем вызов connect() параметром database:

try:

with connect(

host="localhost",

user=input("Имя пользователя: "),

password=getpass("Пароль: "),

database="online\_movie\_rating",

) as connection:

print(connection)

except Error as e:

print(e)

Создание, изменение и удаление таблиц

В этом разделе мы рассмотрим, как с помощью Python выполнять некоторые базовые запросы: CREATE TABLE, DROP и ALTER.

Определение схемы базы данных

Начнем с создания схемы базы данных для рейтинговой системы фильмов. База данных будет состоять из трех таблиц:

1. movies ― общая информация о фильмах:

* id
* title
* release year
* genre
* collection\_in\_mi

2. reviewers ― информация о людях, опубликовавших оценки фильмов:

1. id
2. first\_name
3. last\_name

3. ratings ― информация об оценках фильмов рецензентами:

1. movie\_id (foreign key)
2. reviewer\_id (foreign key)
3. rating

Этих трех таблиц достаточно для целей данного руководства.

Схема системы рейтинга фильмов

Таблицы в базе данных связаны друг с другом: movies и reviewers должны иметь отношение «многие ко многим»: один фильм может быть просмотрен несколькими рецензентами, а один рецензент может рецензировать несколько фильмов. Таблица ratings соединяет таблицу фильмов с таблицей рецензентов.

Создание таблиц с помощью оператора CREATE TABLE

Чтобы создать новую таблицу в MySQL, нам нужно использовать оператор CREATE TABLE. Следующий запрос MySQL создаст таблицу movies нашей базы данных online\_movie\_rating:

CREATE TABLE movies(

id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(100),

release\_year YEAR(4),

genre VARCHAR(100),

collection\_in\_mil INT

);

Если вы раньше встречались с SQL, вам будет понятен смысл приведенного запроса. У диалекта MySQL есть некоторые отличительные черты. Например, MySQL предлагает широкий выбор типов данных, включая YEAR, INT, BIGINT и так далее. Кроме того, MySQL использует ключевое слово AUTO\_INCREMENT, когда значение столбца должно автоматически увеличиваться при вставке новых записей.

Чтобы создать таблицу, необходимо передать указанный запрос в cursor.execute():

create\_movies\_table\_query = """

CREATE TABLE movies(

id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(100),

release\_year YEAR(4),

genre VARCHAR(100),

collection\_in\_mil INT

)

"""

with connection.cursor() as cursor:

cursor.execute(create\_movies\_table\_query)

connection.commit()

Обратите внимание на оператор connection.commit(). По умолчанию коннектор MySQL не выполняет автоматическую фиксацию транзакций. В MySQL модификации, упомянутые в транзакции, происходят только тогда, когда мы используем в конце команду COMMIT. Чтобы внести изменения в таблицу, всегда вызывайте этот метод после каждой транзакции.

Повторим процедуру для таблицы reviewers:

create\_reviewers\_table\_query = """

CREATE TABLE reviewers (

id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(100),

last\_name VARCHAR(100)

)

"""

with connection.cursor() as cursor:

cursor.execute(create\_reviewers\_table\_query)

connection.commit()

Наконец, создадим таблицу ratings:

create\_ratings\_table\_query = """

CREATE TABLE ratings (

movie\_id INT,

reviewer\_id INT,

rating DECIMAL(2,1),

FOREIGN KEY(movie\_id) REFERENCES movies(id),

FOREIGN KEY(reviewer\_id) REFERENCES reviewers(id),

PRIMARY KEY(movie\_id, reviewer\_id)

)

"""

with connection.cursor() as cursor:

cursor.execute(create\_ratings\_table\_query)

connection.commit()

Реализация отношений внешнего ключа в MySQL немного отличается и имеет ограничения в сравнении со стандартным SQL. В MySQL и родитель, и потомок внешнего ключа должны использовать один и тот же механизм хранения ― базовый программный компонент, который система управления базами данных использует для выполнения SQL-операций. MySQL предлагает два вида таких механизмов:

1. Транзакционные механизмы хранения безопасны для транзакций и позволяют откатывать транзакции с помощью простых команд, таких как rollback. К этой категории относятся многие популярные движки MySQL, включая [InnoDB](https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html" \t "_blank) и [NDB](https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html).
2. Нетранзакционные механизмы хранения для отмены операторов, зафиксированных в базе данных, опираются на ручной код. Это, например [MyISAM](https://dev.mysql.com/doc/refman/8.0/en/myisam-storage-engine.html" \t "_blank) и [MEMORY](https://dev.mysql.com/doc/refman/8.0/en/memory-storage-engine.html).

InnoDB ― самый популярный механизм хранения по умолчанию. Соблюдая ограничения внешнего ключа, он помогает поддерживать целостность данных. Это означает, что любая [CRUD](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)-операция с внешним ключом предварительно проверяется на то, что она не приводит к несогласованности между разными таблицами.

Обратите внимание, что таблица ratings использует столбцы movie\_id и reviewer\_id, как два внешних ключа, выступающих вместе в качестве первичного ключа. Эта особенность гарантирует, что рецензент не сможет дважды оценить один и тот же фильм.

Один и тот же курсор можно использовать для нескольких обращений. В этом случае все обращения станут одной атомарной транзакцией. Например, можно выполнить все операторы CREATE TABLE одним курсором, а затем зараз зафиксировать транзакцию:

with connection.cursor() as cursor:

cursor.execute(create\_movies\_table\_query)

cursor.execute(create\_reviewers\_table\_query)

cursor.execute(create\_ratings\_table\_query)

connection.commit()

Отображение схемы таблиц с использованием оператора DESCRIBE

Мы создали три таблицы и можем просмотреть схему, используя оператор DESCRIBE.

Предполагая, что у вас уже есть объект MySQLConnection в переменной connection, мы можем распечатать результаты, полученные с помощью cursor.fetchall(). Этот метод извлекает все строки из последнего выполненного оператора:

show\_table\_query = "DESCRIBE movies"

with connection.cursor() as cursor:

cursor.execute(show\_table\_query)

# Fetch rows from last executed query

result = cursor.fetchall()

for row in result:

print(row)

('id', 'int(11)', 'NO', 'PRI', None, 'auto\_increment')

('title', 'varchar(100)', 'YES', '', None, '')

('release\_year', 'year(4)', 'YES', '', None, '')

('genre', 'varchar(100)', 'YES', '', None, '')

('collection\_in\_mil', 'int(11)', 'YES', '', None, '')

После выполнения приведенного кода мы должны получить таблицу, содержащую информацию о столбцах в таблице movies. Для каждого столбца выводится информация, о типе данных, является ли столбец первичным ключом и т. д.

Изменение схемы таблицы с помощью оператора ALTER

Столбец с именем collection\_in\_mil в таблице movies содержит кассовые сборы фильма в миллионах долларов. Мы можем написать следующую инструкцию MySQL, чтобы изменить тип данных атрибута collection\_in\_mil с INT на DECIMAL:

ALTER TABLE movies MODIFY COLUMN collection\_in\_mil DECIMAL(4,1);

DECIMAL(4,1) указывает на десятичное число, которое может иметь максимум 4 цифры, из которых 1 соответствует разряду десятых, например, 120.1, 3.4, 38.0 и т. д.

alter\_table\_query = """

ALTER TABLE movies

MODIFY COLUMN collection\_in\_mil DECIMAL(4,1)

"""

show\_table\_query = "DESCRIBE movies"

with connection.cursor() as cursor:

cursor.execute(alter\_table\_query)

cursor.execute(show\_table\_query)

# Получить строки из последнего выполненного запроса

result = cursor.fetchall()

print("Схема таблицы movie после внесения изменений:")

for row in result:

print(row)

Схема таблицы movie после внесения изменений:

('id', 'int(11)', 'NO', 'PRI', None, 'auto\_increment')

('title', 'varchar(100)', 'YES', '', None, '')

('release\_year', 'year(4)', 'YES', '', None, '')

('genre', 'varchar(100)', 'YES', '', None, '')

('collection\_in\_mil', 'decimal(4,1)', 'YES', '', None, '')

Как показано в выходных данных, атрибут collection\_in\_mil сменил тип на DECIMAL(4,1). Обратите внимание, что в приведенном выше коде мы дважды вызываем cursor.execute(), но cursor.fetchall() выбирает строки только из последнего выполненного запроса, которым является show\_table\_query.

Удаление таблиц с помощью оператора DROP

Для удаления таблиц служит оператор DROP TABLE. Удаление таблицы ― необратимый процесс. Если вы выполните приведенный ниже код, вам нужно будет снова вызвать запрос CREATE TABLE для таблицы ratings:

drop\_table\_query = "DROP TABLE ratings"

with connection.cursor() as cursor:

cursor.execute(drop\_table\_query)

Вставка записей в таблицы

Заполним таблицы данными. В этом разделе мы рассмотрим два способа вставки записей с помощью MySQL Connector в коде Python.

Первый метод, .execute(), хорошо работает, когда количество записей невелико. Второй, .executemany() лучше подходит для реальных сценариев.

Вставка записей с помощью .execute()

Первый подход использует тот же метод cursor.execute(), который мы применяли до сих пор. Пишем запрос INSERT INTO и передаем в cursor.execute():

insert\_movies\_query = """

INSERT INTO movies (title, release\_year, genre, collection\_in\_mil)

VALUES

("Forrest Gump", 1994, "Drama", 330.2),

("3 Idiots", 2009, "Drama", 2.4),

("Eternal Sunshine of the Spotless Mind", 2004, "Drama", 34.5),

("Good Will Hunting", 1997, "Drama", 138.1),

("Skyfall", 2012, "Action", 304.6),

("Gladiator", 2000, "Action", 188.7),

("Black", 2005, "Drama", 3.0),

("Titanic", 1997, "Romance", 659.2),

("The Shawshank Redemption", 1994, "Drama",28.4),

("Udaan", 2010, "Drama", 1.5),

("Home Alone", 1990, "Comedy", 286.9),

("Casablanca", 1942, "Romance", 1.0),

("Avengers: Endgame", 2019, "Action", 858.8),

("Night of the Living Dead", 1968, "Horror", 2.5),

("The Godfather", 1972, "Crime", 135.6),

("Haider", 2014, "Action", 4.2),

("Inception", 2010, "Adventure", 293.7),

("Evil", 2003, "Horror", 1.3),

("Toy Story 4", 2019, "Animation", 434.9),

("Air Force One", 1997, "Drama", 138.1),

("The Dark Knight", 2008, "Action",535.4),

("Bhaag Milkha Bhaag", 2013, "Sport", 4.1),

("The Lion King", 1994, "Animation", 423.6),

("Pulp Fiction", 1994, "Crime", 108.8),

("Kai Po Che", 2013, "Sport", 6.0),

("Beasts of No Nation", 2015, "War", 1.4),

("Andadhun", 2018, "Thriller", 2.9),

("The Silence of the Lambs", 1991, "Crime", 68.2),

("Deadpool", 2016, "Action", 363.6),

("Drishyam", 2015, "Mystery", 3.0)

"""

with connection.cursor() as cursor:

cursor.execute(insert\_movies\_query)

connection.commit()

аблица movies теперь заполнена тридцатью записями. В конце код вызывает connection.commit(). Не забывайте вызывать .commit() после выполнения любых изменений в таблице.

Вставка записей с помощью .executemany()

Предыдущий подход годится, когда количество записей мало, и их можно вставить из кода. Но обычно данные хранятся в файле или генерируются другим сценарием. Вот где пригодится .executemany(). Метод принимает два параметра:

1. Запрос, содержащий заполнители для записей, которые необходимо вставить.
2. Список записей для вставки.

Применим подход для заполнения таблицы reviewers:

insert\_reviewers\_query = """

INSERT INTO reviewers

(first\_name, last\_name)

VALUES ( %s, %s )

"""

reviewers\_records = [

("Chaitanya", "Baweja"),

("Mary", "Cooper"),

("John", "Wayne"),

("Thomas", "Stoneman"),

("Penny", "Hofstadter"),

("Mitchell", "Marsh"),

("Wyatt", "Skaggs"),

("Andre", "Veiga"),

("Sheldon", "Cooper"),

("Kimbra", "Masters"),

("Kat", "Dennings"),

("Bruce", "Wayne"),

("Domingo", "Cortes"),

("Rajesh", "Koothrappali"),

("Ben", "Glocker"),

("Mahinder", "Dhoni"),

("Akbar", "Khan"),

("Howard", "Wolowitz"),

("Pinkie", "Petit"),

("Gurkaran", "Singh"),

("Amy", "Farah Fowler"),

("Marlon", "Crafford"),

]

with connection.cursor() as cursor:

cursor.executemany(insert\_reviewers\_query,

reviewers\_records)

connection.commit()

Этот код использует %s в качестве заполнителей для двух строк, которые вставляются в insert\_reviewers\_query. Заполнители действуют как спецификаторы формата и помогают зарезервировать место для переменной внутри строки.

Аналогичным образом заполним таблицу ratings:

insert\_ratings\_query = """

INSERT INTO ratings

(rating, movie\_id, reviewer\_id)

VALUES ( %s, %s, %s)

"""

ratings\_records = [

(6.4, 17, 5), (5.6, 19, 1), (6.3, 22, 14), (5.1, 21, 17),

(5.0, 5, 5), (6.5, 21, 5), (8.5, 30, 13), (9.7, 6, 4),

(8.5, 24, 12), (9.9, 14, 9), (8.7, 26, 14), (9.9, 6, 10),

(5.1, 30, 6), (5.4, 18, 16), (6.2, 6, 20), (7.3, 21, 19),

(8.1, 17, 18), (5.0, 7, 2), (9.8, 23, 3), (8.0, 22, 9),

(8.5, 11, 13), (5.0, 5, 11), (5.7, 8, 2), (7.6, 25, 19),

(5.2, 18, 15), (9.7, 13, 3), (5.8, 18, 8), (5.8, 30, 15),

(8.4, 21, 18), (6.2, 23, 16), (7.0, 10, 18), (9.5, 30, 20),

(8.9, 3, 19), (6.4, 12, 2), (7.8, 12, 22), (9.9, 15, 13),

(7.5, 20, 17), (9.0, 25, 6), (8.5, 23, 2), (5.3, 30, 17),

(6.4, 5, 10), (8.1, 5, 21), (5.7, 22, 1), (6.3, 28, 4),

(9.8, 13, 1)

]

with connection.cursor() as cursor:

cursor.executemany(insert\_ratings\_query, ratings\_records)

connection.commit()

Теперь все три таблицы заполнены данными. Следующий шаг ― разобраться, как с этой базой данных взаимодействовать.

Чтение записей из базы данных

До сих пор мы только создавали элементы базы данных. Пришло время выполнить несколько запросов и найти интересующие нас свойства. В этом разделе мы узнаем, как читать записи из таблиц базы данных с помощью оператора SELECT.

Чтение записей с помощью оператора SELECT

Чтобы получить записи, необходимо отправить в cursor.execute() запрос SELECT и вернуть результат с помощью cursor.fetchall():

select\_movies\_query = "SELECT \* FROM movies LIMIT 5"

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

result = cursor.fetchall()

for row in result:

print(row)

(1, 'Forrest Gump', 1994, 'Drama', Decimal('330.2'))

(2, '3 Idiots', 2009, 'Drama', Decimal('2.4'))

(3, 'Eternal Sunshine of the Spotless Mind', 2004, 'Drama', Decimal('34.5'))

(4, 'Good Will Hunting', 1997, 'Drama', Decimal('138.1'))

(5, 'Skyfall', 2012, 'Action', Decimal('304.6'))

Переменная result содержит записи, возвращенные с помощью .fetchall(). Это список кортежей, представляющих отдельные записи таблицы.

В приведенном запросе мы используем ключевое слово LIMIT, чтобы ограничить количество строк, получаемых от оператора SELECT. Разработчики часто используют LIMIT для разбивки выдачи на страницы при обработке больших объемов данных.

В MySQL оператору LIMIT можно передать два неотрицательных числовых аргумента:

SELECT \* FROM movies LIMIT 2,5;

При использовании двух числовых аргументов первый указывает смещение, равное в данном примере 2, а второй ограничивает количество возвращаемых строк до 5. То есть запрос из примера вернет строки с 3 по 7.

select\_movies\_query = "SELECT title, release\_year FROM movies LIMIT 2, 5"

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

for row in cursor.fetchall():

print(row)

('Eternal Sunshine of the Spotless Mind', 2004)

('Good Will Hunting', 1997)

('Skyfall', 2012)

('Gladiator', 2000)

('Black', 2005)

Фильтрация результатов с помощью WHERE

Записи таблицы также можно фильтровать, используя WHERE. Чтобы получить все фильмы с кассовыми сборами свыше 300 млн долларов, выполним следующий запрос:

select\_movies\_query = """

SELECT title, collection\_in\_mil

FROM movies

WHERE collection\_in\_mil > 300

ORDER BY collection\_in\_mil DESC

"""

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

for movie in cursor.fetchall():

print(movie)

('Avengers: Endgame', Decimal('858.8'))

('Titanic', Decimal('659.2'))

('The Dark Knight', Decimal('535.4'))

('Toy Story 4', Decimal('434.9'))

('The Lion King', Decimal('423.6'))

('Deadpool', Decimal('363.6'))

('Forrest Gump', Decimal('330.2'))

('Skyfall', Decimal('304.6'))

Словосочетание ORDER BY в запросе позволяет отсортировать сборы от самого высокого до самого низкого.

MySQL предоставляет множество операций форматирования строк, таких как CONCAT для объединения строк. Например, названия фильмов, чтобы избежать путаницы, обычно отображается вместе с годом выпуска. Получим названия пяти самых прибыльных фильмов вместе с датами их выхода в прокат:

select\_movies\_query = """

SELECT CONCAT(title, " (", release\_year, ")"),

collection\_in\_mil

FROM movies

ORDER BY collection\_in\_mil DESC

LIMIT 5

"""

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

for movie in cursor.fetchall():

print(movie)

('Avengers: Endgame (2019)', Decimal('858.8'))

('Titanic (1997)', Decimal('659.2'))

('The Dark Knight (2008)', Decimal('535.4'))

('Toy Story 4 (2019)', Decimal('434.9'))

('The Lion King (1994)', Decimal('423.6'))

Если вы не хотите использовать LIMIT и вам не нужно получать все записи, можно использовать методы курсора .fetchone() и .fetchmany():

* .fetchone() извлекает следующую строку результата в виде кортежа, либо None, если доступных строк больше нет.
* .fetchmany() извлекает следующий набор строк из результата в виде списка кортежей. Для этого ему передается аргумент, по умолчанию равный 1. Если доступных строк больше нет, метод возвращает пустой список.

Снова извлечем названия пяти самых кассовых фильмов с указанием года выпуска, но на этот раз используя .fetchmany():

select\_movies\_query = """

SELECT CONCAT(title, " (", release\_year, ")"),

collection\_in\_mil

FROM movies

ORDER BY collection\_in\_mil DESC

"""

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

for movie in cursor.fetchmany(size=5):

print(movie)

cursor.fetchall()

('Avengers: Endgame (2019)', Decimal('858.8'))

('Titanic (1997)', Decimal('659.2'))

('The Dark Knight (2008)', Decimal('535.4'))

('Toy Story 4 (2019)', Decimal('434.9'))

('The Lion King (1994)', Decimal('423.6'))

Вы могли заметить дополнительный вызов cursor.fetchall(). Мы делаем это, чтобы очистить все оставшиеся результаты, которые не были прочитаны .fetchmany().

Перед выполнением любых других операторов в том же соединении необходимо очистить все непрочитанные результаты. В противном случае вызывается исключение InternalError.

Обработка нескольких таблиц с помощью оператора JOIN

Чтобы узнать названия пяти фильмов с самым высоким рейтингом, выполним следующий запрос:

select\_movies\_query = """

SELECT title, AVG(rating) as average\_rating

FROM ratings

INNER JOIN movies

ON movies.id = ratings.movie\_id

GROUP BY movie\_id

ORDER BY average\_rating DESC

LIMIT 5

"""

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

for movie in cursor.fetchall():

print(movie)

('Night of the Living Dead', Decimal('9.90000'))

('The Godfather', Decimal('9.90000'))

('Avengers: Endgame', Decimal('9.75000'))

('Eternal Sunshine of the Spotless Mind', Decimal('8.90000'))

('Beasts of No Nation', Decimal('8.70000'))

Найти имя рецензента, давшего наибольшее количество оценок, можно так:

select\_movies\_query = """

SELECT CONCAT(first\_name, " ", last\_name), COUNT(\*) as num

FROM reviewers

INNER JOIN ratings

ON reviewers.id = ratings.reviewer\_id

GROUP BY reviewer\_id

ORDER BY num DESC

LIMIT 1

"""

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

for movie in cursor.fetchall():

print(movie)

('Mary Cooper', 4)

Как видим, больше всего рецензий написала Mary Cooper.

Не имеет значения, насколько сложен запрос ― в конечном счете он обрабатывается сервером MySQL. Процесс выполнения запроса всегда остается прежним: передаем запрос в cursor.execute(), получаем результаты с помощью .fetchall().

Обновление и удаление записей из базы данных

В этом разделе мы обновим и удалим часть записей. Необходимые строки мы выберем с помощью ключевого слова WHERE.

Команда UPDATE

Представим, что рецензент Amy Farah Fowler вышла замуж за Sheldon Cooper. Она сменила фамилию на Cooper, и нам необходимо обновить базу данных. Для обновления записей в MySQL используется оператор UPDATE:

update\_query = """

UPDATE

reviewers

SET

last\_name = "Cooper"

WHERE

first\_name = "Amy"

"""

with connection.cursor() as cursor:

cursor.execute(update\_query)

connection.commit()

Код передает запрос на обновление в cursor.execute(), а .commit() вносит необходимые изменения в таблицу reviewers.

Представим, что мы хотим дать возможность рецензентам изменять оценки. Программа должна знать movie\_id, reviewer\_id и новый rating. Пример на SQL:

UPDATE

ratings

SET

rating = 5.0

WHERE

movie\_id = 18 AND reviewer\_id = 15;

SELECT \*

FROM ratings

WHERE

movie\_id = 18 AND reviewer\_id = 15;

Указанные запросы сначала обновляют рейтинг, а затем выведут обновленный. Напишем скрипт на Python, который позволит корректировать оценки:

modify\_ratings.py

from getpass import getpass

from mysql.connector import connect, Error

movie\_id = input("Enter movie id: ")

reviewer\_id = input("Enter reviewer id: ")

new\_rating = input("Enter new rating: ")

update\_query = """

UPDATE

ratings

SET

rating = "%s"

WHERE

movie\_id = "%s" AND reviewer\_id = "%s";

SELECT \*

FROM ratings

WHERE

movie\_id = "%s" AND reviewer\_id = "%s"

""" % (

new\_rating,

movie\_id,

reviewer\_id,

movie\_id,

reviewer\_id,

)

try:

with connect(

host="localhost",

user=input("Enter username: "),

password=getpass("Enter password: "),

database="online\_movie\_rating",

) as connection:

with connection.cursor() as cursor:

for result in cursor.execute(update\_query, multi=True):

if result.with\_rows:

print(result.fetchall())

connection.commit()

except Error as e:

print(e)

Enter movie id: 18

Enter reviewer id: 15

Enter new rating: 5

Enter username: root

Enter password: ········

[(18, 15, Decimal('5.0'))]

Чтобы передать несколько запросов одному курсору, мы присваиваем аргументу multi значение True. В этом случае cursor.execute() возвращает итератор. Каждый элемент в итераторе соответствует объекту курсора, который выполняет инструкцию, переданную в запросе. Приведенный код запускает на этом итераторе цикл for, вызывая .fetchall() для каждого объекта курсора.

Если для операции не был получен набор результатов, то .fetchall() вызывает исключение. Чтобы избежать этой ошибки, в приведенном коде мы используем свойство cursor.with\_rows, которое указывает, создавала ли строки последняя выполненная операция.

Хотя этот код решает поставленную задачу, инструкция WHERE в текущем виде является заманчивой целью для хакеров. Она уязвима для атаки с использованием [SQL-инъекции](https://proglib.io/p/vzlamyvaem-sayty-shpargalka-po-sql-inekciyam-2019-12-21), позволяющей злоумышленникам повредить базу данных или использовать ее не по назначению.

Например, если пользователь отправляет movie\_id = 18, reviewer\_id = 15 и rating = 5.0 в качестве входных данных, то результат будет выглядеть так:

$ python modify\_ratings.py

Enter movie id: 18

Enter reviewer id: 15

Enter new rating: 5.0

Enter username: <user\_name>

Enter password:

[(18, 15, Decimal('5.0'))]

Оценка для movie\_id = 18 и reviewer\_id = 15 изменилась на 5.0. Но если бы вы были хакером, вы могли отправить на вход скрытую команду:

$ python modify\_ratings.py

Enter movie id: 18

Enter reviewer id: 15"; UPDATE reviewers SET last\_name = "A

Enter new rating: 5.0

Enter username: <user\_name>

Enter password:

[(18, 15, Decimal('5.0'))]

И снова выходные данные показывают, что указанный рейтинг был изменен на 5.0. Что изменилось?

Хакер перехватил запрос на обновление данных. Запрос на обновление, изменит last\_name всех записей в таблице рецензентов "A":

>>> select\_query = """

... SELECT first\_name, last\_name

... FROM reviewers

... """

>>> with connection.cursor() as cursor:

... cursor.execute(select\_query)

... for reviewer in cursor.fetchall():

... print(reviewer)

...

('Chaitanya', 'A')

('Mary', 'A')

('John', 'A')

('Thomas', 'A')

('Penny', 'A')

('Mitchell', 'A')

('Wyatt', 'A')

('Andre', 'A')

('Sheldon', 'A')

('Kimbra', 'A')

('Kat', 'A')

('Bruce', 'A')

('Domingo', 'A')

('Rajesh', 'A')

('Ben', 'A')

('Mahinder', 'A')

('Akbar', 'A')

('Howard', 'A')

('Pinkie', 'A')

('Gurkaran', 'A')

('Amy', 'A')

('Marlon', 'A')

Приведенный код отображает first\_name и last\_name для всех записей в таблице проверяющих. Атака с использованием SQL-инъекции повредила эту таблицу, изменив last\_name всех записей на «A».

Есть быстрое решение для предотвращения таких атак. Не добавляйте значения запроса, предоставленные пользователем, напрямую в строку запроса. Лучше обнолять сценарий с отправкой значений запроса в качестве аргументов в .execute():

modify\_ratings.py

from getpass import getpass

from mysql.connector import connect, Error

movie\_id = input("Enter movie id: ")

reviewer\_id = input("Enter reviewer id: ")

new\_rating = input("Enter new rating: ")

update\_query = """

UPDATE

ratings

SET

rating = %s

WHERE

movie\_id = %s AND reviewer\_id = %s;

SELECT \*

FROM ratings

WHERE

movie\_id = %s AND reviewer\_id = %s

"""

val\_tuple = (

new\_rating,

movie\_id,

reviewer\_id,

movie\_id,

reviewer\_id,

)

try:

with connect(

host="localhost",

user=input("Enter username: "),

password=getpass("Enter password: "),

database="online\_movie\_rating",

) as connection:

with connection.cursor() as cursor:

for result in cursor.execute(update\_query, val\_tuple, multi=True):

if result.with\_rows:

print(result.fetchall())

connection.commit()

except Error as e:

print(e)

Обратите внимание, что плейсхолдеры %s больше не заключены в строковые кавычки. cursor.execute() проверяет, что значения в кортеже, полученном в качестве аргумента, имеют требуемый тип данных. Если пользователь попытается ввести какие-то проблемные символы, код вызовет исключение:

$ python modify\_ratings.py

Enter movie id: 18

Enter reviewer id: 15"; UPDATE reviewers SET last\_name = "A

Enter new rating: 5.0

Enter username: <user\_name>

Enter password:

1292 (22007): Truncated incorrect DOUBLE value: '15";

UPDATE reviewers SET last\_name = "A'

Такой подход стоит использовать всегда, когда вы включаете в запрос пользовательский ввод. Не поленитесь узнать и про [другие способы предотвращения атак с использованием SQL-инъекций](https://realpython.com/prevent-python-sql-injection/).

Удаление записей: команда DELETE[¶](http://localhost:8888/lab#%D0%A3%D0%B4%D0%B0%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B7%D0%B0%D0%BF%D0%B8%D1%81%D0%B5%D0%B9:-%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0-DELETE)

Процедура удаления записей очень похожа на их обновление. Поскольку DELETE является необратимой операцией, мы рекомендуем сначала запускать запрос SELECT с тем же фильтром, чтобы убедиться, что вы удаляете нужные записи. Например, чтобы удалить все оценки фильмов, данные reviewer\_id = 2, мы можем сначала запустить соответствующий запрос SELECT:

select\_movies\_query = """

SELECT reviewer\_id, movie\_id FROM ratings

WHERE reviewer\_id = 2

"""

with connection.cursor() as cursor:

cursor.execute(select\_movies\_query)

for movie in cursor.fetchall():

print(movie)

(2, 7)

(2, 8)

(2, 12)

(2, 23)

Приведенный фрагмент кода выводит пары reviewer\_id и movie\_id для записей в таблице оценок, для которых reviewer\_id = 2. Убедившись, что это те записи, которые нужно удалить, выполним запрос DELETE с тем же фильтром:

delete\_query = "DELETE FROM ratings WHERE reviewer\_id = 2"

with connection.cursor() as cursor:

cursor.execute(delete\_query)

connection.commit()

Другие способы соединения Python и MySQL

В этом руководстве мы познакомились с MySQL Connector/Python, который является официально рекомендуемым средством взаимодействия с базой данных MySQL из приложения Python. Вот еще пара популярных коннекторов:

* [mysqlclient](https://github.com/PyMySQL/mysqlclient-python) ― библиотека, которая является конкурентом официального коннектора и активно дополняется новыми функциями. Поскольку ядро библиотеки написано на C, она имеет лучшую производительность, чем официальный коннектор на чистом Python. Большой недостаток состоит в том, что mysqlclient довольно сложно настроить и установить, особенно в Windows.
* [MySQLdb](https://mysqlclient.readthedocs.io/user_guide.html) ― устаревшее программное обеспечение, которое до сих пор используется в коммерческих приложениях. Написано на C и быстрее MySQL Connector/Python, но доступно только для Python 2.

Эти драйверы действуют, как интерфейсы между вашей программой и базой данных MySQL. Фактически вы просто отправляете через них свои SQL-запросы. Но многие разработчики предпочитают использовать для управления данными не SQL-запросы, а объектно-ориентированную парадигму.

Объектно-реляционное отображение ([ORM](https://ru.wikipedia.org/wiki/ORM)) — метод, который позволяет запрашивать и управлять данными из базы данных напрямую, используя объектно-ориентированный язык. ORM-библиотека инкапсулирует код, необходимый для управления данными, освобождая разработчиков от необходимости использовать SQL-запросы. Вот самые популярные ORM-библиотеки для связки Python и SQL:

* [SQLAlchemy](https://docs.sqlalchemy.org/en/13/index.html) ― это ORM, которая упрощает взаимодействие между Python и другими базами данных SQL. Вы можете создавать разные движки для разных баз данных, таких как MySQL, PostgreSQL, SQLite и т. д. Читайте наш [туториал по SQLAlchemy](https://proglib.io/p/upravlenie-dannymi-s-pomoshchyu-python-sqlite-i-sqlalchemy-2020-10-21" \t "_blank).
* [peewee](https://docs.peewee-orm.com/en/latest/) ― легкая и быстрая ORM-библиотека с простой настройкой, что очень полезно, когда ваше взаимодействие с базой данных ограничивается извлечением нескольких записей. Если нужно скопировать отдельные записи из базы данных MySQL в csv-файл, то лучший выбор ― peewee.
* [Django ORM](https://books.agiliq.com/projects/django-orm-cookbook/en/latest/introduction.html) ― одна из самых мощных составляющих веб-фреймворка Django, позволяющая простым образом взаимодействовать с различными базами данных SQLite, PostgreSQL и MySQL. Многие приложения на основе Django используют Django ORM для моделирования данных и базовых запросов, однако для более сложных задач разработчики обычно используют SQLAlchemy.

Заключение

В этом руководстве мы познакомились с применением MySQL Connector/Python для интеграции базы данных MySQL в ваше приложение Python. Мы также разработали тестовый образец базы данных MySQL и повзаимодействовали с ней непосредственно из Python-кода. Дополнительные сведения можно найти в [официальной документации](https://dev.mysql.com/doc/connector-python/en/).

Python имеет коннекторы и для других СУБД, таких как [MongoDB](https://realpython.com/introduction-to-mongodb-and-python/" \t "_blank) и [PostgreSQL](https://proglib.io/p/kak-podruzhit-python-i-bazy-dannyh-sql-podrobnoe-rukovodstvo-2020-02-27" \t "_blank). Будем рады узнать, какие еще материалы по Python и базам данных вам были бы интересны.