**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ» (ДГТУ)

Факультет «Информатика и вычислительная техника»

Кафедра «Кибербезопасность информационных систем»

**Лабораторная работа № 3**

на тему «Анализ производительности БД PostgreSQL»

|  |
| --- |
| Выполнил: студент группы ВКБ43 |
| Ковалев Данил Петрович |
| (Фамилия, имя, отчество) |
| Проверил: |
|  |
| (Фамилия, имя, отчество) |

Оглавление

[Цель 3](#_Toc182337876)

[Задание 3](#_Toc182337877)

[Приложение А. Листинг кода: 10](#_Toc182337878)

[Класс Main (с точкой входа): 10](#_Toc182337879)

[Класс exceptions.NoConnectionException: 11](#_Toc182337880)

[Класс pgsql.DBManager: 11](#_Toc182337881)

[Класс 13](#_Toc182337882)

# **Цель**: изучить особенности работы с базой данных PostgreSQL. Провести временной анализ выполнения запросов.

# **Задание 1.** Создать схему istudents.

Для выполнение данной лабораторной работы буду использовать систему контейнирзации Docker, которая позволяет разворачивает любое ПО за минимальное количество телодвижений. В моем случае был написан такой docker-compose, который представлен на рисунке 1.

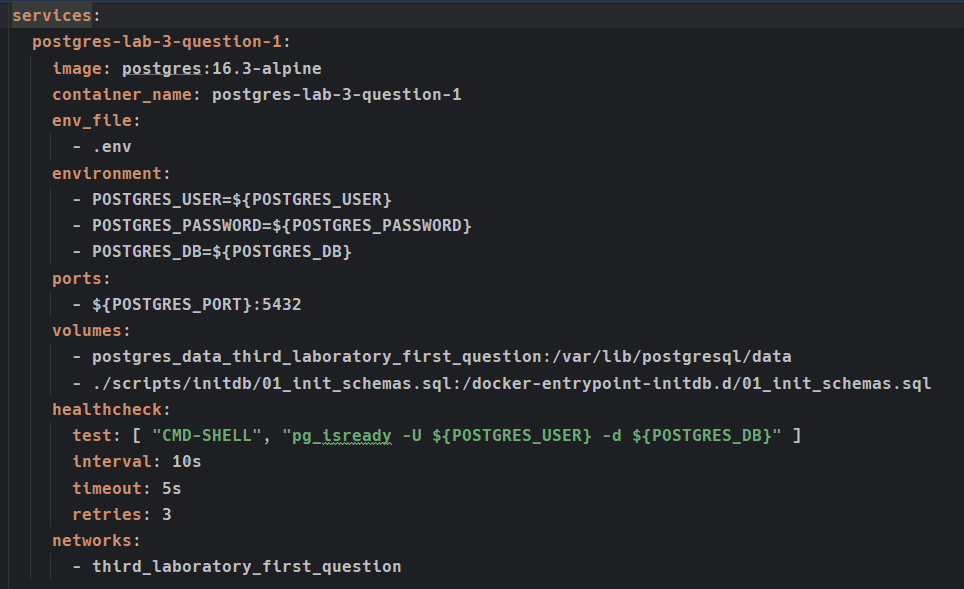


Рисунок 1 – файл для создания PostgreSQL базы данных через Docker compose

Для создания схемы был написан такой код с использованием SQL. Мною написанный скрипт прогоняется, как миграция для базы данных через систему Docker. Код представлен на рисунке 2.



Рисунок 2 – код создания схемы

**Задание 2**. Восстановить БД из бэкапа (разархивировать, восстановить в pgAdmin) в схему istudents.

Через pgAdmin не получится автоматизировать восстановление бэкапа, так как каждый раз придется пользователю вручную делать такие действия. Было принято написать отдельный Docker контейнер в Docker compose, который применяет миграции к базе данных, когда она была уже успешно запущена. На рисунке 2 представлен контейнер, который применяет бэкап к нашей базе данных.



Рисунок 2 – контейнер для применения бэкапа базы данных

Мною был написан скрипт для автоматизации подключения к базе данных и применения бэкапа. Данный код был написан на языке Bash, сниппет представлен на рисунке 3. Если говорить по коду, то в моем случае устанавливаются параметры для подключения к базе данных. Далее идут проверки, что база данных готова к подключению и применению миграций. В конце применяется дамп в указанную схему, после чего пишется сообщение, что миграция была применена успешно.

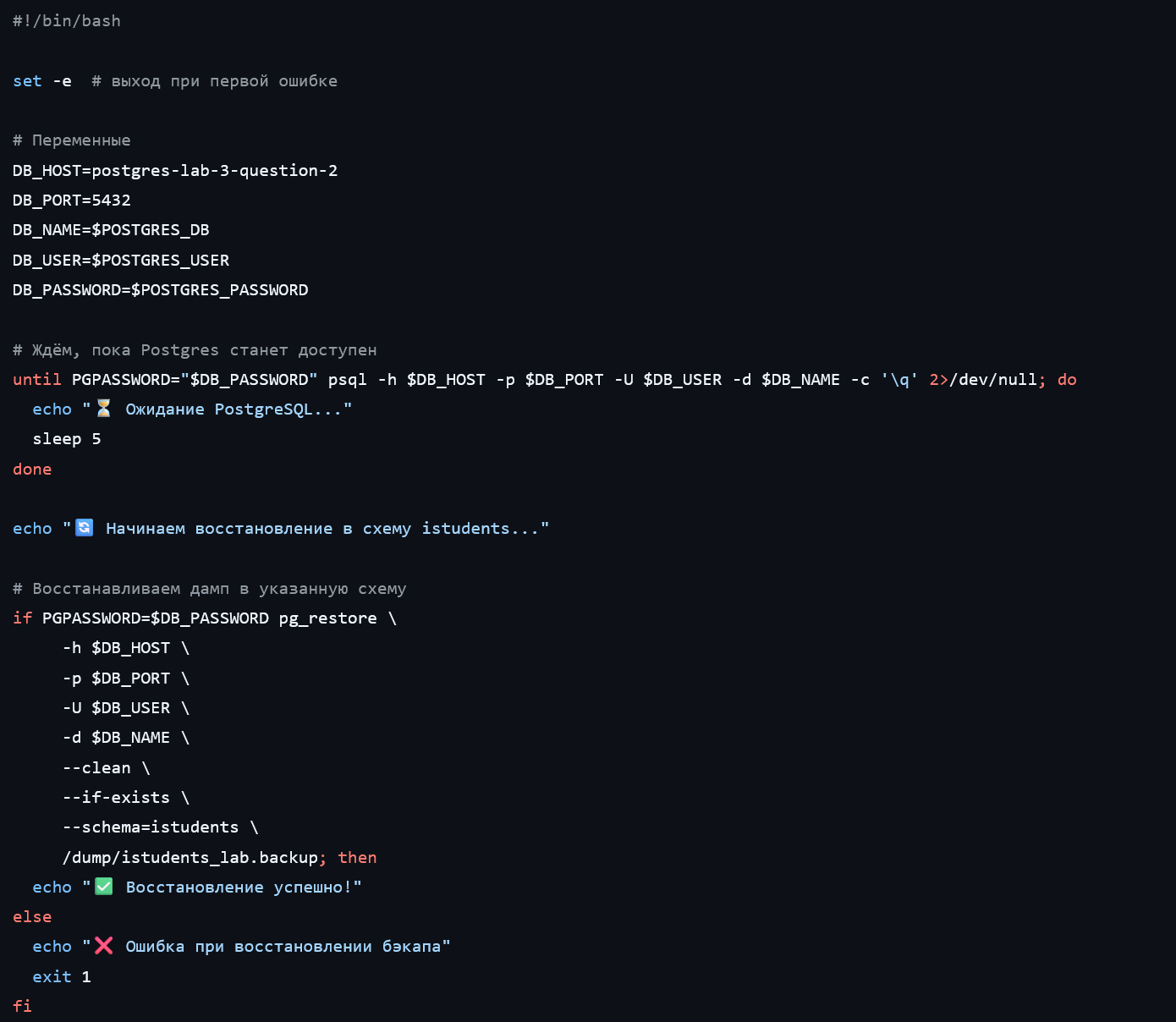


Рисунок 3 – скрипт для восстановления бэкапа для базы данных

Теперь проверим, что бэкап удачно применился, для этого просмотрим какие таблицы появились в базе данных. Для просмотра таблиц я использую Pycharm Professional. Результат представлен на рисунке 4.

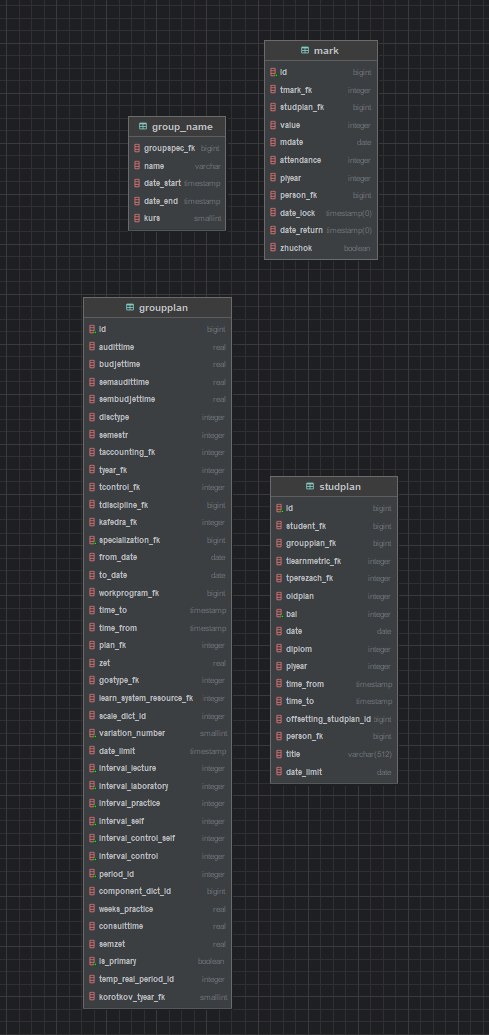


Рисунок 4 – таблицы в схеме istudents

**Задание 3**. Замерить скорость выборки и модификации данных в таблице istudents.mark:

Для выполнения данного задания было написано небольшое Python приложение, которое позволило автоматизировать выполнение данного задания. Были использованы библиотеки: sqlalchemy, uv, asyncpg, pandas.

**Пункт a**. Выполнить выборку данных (30 раз с различными условиями, фильтрация по полю **tmark\_fk** – условия могут циклически повторяться) – замерить время – получим ряд t1;

Код для выполнения данного задания представлен ниже на рисунке 5. Здесь используется асинхронная обработка запросов, что повышает производительность.

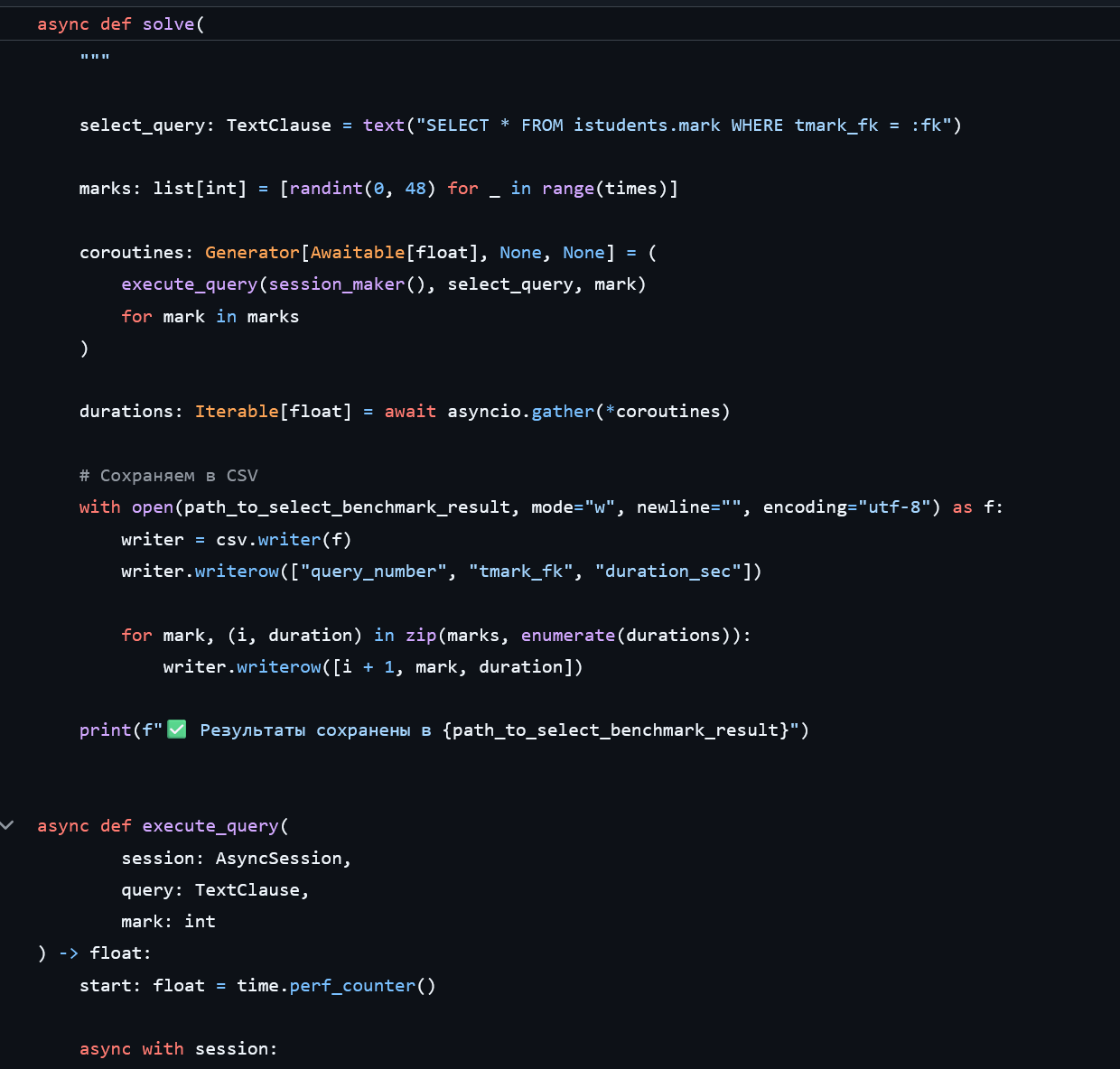


Рисунок 5 – код для выполнения задания

В результате получились такие результаты, которые представлены на рисунке 6. Здесь в столбцах описан номер запроса, какой tmark\_fk и сколько времени заняло выполнение запроса.

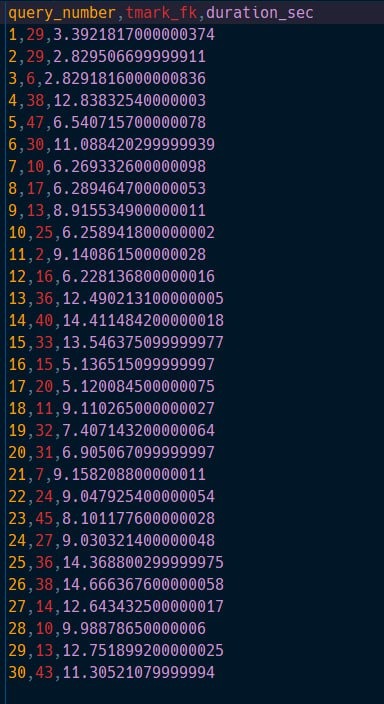


Рисунок 6 – статистика для задания

**Пункт б**. Выполнить вставку данных (30 раз различные данные) – замерить время – получим ряд t2;

Для выполнения данного задания был написан код на языке программирования Python. Сниппет можно увидеть на рисунке 7. Код был написан в асинхронном стиле с использованием библиотеки sqlalchemy с асинхронным драйвером asyncpg. Для максимальной нагрузки на базу данных был использован механизм – asyncio.gather, который позволяет запустить множество корутин параллельно.



Рисунок 7 – скрипт для выполнения задания

После работы скрипта получился csv файл, содержащий время измерения запросов. Содержимое данного файла прилагается на рисунке 8.

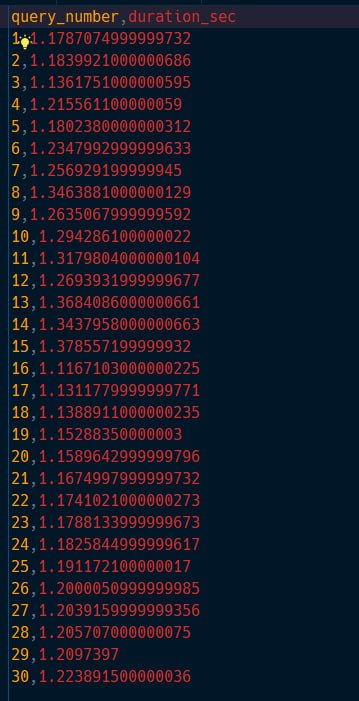


Рисунок 8 – содержимое insert\_benchmark.csv

**Пункт c**. Изменения t1 и t2 представить на графиках – объяснить их форму;

Для выполнения данного задания использовалась библиотека matplotlib, которая позволяет строить графики функций в зависимости от данных. В моем случае все измерения были записаны в csv файл, что позволяет измерить время выполнения программы. Сниппет для выполнения данного задания представлен на рисунке 9.



Рисунок 9 – код для построения графиков.

После выполнения программы можно увидеть 2 разных графика. В моем случае они вышли такими, как представлено на рисунках 10 и 11.



Рисунок 10 – график производительности select\_benchmark

Почему же график на рисунке 10 настолько прыгающий? Все дело в том, что база данных кэширует запросы. Если запрос дублируется, то база данных возвращает результат с кэша, поэтому видим такой прыгающий график.

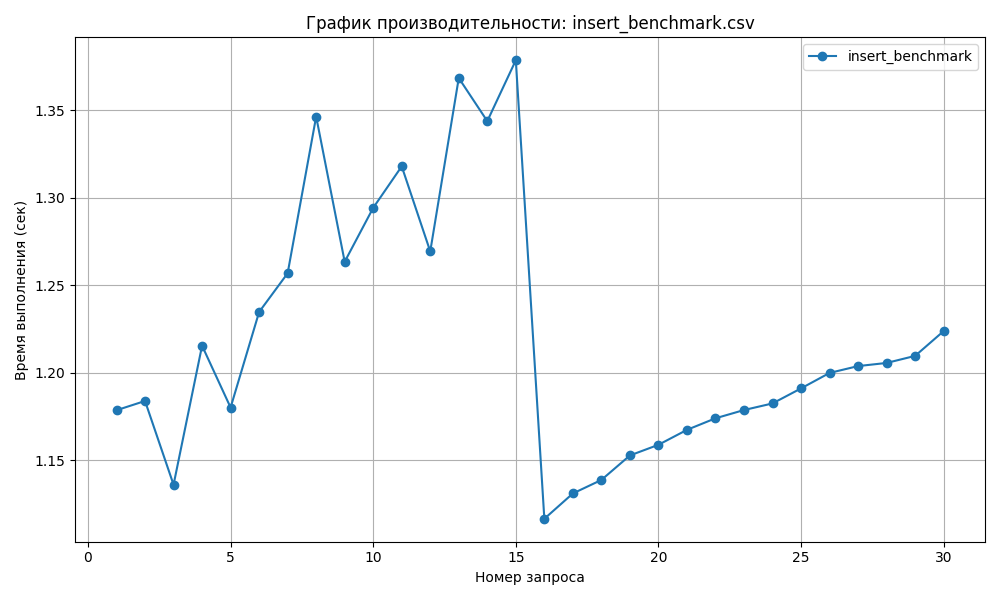


Рисунок 11 – график производительности insert\_benchmark

Почему же идет на вырост, а потом на спад? Причина - Конкуренция транзакций. Я использую асинхронные запросы, которые выполняются параллельно. Это может привести к следующим проблемам:

* Блокировки: если несколько вставок пытаются модифицировать одну и ту же таблицу или строки, PostgreSQL может блокировать некоторые запросы, пока другие завершаются. Это вызывает "простой" и увеличивает время выполнения.
* Contention на ресурсы: параллельные запросы конкурируют за CPU, память и I/O дисков. Это особенно заметно на слабых машинах или в Docker-контейнерах с ограниченными ресурсами.

### **Пункт d**. Для полученных рядов (t1, t2) посчитать основные статистические показатели - мат. ожидание, дисперсию, СКО и оценить полученные результаты.

Для подсчета основных характеристик была написана программа на языке программирования Python. Сниппет данной программы представлен на рисунке 12.



Рисунок 12 – код для отображения основных статистических показателей

При запуске данного кода получился такой вот результат, который представлен на рисунке 13.

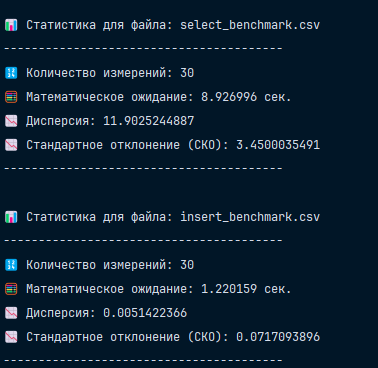


Рисунок 13 – основные математические характеристики

По результатам измерения можно сказать, что база данных достаточно быстро отрабатывает на вставку. Максимальное время вставки – 1с. Селективность достаточно медленная. Избежать таких визуальных задержек можно с помощью использования индексов и асинхронности.

**Задание 4**. Создать в таблице istudents.mark первичный ключ (id) и создать индекс для поля tmark\_fk.

Для выполнения данного задания был написан скрипт, который применяется, как миграция в docker compose для нашего контейнера, содержащего базу данных. Скрипт представлен на рисунке 14.

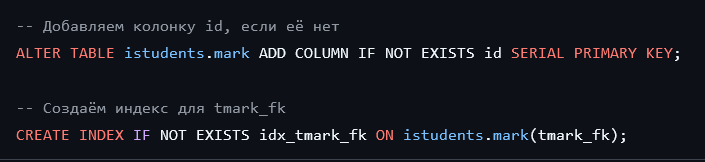


Рисунок 14 - создание первичного ключа и индекса для поля tmark\_fk таблица mark

**Задание 5**. Повторить произведенные в п.3 замеры, построить графики, произвести статистическую оценку полученных данных. Как изменились временные показатели?

Для измерения производительности используется те же самые программы, которые были описаны в прошлых заданиях. Представлю ниже измерения, которые получились после того, как был добавлен индекс. На рисунке 15 представлен csv файл, содержащий временные метки по выполнению select запросов.

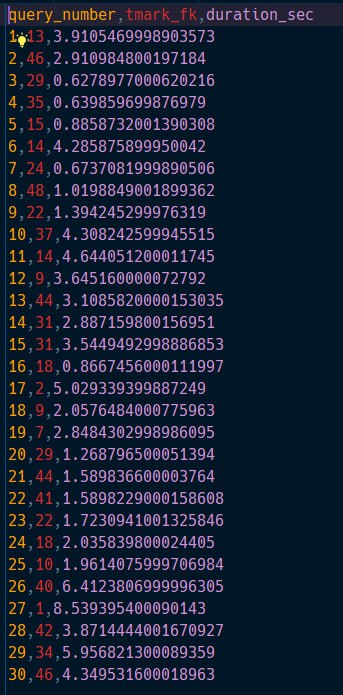


Рисунок 15 – результат тестов селективности с индексом

Теперь посмотрим какие показатели получились для вставки, результаты тестов приложены на рисунке 16.

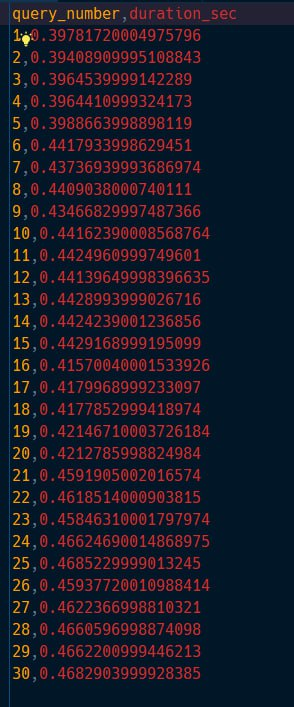


Рисунок 16 – результат тестов вставки с индексом

Ниже представим графики, которые получились в данных тестах, они представлены на рисунке 17, 18.

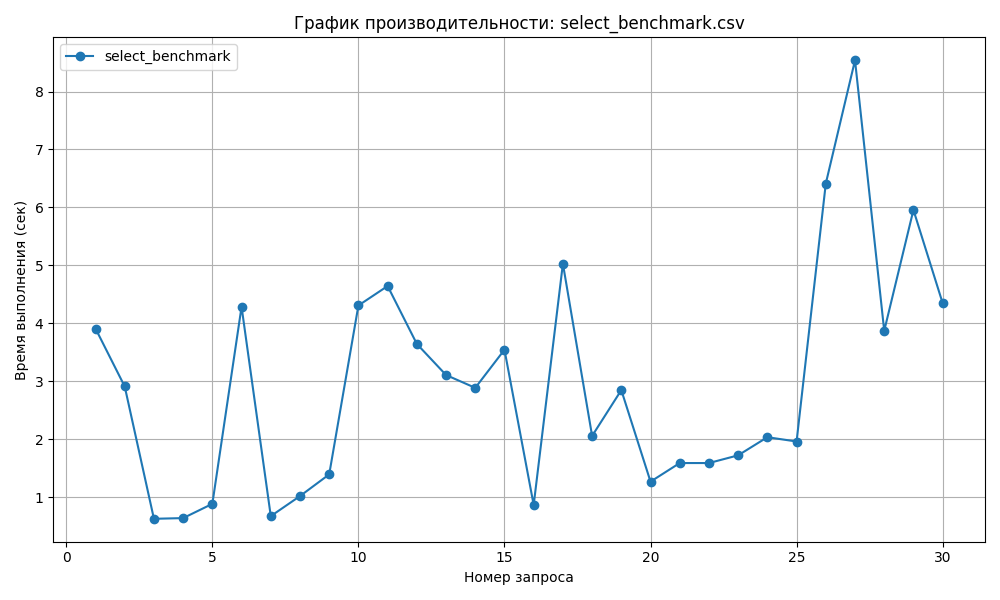


Рисунок 17 – график производительности для select

При добавлении индекса хорошо заметно, что среднее время запроса уменьшилось. Скачкообразное поведение обусловлено опять-таки поведением из-за кэширования в базе данных. Рассмотрим теперь график производительности для вставки данных, который представлен на рисунке 18.

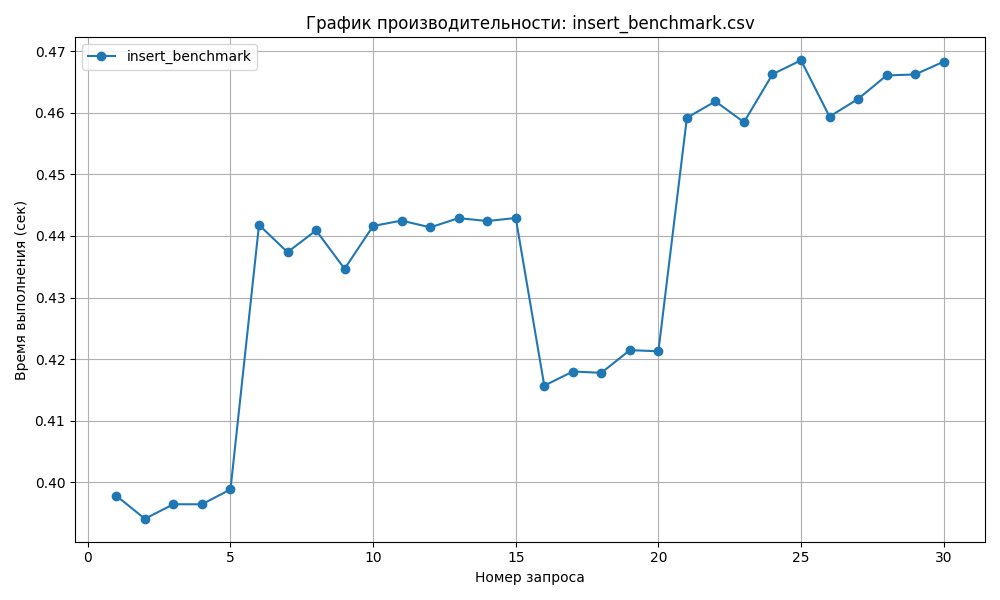


Рисунок 18 – график производительности для insert

Видно, что при добавлении индекса поведение вставки улучшилось, что является скорее аномалией, чем нормой, потому что индекс – это отдельная структура данных, которая повышает скорость считывания, но остальные характеристики ухудшает. Например, каждая вставка заставляет делать заново балансировку дерева для хранения индексов, указывающих на строки базы данных. Был найден такой комментарий, подтверждающий наши слова. Он представлен на рисунке 19.

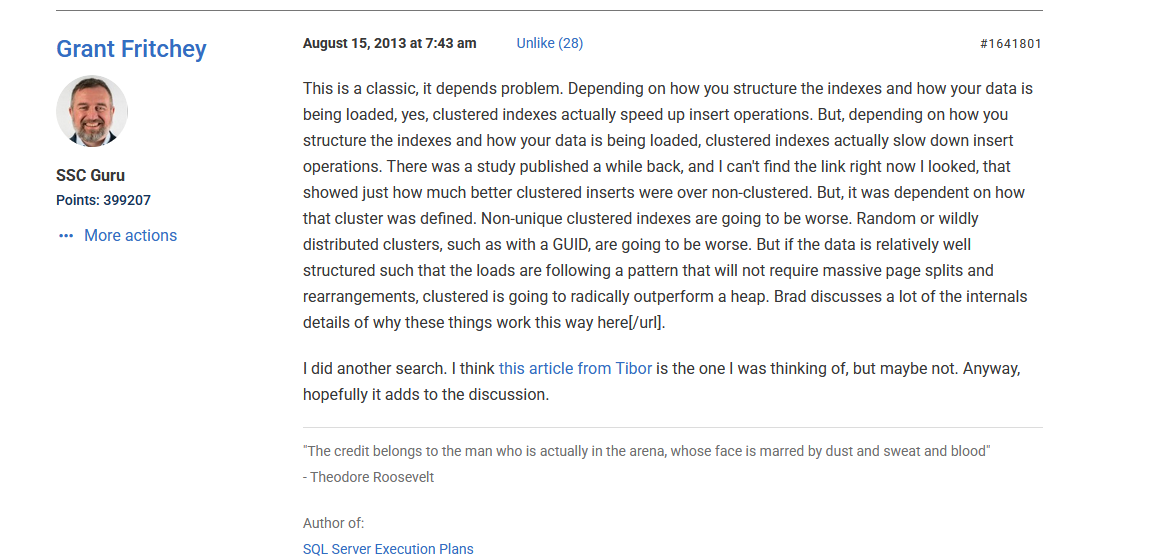


Рисунок 19 – комментарий по поводу аномалии

В результате получились такие вот статистические характеристики, которые представлены ниже на рисунке 20.

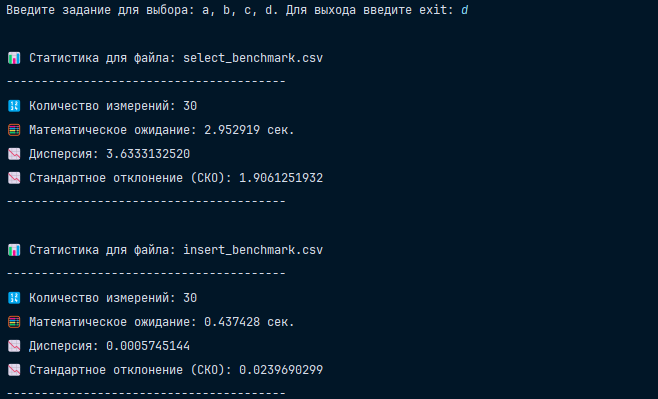


Рисунок 20 – статистические измерения после добавления индекса

**Задание 6**. Написать приложение, которое для каждого значения атрибута «год действия плана» (атрибут plyear в таблице mark) считает количество положительных оценок в таблице (положительными считаются оценки, у которых (bal>40). При этом должен быть реализован следующий алгоритм: Выбор всех возможных значений атрибута plyear из таблицы (например, с помощью запроса SELECT DISTINCT plyear FROM istudents.mark); Организовать цикл по результатам запроса в п.а и для каждого значения plyear независимо считать количество положительных оценок (например, с помощью запроса SELECT count(id) FROM istudents.mark WHERE bal > 40 and plyear = 1); Вывести полученные данные (по каждому значению plyear) пользователю, вывести время, за которое значения были получены.

Для написания приложения был выбран язык Python с следующими библиотеками: sqlalchemy, asyncpg, pandas, numpy. Структура проекта представлена на рисунке 21.

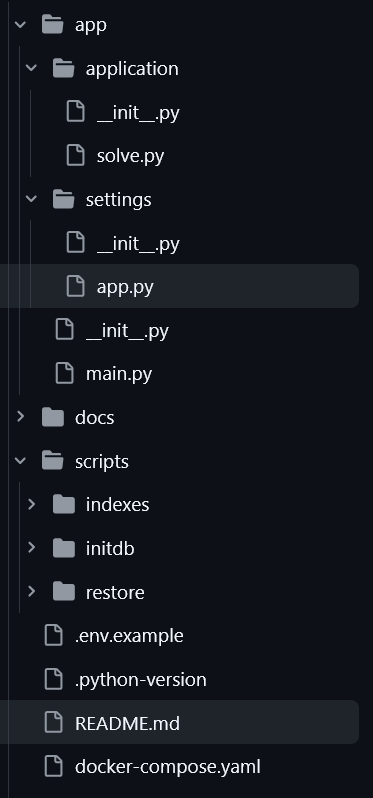


Рисунок 21 – структура приложения

В результате выполнения программы получилось то, что представлено на рисунке 22. Обращаем внимание, что при написании программы использовался асинхронный подход для взаимодействия с базой данных, поэтому общее время выполнения очень маленькое.

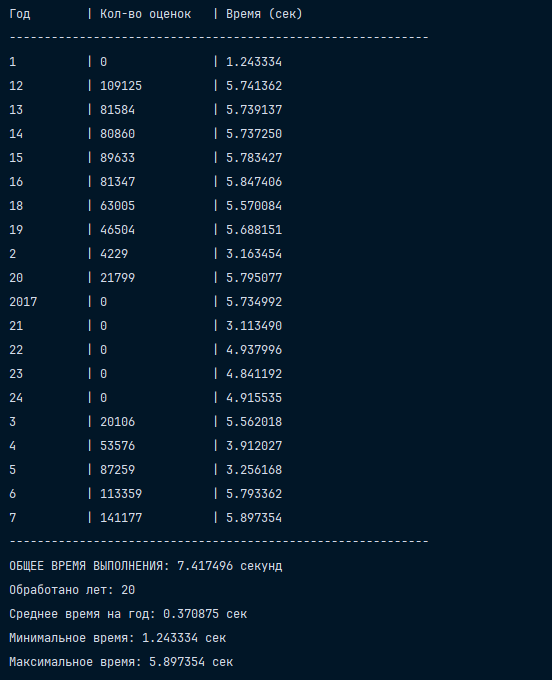


Рисунок 22 – результаты измерений при выполнении программы

**Задание 7**. Создать индекс по полю plyear и повторить действия, описанные в п.6 – сравнить полученные временные показатели и результаты;

Для выполнения данного задания был написана команда языке программирования SQL. Данная команда применяется, как миграция к базе данных при запуске приложения в Docker.

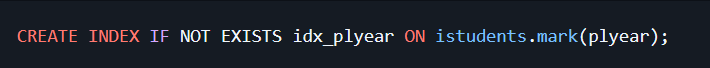


Рисунок 23 - запрос на создание индекса по полю plyear

Запустим теперь скрипт и посмотрим, насколько повысилась производительность. Результат тестов представлен на рисунке 24.

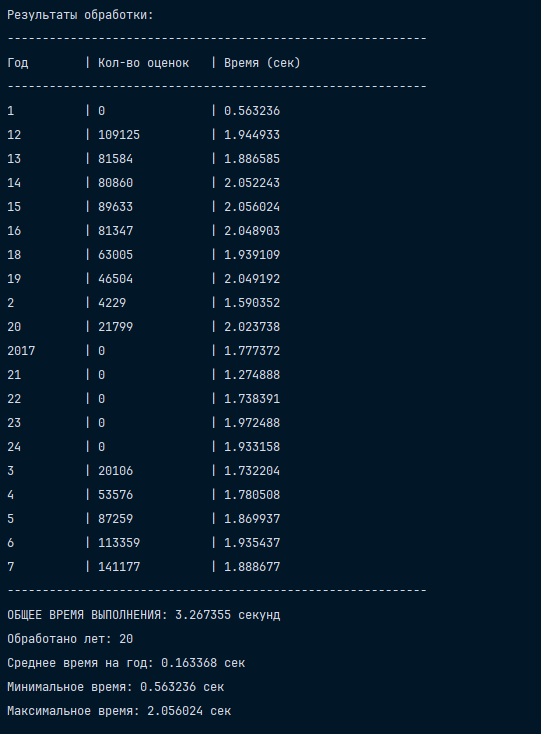


Рисунок 24 – результат тестов после добавления индекса

Можно замтетить, что среднее время запроса уменьшилось в несколько раз, что говорит о том, что применение индексов очень важно для повышения производительности системы.

**Задание 8.** Написать приложение, которое для каждого значения атрибута «год действия плана» (атрибут plyear в таблице mark) считает количество положительных оценок в таблице (положительными считаются оценки у которых (bal>40). При этом должен быть реализован следующий алгоритм: выполнить запрос с группировкой по полю plyear для расчета значений по каждой группе (SELECT plyear, count(id) FROM istudents.mark WHERE bal>40 GROUP BY plyear); Полученные данные и время их получения представить пользователю, сравнить эти показатели с результатами, полученными в п.6, 7;

Для разработки ПО был выбран язык Python с использованием следующих библиотек: sqlalchemy, asyncpg, numpy, pandas. Структура проекта выглядит следующим образом, как представлено ниже на рисунке 25.

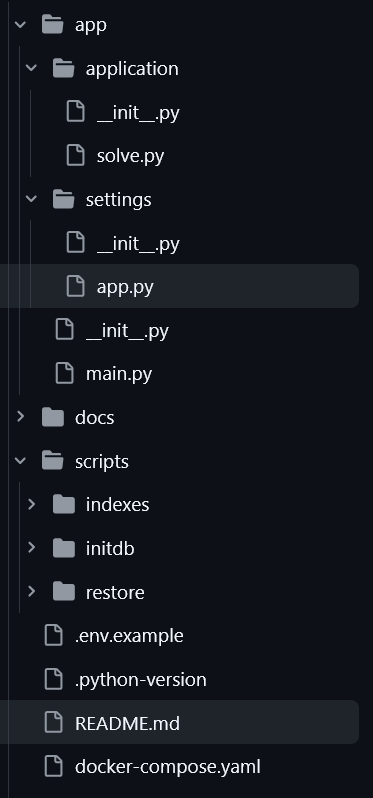


Рисунок 25 – структура проекта для 8 задания

На рисунке 26 представлен результат выполнения запросов. Можно заметить, что скорость выполнения намного выше, чем в пунктах 6 и 7. Это говорит нам о том, что GROUP BY работает быстрее, чем DISTICT + COUNT даже с индексом.

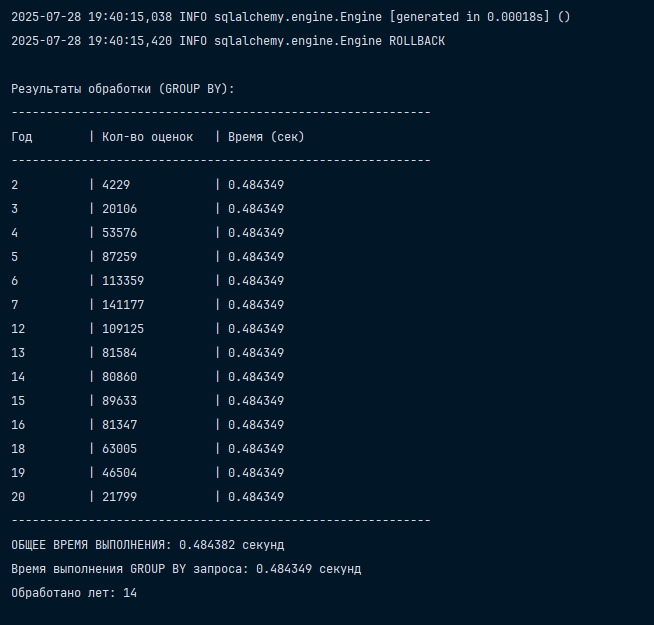


Рисунок 26 - предоставленные пользователю результат выборки и время выполнения запроса

**Задание 9**. Создать первичный ключ в таблице istudents.studplan.

Для создания первичного ключа воспользуемся SQL командой, которая представлена на рисунке 27.

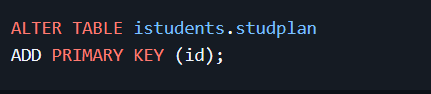


Рисунок 27 – создание первичного ключа в studplan

**Задание 10**. Замерить время, требуемое для выполнение внутреннего объединения двух таблиц (istudents.mark и istudents.studplan) по условию объединения: mark INNER JOIN studplan on mark.studplan\_fk = studplan.id. При этом выбирать **не более 100 записей** из результатов объединения (для этого использовать LIMIT 100). Выполнить запрос 30 раз и каждый раз выбирать разные 100 записей – для этого использовать OFFSET (например, так SELECT \* FROM mark INNER JOIN studplan on mark.studplan\_fk = studplan.id LIMIT 100 OFFSET 900).

Для выполнения был написан скрипт на языке Python с использованием библиотек: numpy, pandas, sqlalchemy, asyncpg. Структура проекта никак не поменялась с прошлых заданий. На рисунках 28, 29 приложу полученные результаты при выполнении задания.

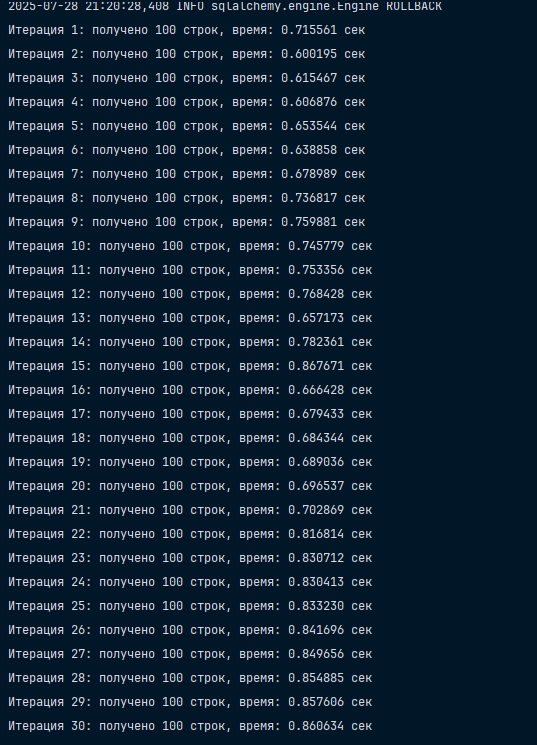


Рисунок 28 – время выполнения каждого запроса

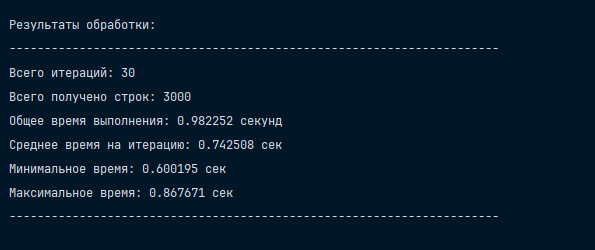


Рисунок 29 – результат обработки тестов

**Задание 11.** Выполнить запросы из п.10 – 30 раз, замерить время и оценить полученные результаты статистически.

В принципе задание от прошлого ничем не отличается, просто нужно было запустить все приложении и также оценить. В результате получилось то, что представлено на рисунке 30.

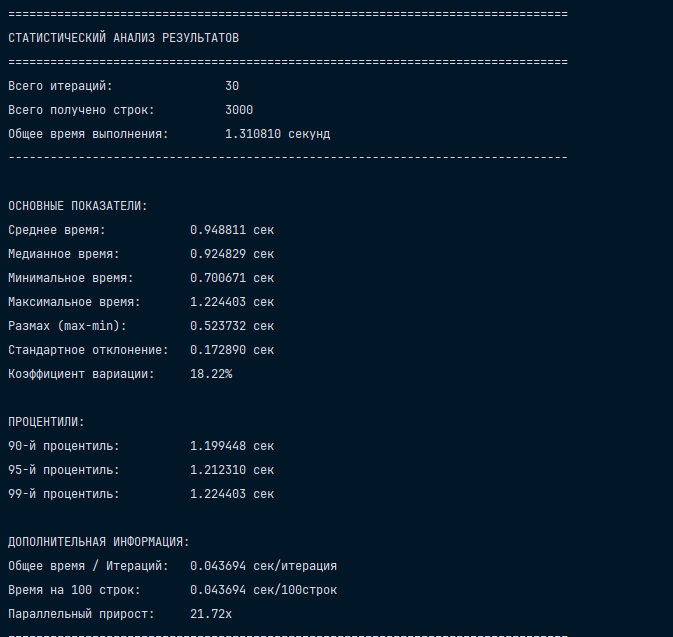


Рисунок 30 – результат обработки тестов

**Задание 12**. Создать внешний ключ в таблице istudents.mark (studplan\_fk -> studplan.id) и для внешнего ключа обязательно **создать индекс**.

Для создания индекса и внешнего ключа был использован SQL скрипт, который представлен на рисунке 31.

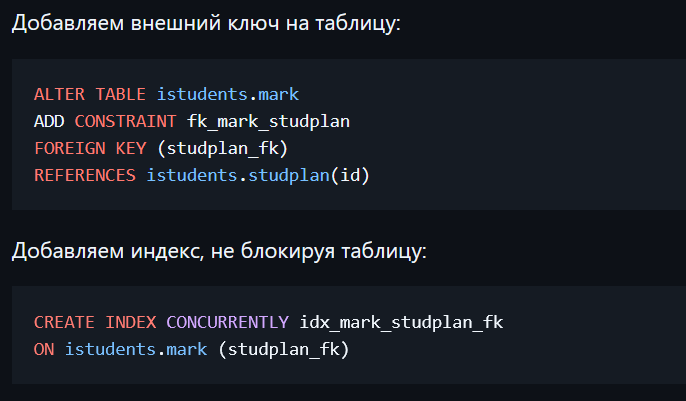


Рисунок 31 – скрипт для выполнения задания 12

**Задание 13**. Повторить запросы из п.10 – 30 раз, замерить время и оценить полученные результаты статистически.

Для запуска тестов использовал тот же самый код, который был до этого описан в прошлых заданиях. Результаты тестов показаны на рисунках 32, 33.

Единственное отличие от прошлого задание – это, что были применены миграции через Docker.

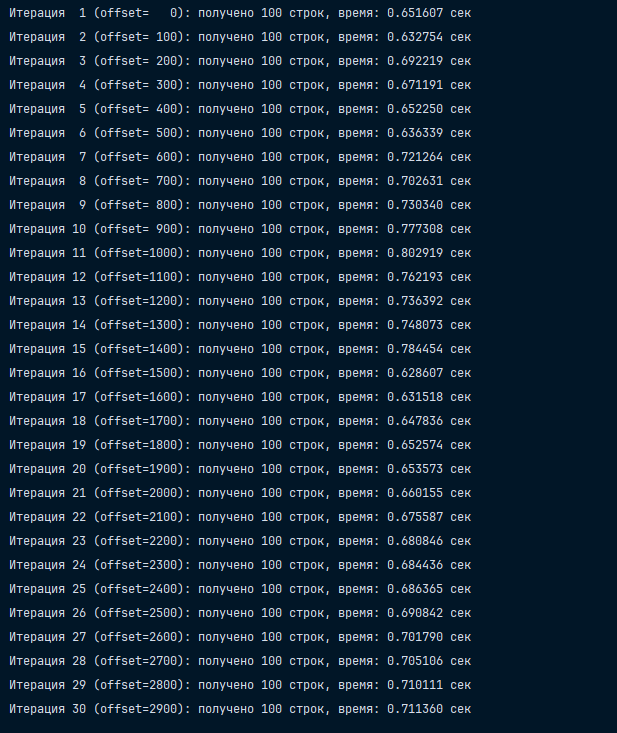


Рисунок 32 – время выполнения каждого запроса

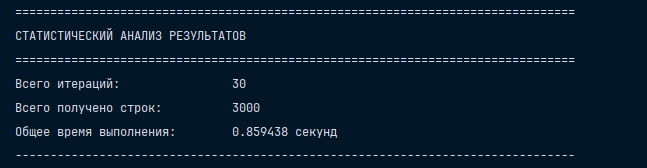


Рисунок 33 – общий анализ

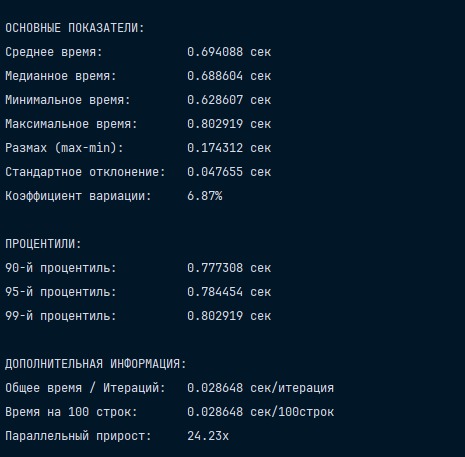


Рисунок 34 – более подробное описание тестов

**Задание 14**. Сравнить результаты из п.11 и п.13: отличаются или нет и почему.

В нашем случае судя по тестам они отличаются. Повышение производительности идет за счет того, что мы используем индексы, которые повышают быстрее находить нужные записи в столбцах у базы данных.