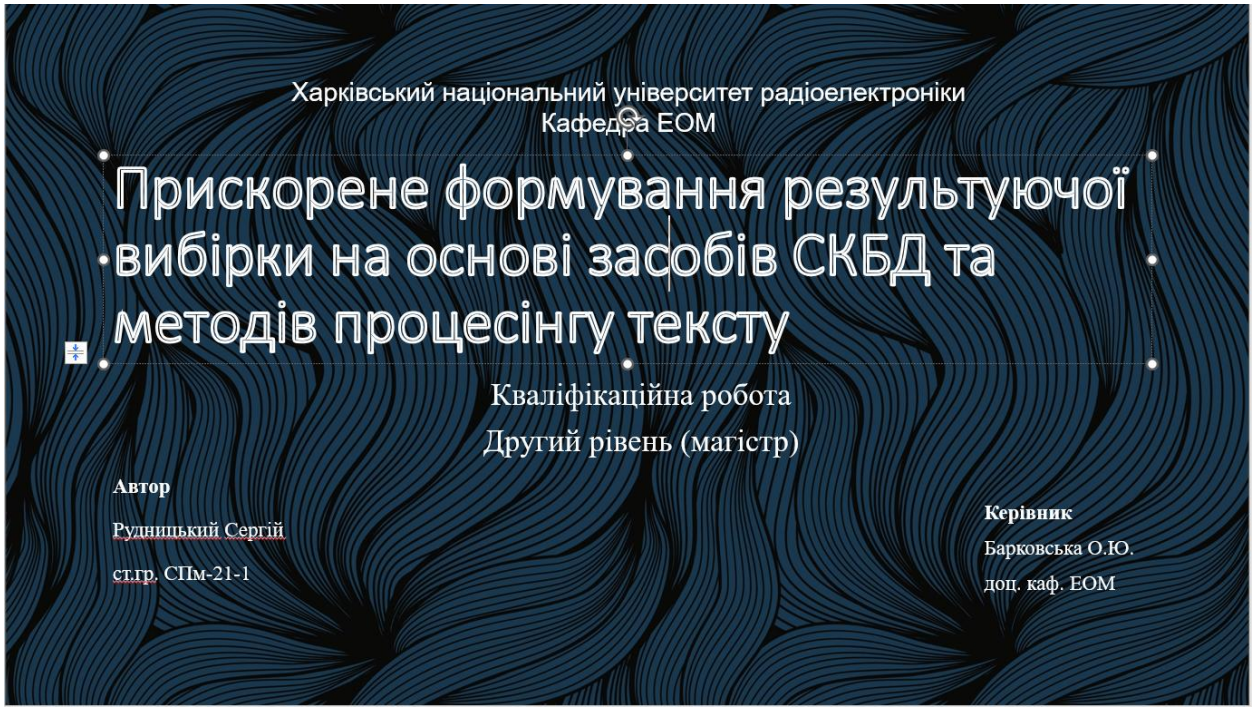
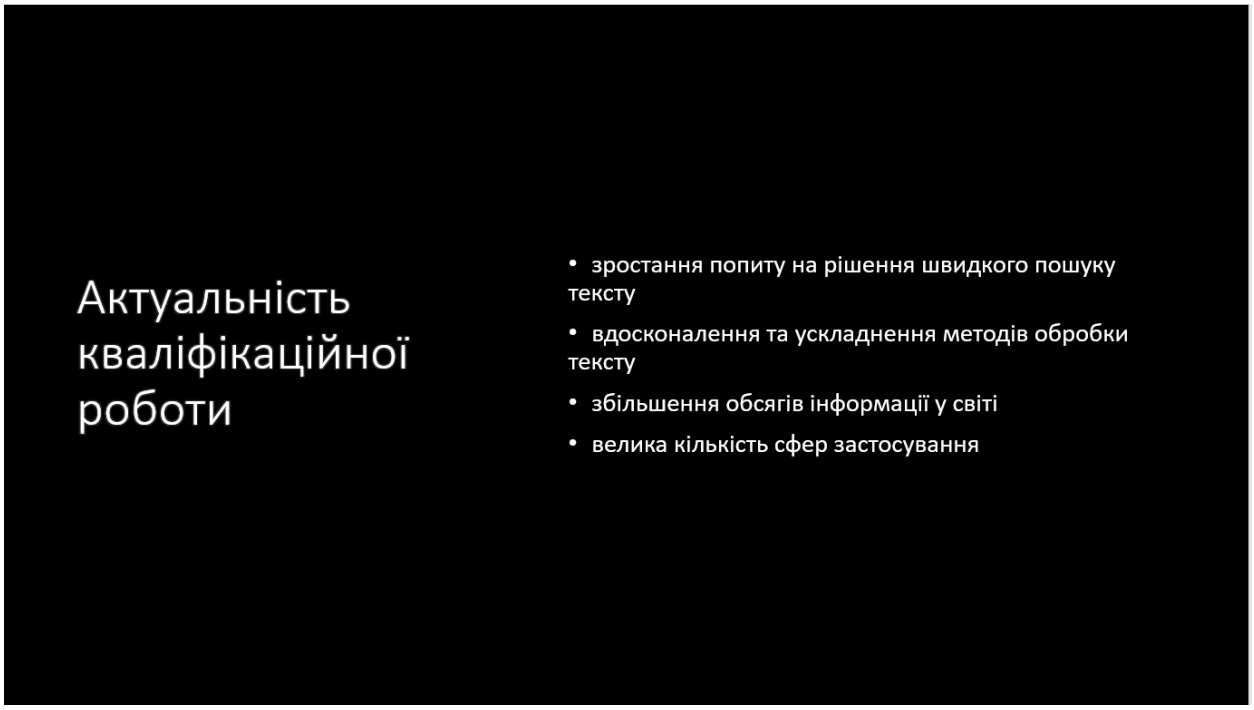
ДОДАТОК А

Графічний матеріал кваліфікаційної роботи


Харківський національний університет радіоелектроніки
Кафедра ЕОМ
Прискорене формування результуючої вибірки на основі засобів СКБД та методів процесінгу тексту
Кваліфікаційна робота
Другий рівень (магістр)
Автор
Руднищький Сергій
ст.гр. СПм-21-1
Керівник
Барковська О.Ю.
доц. каф. ЕОМ


Актуальність кваліфікаційної роботи
• зростання попиту на рішення швидкого пошуку тексту
• вдосконалення та ускладнення методів обробки тексту
• збільшення обсягів інформації у світі
• велика кількість сфер застосування

Огляд існуючих рішень

Click to add subtitle

Характеристика	Патерновий пошук	Секціонування	Full-Text Search
Швидкодія	Залежить від розміру	Залежить від розміру	Задовільна
Навантаження	Припустиме	Припустиме	Іноді велике
Вплив на пам’ять	Немає	Є	Є
Складність використання	Відносна	Допустима	Допустима

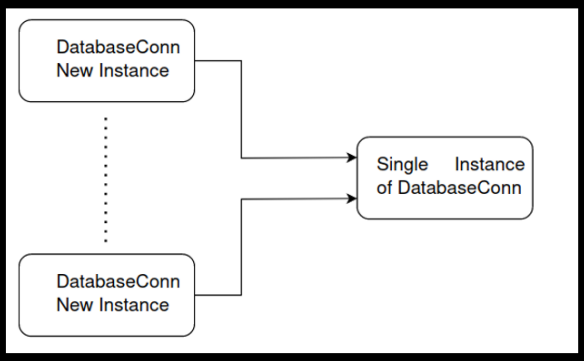
Мета кваліфікаційної роботи

- є пошук та покращення методів пришвидшення отримання результуючої вибірки та обробки тексту в великих базах даних, а також створення застосунка для аналізу отриманих результатів

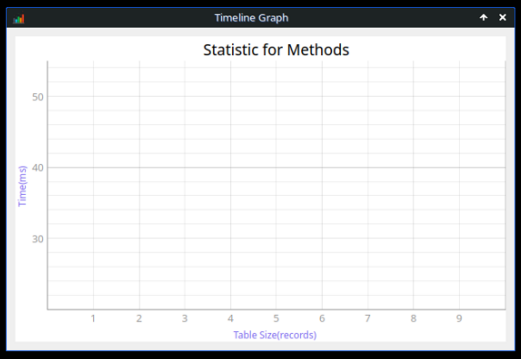
Задачі кваліфікаційної роботи

- проаналізувати чинники, що призводять до уповільнення обробки запитів у БД;
- провести порівняльний аналіз існуючих підходів обробки тексту в БД;
- розробити модель прискореного формування результуючої вибірки в БД;
- модифікувати метод повнотекстового пошуку задля пришвидшення пошуку тексту;
- підтвердити працездатність запропонованої моделі, чи комбінування методів повнотекстового пошуку із патерновим та вплив модифікації методу за допомогою існуючих можливостей реляційних СКБД;
- виконати експериментальне дослідження конкурентоспроможності розробленого рішення супроти існуючих;
- проаналізувати отримані результати.

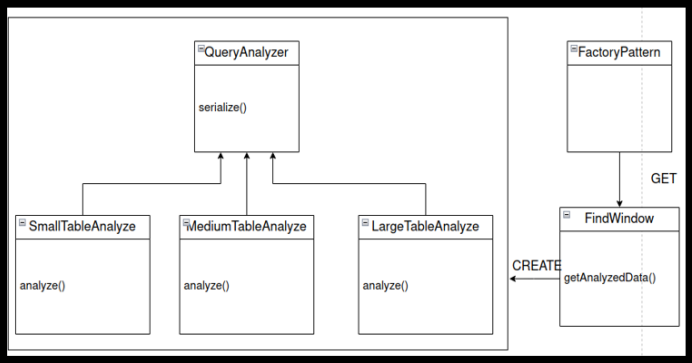
Рішення поставленої задачі Singleton для СКБД



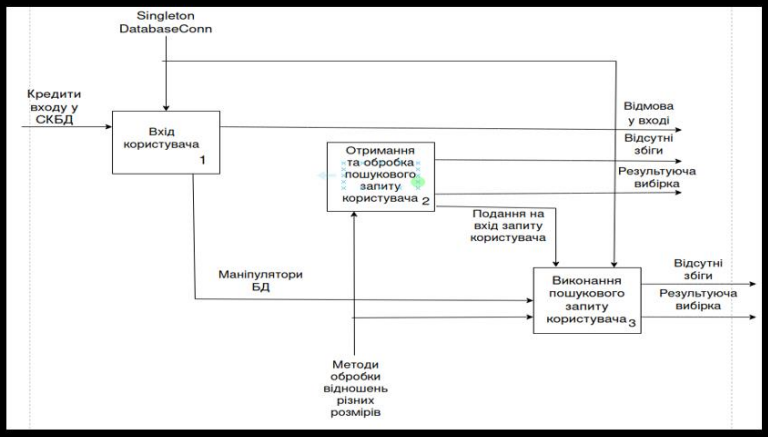
Рішення поставленої задачі. Обробка та вікно статистики



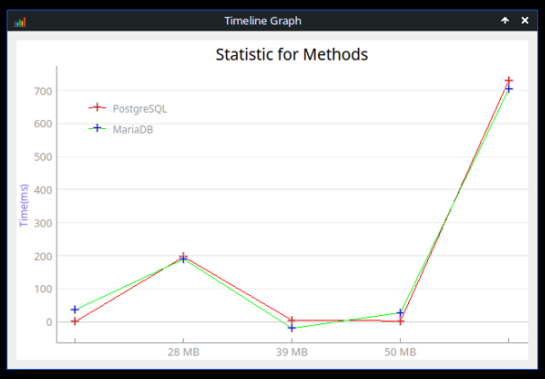
Рішення поставленої задачі. Обробка запиту користувача

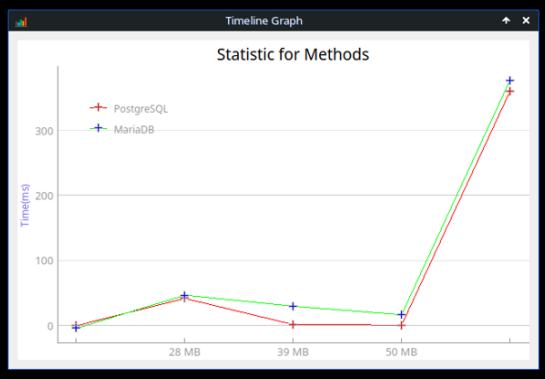


Рішення поставленої задачі. IDEF0 діаграма застосунку



Аналіз отриманих результатів. Графіки результату двох експериментів





Аналіз отриманих результатів. Таблиці порівняння результатів експериментів

Результати	small_1	small_2	medium_1	medium_2	large_1
PostgreSQL(перший)	0.80695	196.99	4.367	2.4	731.3288
PostgreSQL(другий)	0.04695	41.84495	1.798	0.7349	360.0453
Прискорення(%)	17.187433	4.707617	2.428809	3.265750	2.031213

Результати	small_1	small_2	medium_1	medium_2	large_1
MariaDB(перший)	1.04695	180.84495	5.798	2.7349	755.0453
MariaDB(другий)	0.0574	45.986	5.452	2.156	365.991
Прискорення(%)	18.239547	3.932606	1.063462	1.268506	2.063016

Висновки

- В процесі роботи над кваліфікаційною роботою було проведено дослідження методів пришвидшення видачі результуючої вибірки та обробки тексту;
- На основі проведеного дослідження було запропоновано та покращено методи пришвидшення повернення результуючої вибірки та обробки тексту. В процесі розробки було проведено адаптацію цих методів до різних запитів користувача;
- В ході дослідження було проведено оцінку результативності та часу роботи розроблених методів;
- У результаті виконаної роботи було створено програму та методи, що повністю задовольняють поставлені вимоги, а програма має необхідний функціонал.

ДОДАТОК Б

програмний код

Б.1 Реалізація модулів роботи із БД

Б.1.1 Реалізація класу DatabaseConn

```
import psycopg2 as postgres
import mariadb
import sys
import os


class DatabaseConn:
    instance = None
    __exception_message = str()
    __query_error = str()
    __which_db = str()
    __params_dict = dict()
    __query_result = None
    __execute_plan = None

    @staticmethod
    def getInstance():
        if DatabaseConn.__instance is None:
            DatabaseConn()
        return DatabaseConn.__instance

    def __init(self, db_name, db_params):
        if DatabaseConn.__instance is not None:
            raise Exception(f'This class is Singleton!')
        else:
            DatabaseConn.__instance = self
            self.__which_db = db_name
            self.__params_dict = db_params
            self.db_manipulators()

    @property
    def which_db(self):
        return self.__which_db


    @which_db.setter
    def which_db(self, value):
        self.__which_db = value
```

```
    @property
    def exception_message(self):
        return self.__exception_message

    @exception_message.setter
    def exception_message(self, value):
        self.__exception_message = value

    @property
    def query_error(self):
        return self.__query_error

    @query_error.setter
    def query_error(self, value):
        self.__query_error = value

    @property
    def query_result(self):
        return self.__query_result

    @query_result.setter
    def query_result(self, value):
        self.__query_result = value

    @property
    def execute_plan(self):
        return self.__execute_plan

    @execute_plan.setter
    def execute_plan(self, value):
        self.__execute_plan = value

    @property
    def params_dict(self):
        return self.__params_dict

    @params_dict.setter
    def params_dict(self, value):
        self.__params_dict = value

    def db_manipulators(self):
        print(self.params_dict)
        try:
            if self.which_db == 'postgres':
                db_exception = postgres.DatabaseError
                self.db_connection =
postgres.connect(**self.params_dict)
                self.db_cursor = self.db_connection.cursor()
            else:
                db_exception = mariadb.Error
                self.db_connection =
mariadb.connect(**self.params_dict)
                self. db_cursor = self.db_connection.cursor()
```

```
        except (Exception, db_exception) as error:
            message = f'Error: {error}'
            self.exception_message = message
            print(message)

    def execute_query(self, query):
        try:
            self.query_result = None
            print(query)
            self.db_cursor.execute(query)
            first_predicat = (query.split(' ')[0]).lower()
            print(first_predicat)
            if first_predicat in ['create', 'insert']:
                pass
            else:
                self.query_result =
self.fields_result_list(self.db_cursor)
            query_plan = 'EXPLAIN ANALYZE ' + query
            self.db_cursor.execute(query_plan)
            self.execute_plan = self.db_cursor.fetchall()
            self.db_connection.commit()
        except Exception as e:
            self.query_error = str(e)
            print(e)
            self.db_cursor.execute('ROLLBACK')
            self.db_connection.commit()

    def close_connection(self):
        DatabaseConn.__instance = None

    def fields_result_list(self, db_cursor):
        result_column = db_cursor.description
        query_result = db_cursor.fetchall()

        return [result_column, query_result]
```

Б.1.2 Реалізація класу StatisticHandler

```
import sys
import sqlite3
import os


class StatisticHandler:

    def init(self):
        self.set_db_manipulators()

    def set_db_manipulators(self):
        self.__db_name = './db_files/analyzingResult'
        self.__table_name = 'analyze_result'
```

```
        self.db_connector = sqlite3.connect(self.__db_name)
        self.db_cursor = self.db_connector.cursor()

    @property
    def table_name(self):
        return self.__table_name

    def write_statistics(self, data_set):
        try:
            placeholders = ', '.join(['%s'] *
len(self.data_set))
            columns = ', '.join(data_set.keys())
            placeholders = ', '.join(['?'] * len(data_set))
            sql = "INSERT INTO %s ( %s ) VALUES (%s)" %
(self.table_name, columns, placeholders)

            self.db_cursor.execute(sql, list(data_set.values()))
            self.db_connector.commit()
            self.db_connector.close()
        except Exception as error:
            print(f'Error: {error}', sql)

    def read_statistics(self, timeline):
        statistic_set = list()
        try:
            temp_dict =  {
                            'today': '0 day',
                            'yesterday': '-1 day',
                            'week': '-7 day',
                            'month': '-30 day',
                            'all_time': '-100 day'
                        }
            db_period = temp_dict[timeline]
            query = """SELECT table_size, exec_time, exec_date,
dbms FROM %s
                        WHERE exec_date >= date('now', '%s')"""
% (self.table_name, db_period,)


            self.db_cursor.execute(query)
            result_set = self.db_cursor.fetchall()
            for i in result_set:
                temp_dict = {
                                'table_size': i[0],
                                'exec_time': i[1].split(' ')[0],
                                'exec_date': i[2],
                                'dbms': i[-1]
                            }
                statistic_set.append(temp_dict)
        except Exception as error:
            print(f'Error: {error}')

        return statistic_set
```

Б.2 Елементи головного вікна

Б.2.1 Реалізація класу головного вікна

```
import sys

from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import *
from PyQt5 import QtWidgets, uic

import windows_ui_py.main_window as main_window
from db_files.database_conn import DatabaseConn
from main_code.message_boxes import InfoBox
from main_code.table_mode import TableModel
from main_code.finder_window import FinderWindow
from main_code.timeline_analyze import TimelineWindow


class FTSAnalyzer(QMainWindow, main_window.Ui_FTSAnalyzer):
    def init(self, parent=None):
        super(FTSAnalyzer, self).init(parent=parent)
        self.setupUi(self)

        self.query_button.clicked.connect(self.submit_query)
        self.init_functionality()

    def init_functionality(self):

self.menuFind.triggered[QAction].connect(self.open_finder_window
)

self.menuTimeline.triggered[QAction].connect(self.open_timeline_
window)

    def submit_query(self):
        self.database_conn = DatabaseConn.getInstance()
        query = self.query_editline.toPlainText()


        self.database_conn.execute_query(query)
        if self.database_conn.query_error:
            InfoBox(self, self.database_conn.query_error)
            self.database_conn.query_error = str()
        else:
            window_name = 'Succes'
            message = 'Executed successfully!'
            InfoBox(my_parent=self, window_name=window_name,
message=message)
            query_result = self.database_conn.query_result
```

```
            query_explain = self.database_conn.execute_plan
            self.process_query_result(query_result,
query_explain)

    def process_query_result(self, query_result, query_explain):
        self.table_model = TableModel(query_result)

        self.result_scope.setModel(self.table_model)
        self.explain_scope.setPlainText(query_explain[0][0])

    def open_finder_window(self, trigger):
        if trigger == self.actionFindText:
            self.output_finder_window()

    def open_timeline_window(self, trigger):
        if trigger == self.actionGraphs:
            self.output_timeline_window()

    def output_finder_window(self):
        self.finder_window = FinderWindow()
        self.finder_window.show()

    def output_timeline_window(self):
        self.timeline_window = TimelineWindow()
        self.timeline_window.show()
```

Б.2.2 Реалізація класу вікна запиту користувача

```
import sys

from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import *
from PyQt5 import QtWidgets, uic

from db_files.database_conn import DatabaseConn
from main_code.query_analyzer import QueryAnalyzer
import windows_ui_py.finder_window as finder_window
from main_code.processing_window import ProcessingWindow


class FinderWindow(QDialog, finder_window.Ui_text_finder):

    def init(self):
        super(FinderWindow, self).init()
        self.setupUi(self)

        self.database_conn = DatabaseConn.getInstance()
        self.init_functionality()

    def init_functionality(self):
```

```
        self.init_tables_list()
        self.setModal(True)

self.findTextButton.clicked.connect(self.submit_user_finding_que
ry)


    def init_tables_list(self):
        is_postgres = self.database_conn.which_db == 'postgres'

        if is_postgres:
            query = """SELECT * FROM pg_catalog.pg_tables WHERE
schemaname='public'"""
            self.database_conn.db_cursor.execute(query)
            query_result =
self.database_conn.db_cursor.fetchall()
            for i in query_result:
                table_name = i[1]
                self.tablesTitleList.addItem(table_name)

    def get_query_text(self):
        return self.textScope.toPlainText()

    def is_case_checked(self):
        return self.matchCase.isChecked()

    def is_entire_checked(self):
        return self.matchEntireContent.isChecked()

    def get_tables_list_value(self):
        return self.tablesTitleList.currentText()

    def submit_user_finding_query(self):
        text_temp = self.get_query_text()

        self.query_text = text_temp if self.is_case_checked()
else text_temp.lower()
        self.query_text = self.query_text if
self.is_entire_checked() else self.query_text.split(' ')

        self.query_table = self.get_tables_list_value()

        self.query_analyzer = QueryAnalyzer(self.query_table,
self.query_text)
        self.analyzed_data =
self.query_analyzer.get_analyzed_data()
        result = self.analyzed_data.analyze_result
        explain = self.analyzed_data.analyze_explain

        self.processing_window = ProcessingWindow(result,
explain)
```

Б.2.3 Реалізація класу вікна статистики

```
import sys
import random

from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import *
from PyQt5 import QtWidgets, uic
from pyqtgraph import PlotWidget, plot
import pyqtgraph as pg


import windows_ui_py.timeline_graph as timeline_graph
from db_files.database_conn import DatabaseConn
from db_files.statistic_handler import StatisticHandler


class TimelineGraph(QDialog, timeline_graph.Ui_timeline_graph):
    def init(self, timeline_period):
        super(TimelineGraph, self).init()
        self.setupUi(self)

        self.timeline_period = timeline_period
        self.database_conn = DatabaseConn.getInstance()
        self.init_functionality()


    def init_functionality(self):
        self.init_elements()
        self.init_graph_elements()
        self.plot_graph()

    def init_elements(self):
        self.graphWidget = pg.PlotWidget()
        self.graphLayout.addWidget(self.graphWidget)
        self.setModal(True)

    def init_graph_elements(self):
        self.graphWidget.setBackground('w')
        styles = {"color": "#7b68ee", "font-size": "12px"}

        self.graphWidget.setTitle('Statistic for Methods',
color="black", size="14pt")

        self.graphWidget.setLabel("left", "Time(ms)", **styles)
        self.graphWidget.setLabel("bottom", "Table
Size(records)", **styles)

        self.graphWidget.showGrid(x=True, y=True)

        self.graphWidget.setXRange(0, 10, padding=0)
```

```
        self.graphWidget.setYRange(20, 55, padding=0)



    def plot_graph(self, table_size='small'):
        self.statistic_handler = StatisticHandler()
        statistic_set =
self.statistic_handler.read_statistics(self.timeline_period)

        mean_values = dict()
        if statistic_set is not None:
            for item in statistic_set:
                dbms = item['dbms']
                rows_num = str(item['table_size'])
                execution = float(item['exec_time'])
                if dbms in mean_values:
                    if rows_num in mean_values[dbms]:
                        temp = execution
                        mean_values[dbms][rows_num].append(temp)
                    else:
                        mean_values[dbms].setdefault(rows_num,
[execution])
                else:
                    mean_values.setdefault(dbms, {rows_num:
[execution]})
        else:
            print('NONE')
        relation_size = list()
        mean_time_postgres = list()
        for key, value in mean_values.items():
            for k_v, v_v in value.items():
                relation_size.append(k_v)
                temp = float(sum(v_v)/len(v_v))
                mean_time_postgres.append(temp)
        relation_dict = dict(enumerate(relation_size))

        mean_time_maria = list()
        for key, value in mean_values.items():
            for k_v, v_v in value.items():
                relation_size.append(k_v)
                temp = float(sum(v_v)/len(v_v))
                mean_time_maria.append(temp)

        print(mean_time_maria)

        x = [1, 2, 3,4 ,5]
        xdict = {1: '7000 kB', 2: '28 MB', 3: '39 MB', 4: '50
MB', 5: '790 MB'}
        pen_1 = pg.mkPen(color=(255, 0, 0))
        pen_2 = pg.mkPen(color=(-0, 255, 0))
        name_1 = "PostgreSQL"
        name_2 = 'MariaDB'
        stringaxis = pg.AxisItem(orientation='bottom')
```