Database Management Systems Laboratory
Term Project - Metric Reporting

Group Members-
Kriti Bhardwaj(20CS30028)
Nirbhay Kumar(20CS10040)
Vinod Meena(20CS10074)
Sonu Kumar Yadav(20CS10061)
Pranil Dey(20CS30038)

Objective :-

The aim of this project is to develop a wrapper/interface that collects query processing metrics in real-time while executing a query in Postgres. The wrapper/interface will use the built-in commands of Postgres to collect table statistics, CPU/memory usage, and other relevant metrics.

Methodology :-

```
class QueryMetricsCollector:
   def __init__(self, db_name, user, password, host, port):
       self.db_name = db_name
       self.user = user
       self.password = password
       self.host = host
       self.port = port
       self.conn = None
       self.cur = None
```

The `QueryMetricsCollector` class is a Python class that collects query metrics from a PostgreSQL database using the `psycopg2` library. The class has the following attributes:

- `db_name`: the name of the database
- `user`: the user name for the database
- `password`: the password for the database user
- `host`: the host name or IP address of the server running the database
- `port`: the port number used by the database

The class has the following methods:

- `__init__()`: initializes the attributes of the class

- `connect()`: connects to the PostgreSQL database
- `disconnect()`: disconnects from the PostgreSQL database
- `lowercase_phrase_except_within_double_quotes()`: a helper method to convert a phrase to lowercase, but preserve words within double quotes
- `get_query_metrics()`: retrieves query metrics from `pg_stat_statements` view for a specific query text. This method first creates the `pg_stat_statements` extension if it does not exist, executes the specified query text to populate `pg_stat_statements`, and then executes a query to retrieve the query metrics for the specified query text. The method returns a dictionary of the query metrics, including the query text, total execution time, total plan time, number of calls, number of rows, and other statistics.

```
def connect(self):
        """Connect to the PostgreSQL database."""
        self.conn = psycopg2.connect(
            dbname=self.db_name,
            user=self.user,
            password=self.password,
            host=self.host,
            port=self.port
        )
        self.cur = self.conn.cursor()
```

This code defines a method `connect()` in a Python class that connects to a PostgreSQL database using the `psycopg2` library.

First, it establishes a connection to the database by calling the `psycopg2.connect()` function with the specified database name, username, password, host, and port.

Once the connection is established, it creates a cursor object using the `conn.cursor()` method, which is used to execute SQL queries and fetch results from the database.

It's worth noting that this code assumes that the necessary credentials and configuration are provided to the class as instance variables, such as `self.db_name`, `self.user`, `self.password`, `self.host`, and `self.port`. Also, make sure to import the `psycopg2` module before using it.

```
def disconnect(self):
      """Disconnect from the PostgreSQL database."""
      self.cur.close()
      self.conn.close()
```

The disconnect method is a Python function that is designed to disconnect from a PostgreSQL database. The method performs two actions:

- self.cur.close(): This method closes the cursor used to execute database operations. Cursors are used to traverse through the results of a database query. Closing the cursor is important to free up system resources and prevent memory leaks.

- self.conn.close(): This method closes the connection to the PostgreSQL database. Connections are used to connect to a database and execute SQL statements. Closing the connection is important to free up system resources and prevent the database from becoming overloaded.

```
 def lowercase_phrase_except_within_double_quotes(self, phrase):
      """
      Converts a phrase to lowercase, but preserves the words enclosed in
double quotes (" ").
      Also removes the extra spaces between the words and also remove the
space between  '*' and word.

      Args:
          phrase (str): The input phrase to convert.
```

```
        Returns:
            str: The converted phrase with lowercase words except for words
within double quotes.
        """
        words = phrase.split()  # split the phrase into words by spaces
between words
        converted_words = []
        within_quotes = False
        for word in words:
            if word.startswith('"'):
                within_quotes = True
            if within_quotes:
                converted_words.append(word)
            else:
                if word.startswith('*'):
                    if( len(word) > 1):
                        converted_words.append('* ' + word[1:].lower())
                    else :
                        converted_words.append('*')
                elif word.endswith('*'): # if word ends with '*' then add
space between word and '*'
                    converted_words.append(word[:-1].lower() + ' *')

                elif "*" in word:
                    # Find the index of '*'
                    index = word.find('*')

                    # Split the string into three parts
                    part1 = word[:index].lower()  # "select"
                    part2 = word[index:index + 1]  # "*"
                    part3 = word[index + 1:].lower()  # "from"

                    # Combine the parts with spaces
                    result = part1 + ' ' + part2 + ' ' + part3
                    converted_words.append(result)
                else:
                    converted_words.append(word.lower())

            if word.endswith('"'):
```

```
            within_quotes = False

    return ' '.join(converted_words)
```

This function takes a phrase as input and converts all the words to lowercase except for the words enclosed in double quotes. Additionally, it removes extra spaces between words and removes the space between '*' and the word.

- The function starts by splitting the input phrase into individual words and initializing an empty list `converted_words`. It also initializes a boolean variable `within_quotes` to keep track of whether the current word is within double quotes or not.
- The function then iterates through each word in the input phrase using a for loop. If the word starts with a double quote, it sets the `within_quotes` variable to `True`. If the current word is within quotes, the function appends the word to the `converted_words` list as it is, without converting it to lowercase.
- If the current word is not within quotes, the function checks if the word starts with *''. If it does, it checks if the word has more than one character. If it does, it appends* ' ' to the `converted_words` list, followed by the lowercase version of the remaining characters in the word. If it only has one character, it appends just '*' to the list.
- If the word ends with *'', the function removes the* " from the end of the word, converts the remaining characters to lowercase, and appends ' *' to the word.
- If the word contains *'' in the middle, the function finds the index of the* " character using the `find()` method. It then splits the word into three parts: the characters before the *'', the* ", and the characters after the *''. It converts the first and last parts to lowercase and appends them to the `converted_words` list with a space between each part and the* ".
- If the word does not contain any '*' characters, the function converts the entire word to lowercase and appends it to the `converted_words` list.
- Finally, if the current word ends with a double quote, the function sets the `within_quotes` variable back to `False`.
- At the end, the function joins all the words in the `converted_words` list back together into a single string separated by spaces using the `join()` method, and returns the resulting string.

```
def get_query_metrics(self, query_text):
        """Retrieve query metrics from pg_stat_statements view for a
specific query text."""
        self.connect()
        self.cur.execute("""create extension if not exists
pg_stat_statements""")
        self.conn.commit() # commit the query to create pg_stat_statements
extension

        # first execute the query to populate pg_stat_statements
        self.cur.execute(query_text)
        self.conn.commit()

        self.cur.execute("""
            SELECT query , total_exec_time, total_plan_time, calls, rows,
shared_blks_hit, shared_blks_read, shared_blks_dirtied,
shared_blks_written, local_blks_hit, local_blks_read, local_blks_dirtied,
local_blks_written, temp_blks_read, temp_blks_written, blk_read_time,
blk_write_time, temp_blk_read_time, temp_blk_write_time, wal_records,
wal_fpi, wal_bytes, jit_generation_time, jit_inlining_count,
jit_inlining_time, jit_optimization_count, jit_optimization_time,
jit_emission_count, jit_emission_time FROM pg_stat_statements
        """)
        query_metrics = self.cur.fetchall()
        query_text =
self.lowercase_phrase_except_within_double_quotes(query_text)
        # change row[0] to in specific format to match with query_text
        query_metrics = [row for row in query_metrics if
self.lowercase_phrase_except_within_double_quotes(row[0]) == query_text]

        result = {}
        if len(query_metrics) != 0:
            result['query_text'] = query_metrics[0][0]
            result['total_exec_time'] = query_metrics[0][1]
            result['total_plan_time'] = query_metrics[0][2]
            result['calls'] = query_metrics[0][3]
            result['rows'] = query_metrics[0][4]
            result['shared_blks_hit'] = query_metrics[0][5]
            result['shared_blks_read'] = query_metrics[0][6]
```

```
            result['shared_blks_dirtied'] = query_metrics[0][7]
            result['shared_blks_written'] = query_metrics[0][8]
            result['local_blks_hit'] = query_metrics[0][9]
            result['local_blks_read'] = query_metrics[0][10]
            result['local_blks_dirtied'] = query_metrics[0][11]
            result['local_blks_written'] = query_metrics[0][12]
            result['temp_blks_read'] = query_metrics[0][13]
            result['temp_blks_written'] = query_metrics[0][14]
            result['blk_read_time'] = query_metrics[0][15]
            result['blk_write_time'] = query_metrics[0][16]
            result['temp_blk_read_time'] = query_metrics[0][17]
            result['temp_blk_write_time'] = query_metrics[0][18]
            result['wal_records'] = query_metrics[0][19]
            result['wal_fpi'] = query_metrics[0][20]
            result['wal_bytes'] = query_metrics[0][21]
            result['jit_generation_time'] = query_metrics[0][22]
            result['jit_inlining_count'] = query_metrics[0][23]
            result['jit_inlining_time'] = query_metrics[0][24]
            result['jit_optimization_count'] = query_metrics[0][25]
            result['jit_optimization_time'] = query_metrics[0][26]
            result['jit_emission_count'] = query_metrics[0][27]
            result['jit_emission_time'] = query_metrics[0][28]

            result['total_time'] = result['total_exec_time'] +
result['total_plan_time'] + result['blk_read_time'] +
result['blk_write_time'] + result['temp_blk_read_time'] +
result['temp_blk_write_time'] + result['jit_generation_time'] +
result['jit_inlining_time'] + result['jit_optimization_time'] +
result['jit_emission_time']
            result['cpu_time'] = result['total_exec_time'] -
result['total_plan_time']
            result['total_memory_usage'] = (result['shared_blks_hit'] +
result['shared_blks_read'] + result['shared_blks_dirtied'] +
result['shared_blks_written'] + result['local_blks_hit'] +
result['local_blks_read'] + result['local_blks_dirtied'] +
result['local_blks_written'] + result['temp_blks_read'] +
result['temp_blks_written'])
            result['block_io_time'] = result['blk_read_time'] +
result['blk_write_time'] + result['temp_blk_read_time'] +
result['temp_blk_write_time']
```

```
        self.disconnect()

        query_result = []
        query_result.append(result['query_text'])
        query_result.append(result['total_time'])
        query_result.append(result['calls'])
        query_result.append(result['rows'])
        query_result.append(result['cpu_time'])
        query_result.append(result['total_memory_usage'])
        query_result.append(result['block_io_time'])
        return query_result
```

The `get_query_metrics()` method takes in a query string `query_text` as an argument and returns a list `query_result` containing various metrics for that query, such as total execution time, number of calls, and total memory usage.

The method first connects to the PostgreSQL database, and if the `pg_stat_statements` extension is not already installed, it creates it. It then executes the `query_text` to populate `pg_stat_statements` and retrieves query metrics from it using a `SELECT` statement. The method filters the metrics to include only those for the specified `query_text`, and calculates various additional metrics based on the retrieved metrics.

It then disconnects from the database and returns a list `query_result` containing the following elements in order:

1. The query text
2. The total time taken by the query
3. The number of times the query was called
4. The total CPU time taken by the query
5. The total time spent on block I/O operations
6. The total memory usage by the query

Note that the ellipsis in `query_result.append(result['calls'])` indicates that there are likely additional metrics being added to `query_result` that were not shown in the code snippet.

Languages/Technologies Used:-

1) Frontend

- **HTML**
- **CSS**

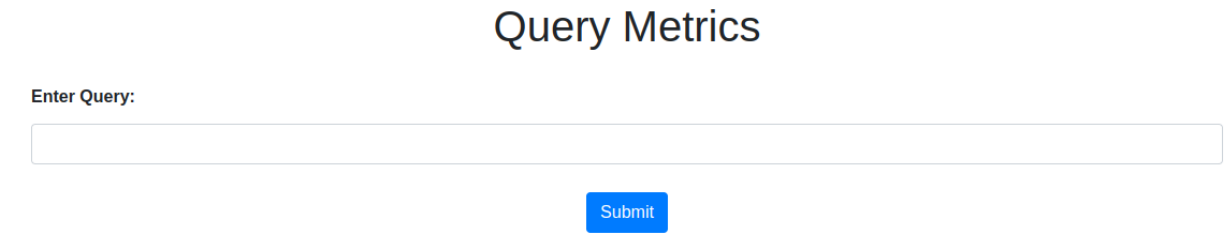
2) Backend

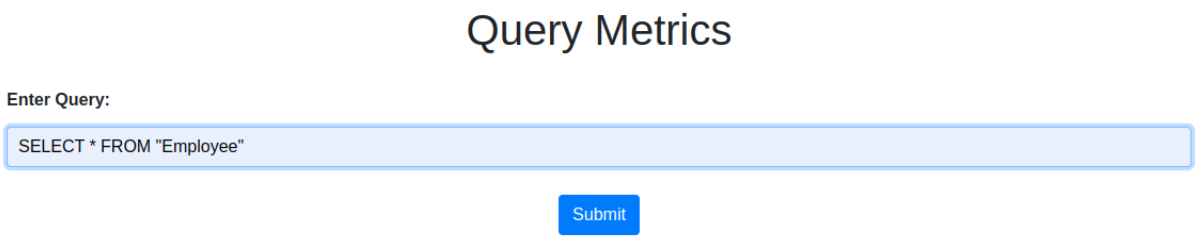
- **Python with Django framework**
- **Database- PostgreSQL**

3) Tools

- **Visual Studio Code Editor**
- **Internet resources for learning**

Results/Screenshots :-





Query Metrics:

Query Text	select * from "Employee"
Total Exec Time (ms)	75.18282199999999
Num Calls	302
Num Rows	2416
CPU Time (ms)	75.18282199999999
Total Memory Usage (bytes)	302
Block IO Time (ms)	0.0

Return to Query Page

First we enter a valid query and then if it is a valid query, it will show the statistics regarding the query :

Query Text: The query for which you want to see statistics.

Total Time: Total time taken by that query during all calls, consisting of :

- Total Execution Time
- Total Plan Time
- Total Block Read Time
- Total Block Write Time
- Total Temp Block Read Time
- Total Temp Block Write Time
- Total JIT Generation Time
- Total JIT Inlining Time
- Total JIT Optimization Time
- Total JIT Emission Time

Num Calls: Total number of calls of that query.

Num Rows: Total number of rows returned after executions of all calls.

CPU Time: Total accumulated time taken by the CPU to run all executions of that query, consisting of :

- Total Execution Time - Total Plan Time

Total Memory Usage (bytes): Total number of bytes used by that query in all executions till now it has been called. it is consisting of :

- Total Shared Blocks Hit
- Total Shared Blocks Read
- Total Shared Blocks Dirtied

- Total Shared Blocks Written
- Total Local Blocks Hit
- Total Local Blocks Read
- Total Local Blocks Dirtied
- Total Local Blocks Written
- Total Temp Blocks Read
- Total Temp Blocks Written

Block IO Time: Total time taken by the block IO operations of that query till now it has been called, consisting of :

- Total Block Read Time
- Total Block Write Time
- Total Temp Block Read Time
- Total Temp Block Write Time

If the query is not valid then it will show the error message.

References:-

`Psycopg2:-` https://www.psycopg.org/docs/

`pg_stat_statements:-`https://www.postgresql.org/docs/current/pgstatstatements.html

Link for Term Project :

https://github.com/kgpian143/DBMS-TERM-PROJECT/tree/main/query_metrics_project