МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ИС

ОТЧЕТ

по учебной практике

Тема: Исследование Clickhouse vs Postgresql

Студентка гр. 1376 Рындыч А.Е.

Преподаватель Цехановский В.В.

Санкт-Петербург
2023

Цель работы.

Научиться работать с PostgreSQL и Clickhouse, провести эксперимент по обработке запроса и сделать вывод о скорости работы перечисленных баз данных.

Задание.

В СУБД Postgresql и Clickhouse создать тестовую таблицу, содержащую 40 колонок строкового и численного типа.

Таблицы должны заполняться 100000 записей скриптом-программой. Каждая запись должна генерироваться путем задания случайных чисел для всех численных полей и случайных строк (из выбранного набора) .

После этого к первой (Clickhouse) и второй (Postgresql) производится обрашение из разрабатываемого приложения, реализующего SQL – запрос (запрос должен быть достаточно сложным, включать до 20 условий на столбцы). Производится замер времени. Эксперименты повторяются 100 раз. После выводится максимальное, минимальное и среднее время выполнения запросов

Основные теоретические положения.

Рассматриваемые в этой работе базы данных являются реляционными базами данных. Данные в реляционной базе данных формируют отношения — двумерные таблицы с информацией о сущностях, т. е. объектах. Строка такой таблицы называется кортежем. Кортежи содержат множество атрибутов одной сущности, категории которых задаются в столбцах.

Для каждого атрибута выделен строго определенный столбец, а каждый столбец может содержать только один тип (или категорию) атрибутов.

Каждая из строк определяет только одну-единственную сущность и содержит уникальный набор его атрибутов.

Таким образом, строки в базе данных не повторяются. Чтобы гарантировать уникальность каждой строки, для нее задается первичный ключ,

своего рода идентификатор, который также используется, когда на кортеж нужно сослаться из другой таблицы, при этом не приводя полного набора атрибутов сущности. Тогда первичный ключ становится внешним ключом. Именно ключи обеспечивают целостность и согласованность данных и отношений.

Первичный ключ позволяет обращаться к кортежам базы данных независимо от того, где физически они расположены, на каком месте, в какой таблице и в каком порядке. Ключи позволяют сортировать, фильтровать, извлекать, обрабатывать и возвращать данные в таблицы без лишних операций: если та или иная сущность встречается в базе данных множество раз, достаточно изменить ее атрибуты в одной таблице (по первичному ключу) — и они обновятся везде, где встречается этот ключ. Кроме того, ключи не позволяют ссылаться на несуществующие данные — а это гарантирует целостность всей базы данных.

В таблице ниже приведены основный характеристики реляционных баз данных (см табл. 1)

Таблица 1 - основный характеристики реляционных баз данных.

Признак	Пояснение
Множество сущностей	Объекты со строго определенным набором атрибутов, с помощью которых они связываются между собой, формируют понятную и простую для восприятия структуру.
Табличный формат	Такой формат гарантирует высокий уровень структурированности с жесткими логическими взаимосвязями, минимальный уровень избыточности данных, их согласованность и целостность.
Язык SQL	SQL является стандартизированным средством общения пользователя с базой данных. Он очень формальный, что делает его удобным и простым в изучении. SQL гарантирует точный результат даже при сложном многоуровневом запросе.

Существует два разных подхода к хранению и организации данных в системах управления реляционными базами данных (RDBMS): хранилища данных, ориентированные на строки и столбцы.

В хранилище данных, ориентированном на строки, данные хранятся и извлекаются построчно, что означает, что все атрибуты конкретной строки хранятся вместе в одном физическом блоке данных. Этот подход оптимизирован для извлечения целых строк данных за один раз и обычно используется в традиционных системах RDBMS.

В большинстве баз данных хранилище располагается построчно: все значения из одной строки таблицы хранятся рядом друг с другом. Базе данных, реализованный построчно, хранящей данные в сотне столбцах, при выполнении запроса лишь по двум из них все равно потребуется загрузить все строки (каждая из которых в данном случае состоит более чем из 100 атрибутов) с диска в оперативную память, выполнить их синтаксический разбор и отфильтровать те, то не удовлетворяют заданным условиям. На это может уйти много времени.

Решением данной проблемы становится идея столбцовых хранилищ: нужно хранить рядом значения не из одной строки, а из одного столбца. Если каждый столбец хранится в отдельном файле, то запросу требуется только прочитать и выполнить синтаксический разбор необходимых запросов столбцов, что может сэкономить массу усилий.

Также преимуществом столбцовых хранилищ является то, что они часто очень хорошо поддаются сжатию. Помимо загрузки с диска только тех столбцов, которые нужно для запроса, можно еще более снизить требования к пропускной способности диска, сжав данные, что также уменьшает время выполнения запроса.

Clickhouse - система управления базами данных, ориентированная на столбцы. Это позволяет анализировать данные, которые обновляются в режиме реального времени. В большинстве случаев он обеспечивает мгновенные

результаты: данные обрабатываются быстрее, чем требуется для создания запроса.

PostgreSQL - мощная объектно-реляционная система баз данных с открытым исходным кодом, ориентированная на строки.

Опираясь на теорию, представленную выше, можно выдвинуть предположение, что при выполнении запроса к Clickhouse и PostgreSQL время работы в первом случае будет быстрее.

Стоит обратить внимание, что эксперимент ориентирован только на операцию чтения. При исследовании операции записи новых данных в хранилище результаты могут отличаться из-за специфики устройства баз данных.

Выполнение работы.

В обеих случаях в самом начале программы было выполнить соединение с базой данных, создать курсор (или же клиент в случае с Clickhouse, то есть специальный объект который делает запросы и получает их результаты). Затем с помощью функции класса *DataBase create* была создана таблица с уникальной колонкой «*id*» и другими 40 текстовыми и строковыми колонками, которые в дальнейшем были заполнены случайно сгенерированными значениями. Заполнение происходит с помощью функции *filling*, создается 100000 записей. С помощью функции *doQuery*, используя простой SQL-синтаксис запрос *SELECT*, с наложенными на него условиями *where, order by, limit*. Также была реализована функция вывода полученных результатов запроса на консоль *printResults*.

Суть эксперимента заключается в измерении и сравнении времени выполнения запроса к базе данных. Количество попыток – 100, результаты, выводящиеся в консоль – худшее, лучшее и среднее время.

Стоит обратить внимание на то, что в данной работе были также рассмотрены два случая проведения эксперимента: с подключением к базе данных каждый раз из ста попыток эксперимента и с единоразовым

подключением. Это было сделано в связи с предположением о том, что кэширование базы данных (сохранение результата запроса) может повлиять на результаты измерений. В случае с многократным подключением к базе данных кэширование избегается. Это выполнено в функциях *withoutCachingOption()* и *withCachingOption()*. Следует сразу добавить, что данный факт на измерения повлиял незначительно.

Результаты работы программы Postgres.py см на рис. 1, результаты работы программы Clickhouse.py см на рис. 2.

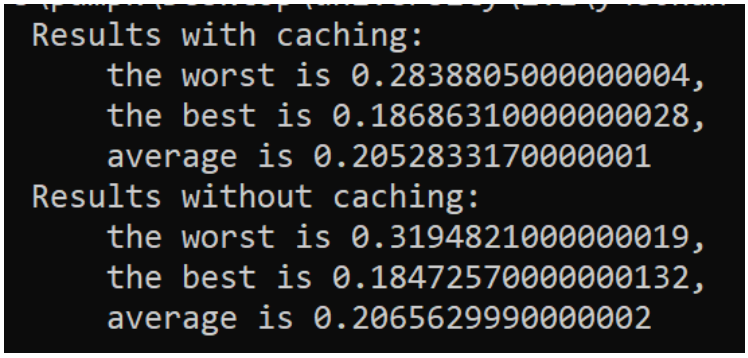

Рисунок 1 – результаты работы программы Postgres.py

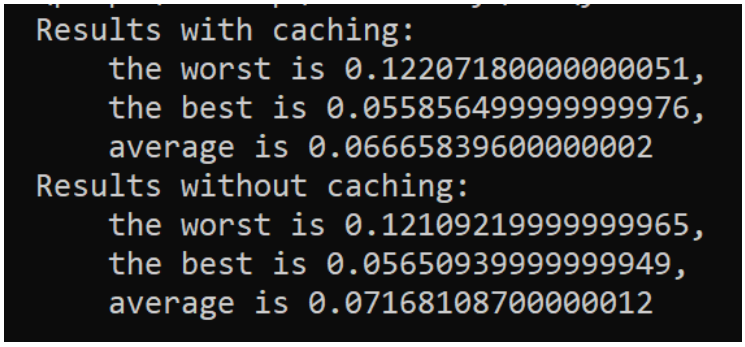

Рисунок 2 - результаты работы программы Clickhouse.py

Для удобства восприятия также результаты исследования приведены в виде графика (см рис. 3-4).

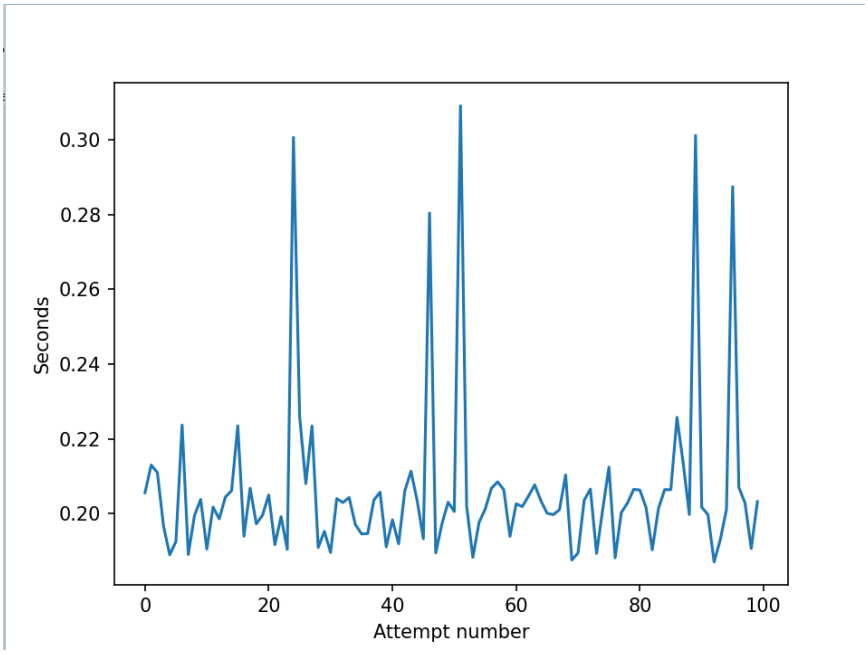

Рисунок 3 – результаты работы программы Postgres.py

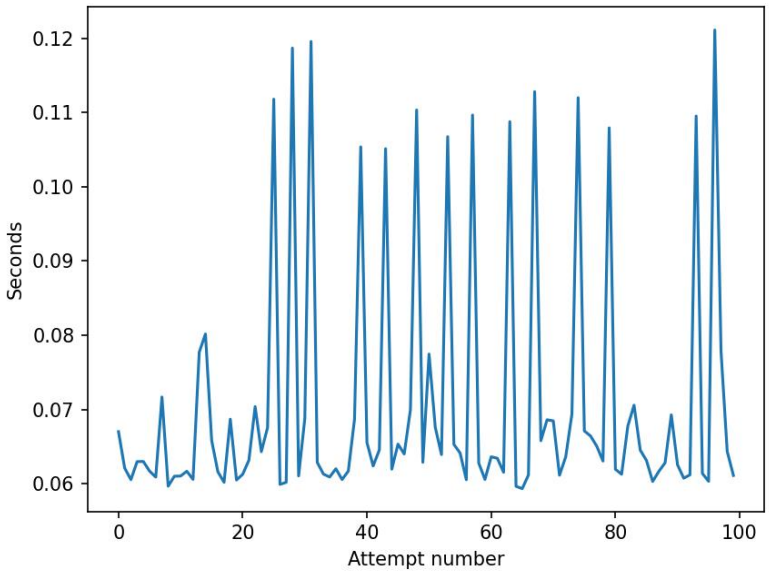

Рисунок 4 - результаты работы программы Clickhouse.py

Выбросы на графике не так велики в абсолютном значении времени и могут быть объяснены спецификой работы операционной системы. В конечном счете эти выбросы не влияют на итоговую картину.

Разработанный программный код см. в приложении А или в репозитории по ссылке https://github.com/Pumpkin-A/DataBaseResearch .

Выводы.

В ходе выполненной работы было выполнено знакомство с базами данных PostgreSQL и Clickhouse.

Также был проведен эксперимент по обработке запроса на операцию чтения, в результате которого выдвинутая теория в ходе исследования была подтверждена на практике. Время работы реализованной через столбцовую ориентацию базы данных Clickhouse при выполнении операции чтения действительно меньше времени работы PostgreSQL – строкового реляционного хранилища.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. «Высоконагруженные приложения. Программирование, масштабирование, поддержка» Мартин Клеппман.
2. Статья «Реляционные базы данных» на Yandex Cloud https://cloud.yandex.ru/docs/glossary/relational-databases
3. Статья «SQL запросы» https://habr.com/ru/articles/480838/
4. Psycopg 2.9.6 documentation https://www.psycopg.org/docs/usage.html
5. Python Integration with ClickHouse Connect https://clickhouse.com/docs/en/integrations/python

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: postgres.py

```
import psycopg2
import random
import string
import time
import matplotlib.pyplot as plt

#функция генерации случайной строки для заполнения столбцов типа str
def generate_random_string(length):
    letters = string.ascii_lowercase
    rand_string = ''.join(random.choice(letters) for i in range(length))
    return rand_string

#класс с функционалом работы с базой данных
class DataBase:
    def __init__(self, cursor): #конструктор класса
        self.cursor = cursor

    def create(self, cursor): #создание таблицы
        #SQL запрос создания таблицы postgresTable, если она еще не существует
        #с начальным заданным уникальным столбцом id
        cursor.execute("CREATE TABLE IF NOT EXISTS public.postgresTable (id int NOT NULL GENERATED ALWAYS AS IDENTITY)")
        intColumnsNames = ['a_int', 'b_int', 'c_int', 'd_int',
                           'e_int', 'f_int', 'g_int', 'h_int',
                           'i_int', 'j_int', 'k_int', 'l_int',
                           'm_int', 'n_int', 'o_int', 'p_int',
                           'q_int', 'r_int','s_int', 't_int']
        strColumnsNames = ['a_str', 'b_str', 'c_str', 'd_str',
                           'e_str', 'f_str', 'g_str', 'h_str',
                           'i_str', 'j_str', 'k_str', 'l_str',
                           'm_str', 'n_str', 'o_str', 'p_str',
                           'q_str', 'r_str','s_str', 't_str']
        #SQL запрос добавления столбцов
        for i in range (len(strColumnsNames)):
            cursor.execute("ALTER TABLE public.postgresTable ADD COLUMN IF NOT EXISTS {name} varchar NULL".format(name = strColumnsNames[i]))
        for i in range (len(intColumnsNames)):
            cursor.execute("ALTER TABLE public.postgresTable ADD COLUMN IF NOT EXISTS {name} integer NULL".format(name = intColumnsNames[i]))

    def filling(self, cursor): #функция заполнения таблицы
```

```
            for i in range (100000):
                allValues = ()
                for i in range (20):
                    allValues                                    +=
(generate_random_string(random.randint(1, 10)),) #заполнение рандомными строковыми значениями
                for i in range (20):
                    allValues += (random.randint(0, 500000),) #заполнение случайными целочисленными значениями
                # Делаем INSERT запрос к базе данных, используя обычный SQL-синтаксис
                cursor.execute("""insert into postgresTable (a_str,
b_str, c_str, d_str,
                                    e_str, f_str, g_str, h_str,
                                    i_str, j_str, k_str, l_str,
                                    m_str, n_str, o_str, p_str,
                                    q_str, r_str,s_str, t_str,
                                    a_int, b_int, c_int, d_int,
                                    e_int, f_int, g_int, h_int,
                                    i_int, j_int, k_int, l_int,
                                    m_int, n_int, o_int, p_int,
                                    q_int,  r_int,s_int,  t_int)  values
(%s,%s,%s,%s,%s,

            %s,%s,%s,%s,%s,

            %s,%s,%s,%s,%s,

            %s,%s,%s,%s,%s,

            %s,%s,%s,%s,%s,

            %s,%s,%s,%s,%s,

            %s,%s,%s,%s,%s)""", allValues)

        def doQuery(self, cursor): #функция запроса (операции чтения)
            # Делаем SELECT запрос к базе данных, используя обычный SQL-синтаксис
            cursor.execute("""
                        SELECT id, a_str, o_str, k_str, a_int, f_int,
t_int
                        FROM postgresTable
                        WHERE (((a_int > 5000) and (b_int % 10 = 5) and
(f_int/k_int > 135)
                            and (t_int % 100 > 2) and (i_int + j_int +
n_int + h_int < 40000000)) or
                            ((a_str not in ('qwert', 'asd', 'aaaaa'))
and (o_str in ('ffff', 'f', 'fff', 'ff', 'fffff'))
                            and (e_str not in ('poi', 'fdgdgdf',
'fff'))) or
                            ((k_str in ('aa')) and (j_str not in ('a',
'f', 'k'))))
```

```
                                and ((id > 30) or (id < 80000) or (id % 10
= 9) or (id = 17) or (id % 2 = 8))
                            ORDER BY a_int, id
                            LIMIT 40
                            """
                            )
                results = cursor.fetchall() #получение результатов запроса
                return results


            def printResults(self, cursor): #вывод результатов запроса в консоль
                results = cursor.fetchall()
                for i in range (len(results)):
                    print(results[i])

        def withoutCachingOption():
            results = [] #результаты времени
            queryResalts = [] #в массив будут помещены результаты запросов
            for i in range(100):
                #соединение с базой данных
                with psycopg2.connect(dbname='postgres', user='postgres',
                                      password='asmrt122',
host='localhost') as conn:
                    #получение курсора
                    with conn.cursor() as cursor:
                        dataBase = DataBase(cursor)
                        # dataBase.create(cursor)
                        # dataBase.filling(cursor)

                        begin = time.perf_counter() #начало измерения времени
                        queryResalts.append(dataBase.doQuery(cursor))
                        end = time.perf_counter() #конец измерения времени
                        results.append(end - begin) #запись результата
            # plt.plot(results)
            # plt.xlabel('Attempt number')
            # plt.ylabel('Seconds')
            # plt.show() # построение графика с помощью библиотеки mathplotlib

            theWorstRes, theBestRes, averageRes = max(results),
min(results), sum(results)/len(results)
            return (theWorstRes, theBestRes, averageRes)

        def withCachingOption():
            queryResalts = [] #в массив будут помещены результаты запросов
            #соединение с базой данных
            with psycopg2.connect(dbname='postgres', user='postgres',
```

```
                                password='asmrt122',  host='localhost')
as conn:
            #получение курсора
            with conn.cursor() as cursor:
                dataBase = DataBase(cursor)
                # dataBase.create(cursor)
                # dataBase.filling(cursor)

                results = [] #результаты времени
                for i in range(100):
                    begin = time.perf_counter() #начало измерения времени
                    queryResalts.append(dataBase.doQuery(cursor))
                    end = time.perf_counter() #конец измерения времени
                    results.append(end - begin) #запись результата

                theWorstRes, theBestRes, averageRes = max(results),
min(results), sum(results)/len(results)
                return(theWorstRes, theBestRes, averageRes)


    if __name__ == "__main__":
        theWorstCachingRes,  theBestCachingRes,  averageCachingRes  =
withCachingOption()
        theWorstNotCachingRes,                    theBestNotCachingRes,
averageNotCachingRes = withoutCachingOption()
        print("""\tResults with caching:
                the worst is {theWorstCachingRes},
                the best is {theBestCachingRes},
                average is {averageCachingRes}
            Results without caching:
                the worst is {theWorstNotCachingRes},
                the best is {theBestNotCachingRes},
                average                                              is
{averageNotCachingRes}""".format(theWorstCachingRes=theWorstCachingRes,

theBestCachingRes=theBestCachingRes,

averageCachingRes=averageCachingRes,

theWorstNotCachingRes=theWorstNotCachingRes,

theBestNotCachingRes=theBestNotCachingRes,

averageNotCachingRes=averageNotCachingRes))
```

Название файла: clickhouse.py

```
import clickhouse_connect
import numpy
import random
import string
import time
import matplotlib.pyplot as plt
```

```
    #функция генерации случайной строки для заполнения столбцов типа str
    def generate_random_string(length):
        letters = string.ascii_lowercase
        rand_string = ''.join(random.choice(letters) for i in range(length))
        return rand_string

    #класс с функционалом работы с базой данных
    class DataBase:
        def __init__(self, client): #конструктор класса
            self.client = client

        def create(self, client): #создание таблицы
             #SQL запрос создания таблицы postgresTable, если она еще не существует
            #с начальным заданным уникальным столбцом id
            client.command("CREATE TABLE IF NOT EXISTS clickhouseTable (id Int32) Engine MergeTree ORDER BY tuple()")
            intColumnsNames = ['a_int', 'b_int', 'c_int', 'd_int',
                               'e_int', 'f_int', 'g_int', 'h_int',
                               'i_int', 'j_int', 'k_int', 'l_int',
                               'm_int', 'n_int', 'o_int', 'p_int',
                               'q_int', 'r_int','s_int', 't_int']
            #SQL запрос добавления столбцов
            for i in range (len(intColumnsNames)):
                client.command("ALTER TABLE `default`.clickhouseTable ADD COLUMN IF NOT EXISTS {name} Int32".format(name = intColumnsNames[i]))
            strColumnsNames = ['a_str', 'b_str', 'c_str', 'd_str',
                               'e_str', 'f_str', 'g_str', 'h_str',
                               'i_str', 'j_str', 'k_str', 'l_str',
                               'm_str', 'n_str', 'o_str', 'p_str',
                               'q_str', 'r_str','s_str', 't_str']
            for i in range (len(strColumnsNames)):
                client.command("ALTER TABLE `default`.clickhouseTable ADD COLUMN IF NOT EXISTS {name} String(10)".format(name = strColumnsNames[i]))


        def filling(self, client): #функция заполнения таблицы
            allRows = []
            for i in range (1, 100001):
                row = []
                row.append(i) #id
                for i in range (20):
                    row.append(random.randint(0, 500000),) #заполнение случайными целочисленными значениями
                for i in range (20):
                    row.append(generate_random_string(random.randint(1, 10)),) #заполнение рандомными строковыми значениями
                allRows.append(row)
```

```
            # Делаем INSERT запрос к базе данных, используя обычный SQL-синтаксис
            client.insert('clickhouseTable', allRows, column_names='*')


        def doQuery(self, client): #функция запроса (операции чтения)
            # Делаем SELECT запрос к базе данных, используя обычный SQL-синтаксис
            #получение результатов запроса
            result = client.query("""
                        SELECT id, a_str, o_str, k_str, a_int, f_int, t_int
                        FROM clickhouseTable
                        WHERE (((a_int > 5000) and (b_int % 10 = 5) and (f_int/k_int > 135)
                            and (t_int % 100 > 2) and (i_int + j_int + n_int + h_int < 40000000)) or
                            ((a_str not in ('qwert', 'asd', 'aaaaa')) and (o_str in ('ffff', 'f', 'fff', 'ff', 'fffff'))
                            and (e_str not in ('poi', 'fdgdgdf', 'fff'))) or
                            ((k_str in ('aa')) and (j_str not in ('a', 'f', 'k'))))
                            and ((id > 30) or (id < 80000) or (id % 10 = 9) or (id = 17) or (id % 2 = 8))
                        ORDER BY a_int, id
                        LIMIT 40
                        """
                        )
            return result

        def printResults(self, client): #вывод результатов запроса в консоль
            results = self.doQuery(client)
            print(results.result_rows)

    def withoutCachingOption():
        queryResalts = []  #в массив будут помещены результаты запросов
        results = [] #результаты времени
        for i in range(100):
            client = clickhouse_connect.get_client(host='localhost', port=18123, username='default', password='')
            dataBase = DataBase(client)
            # dataBase.create(client)
            # dataBase.filling(client)
            dataBase.doQuery(client)
            begin = time.perf_counter() #начало измерения времени
            queryResalts.append(dataBase.doQuery(client))
            end = time.perf_counter() #конец измерения времени
            results.append(end - begin)  #запись результата
        # plt.plot(results)
```

```
        # plt.xlabel('Attempt number')
        # plt.ylabel('Seconds')
        # plt.show() # построение графика с помощью б
иблиотеки mathplotlib
        theWorstRes,    theBestRes,    averageRes    =    max(results),
min(results), sum(results)/len(results)
        return (theWorstRes, theBestRes, averageRes)


    def withCachingOption():
        queryResalts = [] #в массив будут помещены резу
льтаты запросов
        client    =    clickhouse_connect.get_client(host='localhost',
port=18123, username='default', password='')
        dataBase = DataBase(client)
        # dataBase.create(client)
        # dataBase.filling(client)
        dataBase.doQuery(client)
        results = [] #результаты времени
        for i in range(100):
            begin = time.perf_counter()  #начало измерения в
ремени
            queryResalts.append(dataBase.doQuery(client))
            end = time.perf_counter() #конец измерения врем
ени
            results.append(end - begin) #запись результата
        theWorstRes,    theBestRes,    averageRes    =    max(results),
min(results), sum(results)/len(results)
        return(theWorstRes, theBestRes, averageRes)

    if __name__ == "__main__":
        theWorstCachingRes,   theBestCachingRes,   averageCachingRes   =
withCachingOption()
        theWorstNotCachingRes,                    theBestNotCachingRes,
averageNotCachingRes = withoutCachingOption()
        print("""\tResults with caching:
                the worst is {theWorstCachingRes},
                the best is {theBestCachingRes},
                average is {averageCachingRes}
            Results without caching:
                the worst is {theWorstNotCachingRes},
                the best is {theBestNotCachingRes},
                average                                          is
{averageNotCachingRes}""".format(theWorstCachingRes=theWorstCachingRes,

theBestCachingRes=theBestCachingRes,

averageCachingRes=averageCachingRes,

theWorstNotCachingRes=theWorstNotCachingRes,

theBestNotCachingRes=theBestNotCachingRes,

averageNotCachingRes=averageNotCachingRes))
```