Metric Reporting

Group Members
Mradul Agrawal (20CS30034)
Grace Sharma (20CS30022)
Umang Singla (20CS10068)
Vineet Amol Pippal (20CS30058)
Subhajyoti Halder (20CS10064)

Problem statement
Build a wrapper/interface which collects query processing metrics like table statistics, CPU/memory usage in run time while executing a query. We may use the inbuilt commands of Postgres.

Proposals
We propose to create a user friendly application which will report all the necessary query processing metrics such as:

1. Execution time: The total time taken to execute the query, including time spent waiting on locks or I/O.
2. CPU time: The amount of CPU time used to execute the query.
3. Memory usage: The amount of memory used by the query, including shared buffers and work memory.
4. Rows returned: The number of rows returned by the query.
5. Block I/O time: The amount of time spent performing block I/O operations (reading from or writing to disk) during the execution of the query.
6. Disk usage: The amount of disk space used by the query, including temporary files created during execution.
7. Lock time: The amount of time the query spent waiting on locks.
8. Parse and plan time: The amount of time spent parsing and planning the query, including optimization and generating the execution plan.
9. Number of executions: The number of times the query has been executed.
10. Calls and rows per call: The average number of calls and rows returned per call for the query.
11. Statement text: The text of the SQL statement being executed.

We also plan to create a UI application in Python which will easily report all these statistics.

Execution Ideas
To collect query processing metrics while executing a query in Postgres, we can use the built-in `pg_stat_statements` module. This module provides a view that displays statistics about the execution of SQL statements.

We did some research on how these queries can be executed and following is the workflow that would be followed:

1. Enable pg_stat_statements module:

```
CREATE EXTENSION pg_stat_statements;
```

2. Set pg_stat_statements.track parameter to all in postgresql.conf file or per-session basis.

```
pg_stat_statements.track = all
```

3. Execute the query we want to monitor.

4. Retrieve the query processing metrics using pg_stat_statements view:

```
SELECT * FROM pg_stat_statements;
```

This view provides information such as the query text, total execution time, CPU usage, memory usage, disk I/O, and other statistics for each executed query.

5. We can also use pg_stat_activity view to get current query processing status of running queries.

```
SELECT pid, state, query FROM pg_stat_activity WHERE state = 'active';
```

This will display the process ID, state, and query text for all currently running queries.

After collecting these queries, we plan to show them to the user in a friendly UI and report all the necessary metrics. We plan to follow the following plan to execute the above ideas:

1. Create a connection to the Postgres database using psycopg2.
2. Execute the query to collect the required statistics using pg_stat_statements and retrieve the results as a Pandas DataFrame.
3. Use Python frameworks to create an application with a view that displays the collected statistics. We can use templates to create an HTML page that shows the statistics in a user-friendly way.
4. In the view function, pass the Pandas DataFrame containing the collected statistics to the template engine to render the HTML page.
5. Add any necessary styling and interactivity using CSS and JavaScript to create a visually appealing and user-friendly interface.

Benefits of this application

By collecting and displaying these metrics, we can gain insights into how our queries are performing and identify any performance bottlenecks or areas for optimization.
Also by following the above steps, we can create a UI application in Python that reports query processing statistics collected from Postgres in a user-friendly way.