# Ход работы

Была разработана программа на языке Python, в качестве СУБД использовалась SQLite. Был создан класс MyDB, в котором есть методы установки соединения с сервером СУБД и выполнения запросов. При запуске программы создается объект класса, в конструкторе происходит подключение к файлу базы данных, для этого используется следующая функция:

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

Соответственно, для подключения к бд в коде необходимо указать требуемый путь к файлу. Также в конструкторе объявляется курсор для дальнейшей работы с бд:

self.cur = self.connection.cursor()

Далее в main происходит анализ входных параметров и выбор режима работы.

## Создание таблицы

Вызывается метод create\_table, в котором с помощью курсора выполняется запрос по созданию таблицы. Также в запросе проверяется, что указанная таблица не существует перед созданием.

def create\_table(self):  
 try:  
 self.cur.execute(  
 '''CREATE TABLE IF NOT EXISTS employee(fio TEXT NOT NULL, date\_of\_birth DATE, sex TEXT NOT NULL);''')  
 self.connection.commit()  
 print("Query executed successfully")  
 except Error as e:  
 print(f"The error '{e}' occurred")

## Создание записи в таблице

Был создан класс запроса:

class Query:  
 def \_\_init\_\_(self, fio, dob, sex):  
 self.fio = fio  
 self.dob = dob  
 self.sex = sex  
  
 def \_\_str\_\_(self):  
 return f"('{self.fio}', '{self.dob}', '{self.sex}')"

Если был выбран режим работы 2, то проверяется количество аргументов командной строки. Для вставки нескольких строк пользователь должен ввести подряд все строки. Далее в цикле создается объект класса и формируются данные для запроса, на каждой итерации полученная строка данных добавляется к заранее заготовленной строке, которая содержит начало запроса — ключевое слово INSERT.

elif sys.argv[1] == "2":  
 argc = (len(sys.argv) - 2)  
 values\_list = f"INSERT INTO 'employee' (fio, date\_of\_birth, sex) VALUES "  
 i = 2  
 while argc > 0:  
 query = Query(sys.argv[i], sys.argv[i + 1], sys.argv[i + 2])  
 values\_list += query.\_\_str\_\_()  
 i += 3  
 argc -= 3  
 if argc == 0:  
 values\_list += ";"  
 else:  
 values\_list += ","  
 db.execute\_query(values\_list)

Когда запрос полностью сформирован (весь цикл пройден, все аргументы обработаны), он отправляется в базу данных с помощью метода execute\_query. Данный метод принимает готовый запрос и отправляет его в бд, в отличие от некоторых других методов, которые принимают на вход данные и формируют из них запрос самостоятельно.

def execute\_query(self, query):  
 try:  
 self.cur.execute(query)  
 self.connection.commit()  
 print("Query executed successfully")  
 except Error as e:  
 print(f"The error '{e}' occurred")

На Рисунке 1 представлен результат выполнения запроса для нескольких строк.

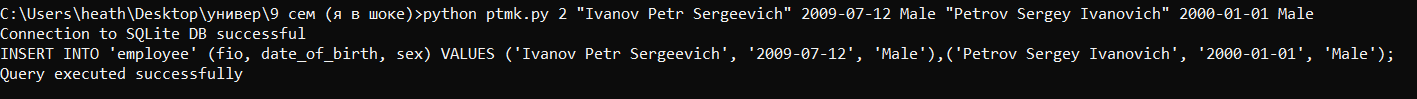


Рисунок 1 — Вставка двух строк

## Вывод строк

Требуется реализовать вывод всех строк справочника сотрудников, с уникальным значением ФИО+дата, отсортированным по ФИО. Вывести ФИО, Дату рождения, пол, кол-во полных лет. Для реализации данного запроса был использован метод группировки одинаковых строк GROUP BY. Для тестирования в таблицу было вставлено несколько одинаковых строк и несколько строк, в которых отличается только ФИО или только дата (Рисунок 2).

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 2 — Данные для тестирования

На Рисунке 3 представлен результат выполнения запроса, также ниже представлен сам запрос:

SELECT \*, (date('now') - date\_of\_birth) FROM employee GROUP BY fio, date\_of\_birth ORDER BY fio;

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 3 — Результат работы в режиме 3

## Автоматическое заполнение

Требуется заполнить автоматически 1000000 строк справочника сотрудников. Распределение пола в них должно быть относительно равномерным, начальной буквы ФИО также. Добавить заполнение автоматически 100 строк в которых пол мужской и Фамилия начинается с "F". У класса необходимо создать метод, который пакетно отправляет данные в БД, принимая массив объектов.

Для отправки пакета данных использовался уже упомянутый выше в пункте 1.2 метод execute\_query. Для генерации большого набора данных рассматривались следующие варианты:

1. Использовать онлайн генератор. Данный инструмент сразу создает файл типа sql с кодом и позволяет импортировать его в бд. Можно даже выбрать, что необходимо сгенерировать имена, пол, возраст и тд. Проблема: имена генерируется по английскому образцу, то есть имеется только имя и фамилия без отчества. Также непонятно, как установить первую букву в фамилии фиксированной величиной.
2. Сгенерировать данные методами СУБД. Проблема: ФИО будет представлено в виде хэша, а также непонятно, как сгенерировать пол (предполагаем, что их только два и они задаются адекватными словами male и female). Также было непонятно, как сгенерировать строку ФИО, которая начиналась бы с определённой буквы.
3. Сгенерировать случайные данные средствами Python. Здесь строка также может быть представлена только непонятным набором букв, зато можно использовать метод random.choice(['Male', 'Female']) для выбора пола. Также для генерации фамилии, начинающейся на букву “F”, необходимо просто выполнить "F" + randomword().

В итоге данные были сгенерированы средствами языка Python. На Рисунке 4 представлен результат вставки 100 записей, в которых пол мужской и Фамилия начинается с "F".

Изображение выглядит как снимок экрана, текст

Автоматически созданное описание

Рисунок 4 — Автоматическое заполнение таблицы

Также на Рисунке 5 представлен размер файла базы данных, по которому можно судить, что миллион записей успешно вставились.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок 5 — Размер файла после большой вставки

## Выборка

Требуется сделать выборку из таблицы по критерию: пол мужской, Фамилия начинается с "F". Для этого был сформирован следующий запрос:

SELECT \* FROM employee WHERE sex = 'Male' AND fio LIKE 'F%';

Результат выполнения представлен на Рисунке 6.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 6 — Часть вывода программы

Также необходимо было сделать вывод времени запроса (Рисунок 7). После нескольких запусков было подсчитано, что средним временем выполнения является значение 0,0010 миллисекунд. Для вывода времени использовался инструмент Python timeit.

execution\_time = timeit.timeit(lambda: self.cur.execute(select\_query), number=1)

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок 7 — Вывод времени выполнения

## Оптимизация запроса

Требуется произвести оптимизацию базы данных или запросов к базе для ускорения выполнения пункта 5. Классический способ оптимизации — это конечно же индексирование. Добавим индекс на столбец ФИО:

CREATE INDEX IF NOT EXISTS fio\_ind ON employee(fio);

При создании индексов важно учитывать два аспекта:

1. Выборка select станет быстрее, а вот вставка и обновление данных значительно медленнее, поэтому не рекомендуется делать индексы в таблицах с частым обновлением;
2. Скорость поиска достигается за счет относительной уникальности индексируемых данных. В рассматриваемом случае поле ФИО можно считать уникальным, поскольку данные в нем представлены 32-буквенными строками, сгенерированными случайным образом. В принципе в обычной жизни ФИО людей в некотором группе тоже достаточно уникальны, поэтому данный столбец можно использовать для индексирования.

На Рисунке 8 представлено 5 замеров времени до оптимизации, среднее время составило 0,0010 мс. На Рисунке 9 представлено 5 замеров времени после оптимизации, среднее время составило 0,0003 мс.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 8 — Замер времени ДО оптимизации

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 9 — Замер времени ПОСЛЕ оптимизации

Дополнительные возможности для оптимизации: если заранее известно, что при выборке будет учитываться возраст (или дата рождения) сотрудника, то можно также провести секционирование таблицы. Например, создать 3 таблицы, в которых были бы люди в трех возрастных категориях. И если нужно выполнить запрос, аналогичный тому, что в пункте 5, но при этом возраст человека должен быть до 30 лет, то поиск будет осуществляться сразу по подтаблице, соответственно даже при полном сканировании будет просмотрено в 3 раза меньше строк. Однако стоит отметить, что универсальных оптимизаций не существует, стоит отталкиваться от природы данных в таблице, их количества, специфике запросов к таблице и т.д.