QueryPilotX: Query Performance Analysis and Optimization through Real-time Metrics Collection in Postgres

Aditya Choudhary, Astitva, Rishi Raj, Vikas Vijaykumar Bastewad, Yashraj Singh

Here's the link to the Github Repo

1 Introduction

In today's data-driven world, efficient and effective query processing is essential for the smooth functioning of modern database systems. Query processing metrics such as table statistics, CPU usage, and memory usage provide valuable insights into query performance and can help identify bottlenecks and areas for optimization.

In this project, we aim to develop a wrapper or interface that can be used to collect and analyze query processing metrics in real time for Postgres, one of the most widely used relational database management systems. Leveraging Python, Golang, and the psycopg2 module, we will execute queries and collect metrics during query execution.

Our wrapper will be modular, extensible, and user-friendly, and will provide advanced analytics and visualization features to enable users to optimize query performance at a granular level. The tool will be valuable for database administrators and developers looking to improve the efficiency of their database systems and drive better business outcomes.

1.1 Related work

Efficient query processing is a widely studied topic in the database research community. Many existing works have proposed techniques for improving query performance, including query optimization, query scheduling, and query processing metrics collection and analysis.

One related work is the pg_stat_statements module in Postgres, which collects statistics on the execution of SQL statements. This module provides insights into query performance, including the total execution time, number of calls, and rows affected. However, it does not provide real-time metrics collection during query execution.

Another related work is sys.dm_exec_query_stats in Microsoft SQL Server, which collects query performance metrics such as execution count, total execution time, and query plan analysis. However, this system is limited to Microsoft SQL Server and does not provide real-time metrics collection during query execution.

There are also several commercial tools available for query performance analysis and optimization, including SolarWinds Database Performance Analyzer, Dynatrace, and AppDynamics. These tools provide advanced analytics and visualization features for query performance optimization, but they come with a significant cost and are often proprietary.

In contrast, our project proposes a free and open-source tool for real-time query performance analysis in Postgres. Our wrapper provides real-time metrics collection during query execution, and our analytics and visualization features are designed to be user-friendly and accessible for database administrators and developers.

1.2 Objective

The objective of this project is to develop a wrapper/interface that can collect and report query processing metrics in real time while executing queries on a database system. The wrapper/interface will be designed to interact with the database system and gather relevant performance metrics, such as table statistics, CPU usage, and memory usage. The collected metrics will be reported in a user-friendly format, which can be easily interpreted and analyzed by stakeholders, such as database administrators, query optimizers, and performance analysts.

2 Methodology

2.1 Inbuilt postgres statistics

PostgreSQL offers a built-in statistics collector that automatically aggregates key metrics for tracking the health and performance of databases. By querying predefined statistics views, users can gain visibility into various aspects of their databases, including read and write query throughput, performance, replication and reliability, and resource utilization. Some of the key statistics views available in PostgreSQL are:

1. **pg_stat_database**: This view collects statistics about each database in the cluster, including the number of connections, commits, rollbacks, and rows/tuples fetched and returned.

2. **pg_stat_user_tables**: This view offers statistics on user-defined tables, such as the number of rows inserted, updated, or deleted, as well as information on table size and cache usage.

3. **pg_stat_user_indexes**: This view provides statistics on user-defined indexes, including metrics related to index scans, index size, and cache usage.

4. **pg_stat_bgwriter**: This view offers statistics on the background writer process, which is responsible for managing the write-back process to disk. It provides information on buffer allocations, writes, and evictions.

5. **pg_statio_user_tables**: This view displays a cumulative count of blocks read, the number of blocks that were hit in the shared buffer cache, as well as other information about the types of blocks that were read from each table.

By leveraging these predefined statistics views, users can gain insights into the performance and resource utilization of their PostgreSQL databases, allowing them to optimize their database configuration and performance tuning for better overall database health.

2.2 QueryPilotX's Statistics

Following are the statistics provided by us.

2.2.1 Database Specific Metrics

1. **get_stats_disk_usage_for_database:** This metric provides insights into the size of each database in bytes, helping us understand the storage footprint of the databases in a clear and concise manner.

2. **get_stats_tx_rate_for_database:** This metric offers a comprehensive view of the rate of transactions executed per second, as well as the rate of rollbacks executed separately since the last function call. This information helps us monitor the efficiency and effectiveness of the database operations with ease.

3. **get_stats_seconds_since_last_vacuum_per_table:** This metric presents the amount of time passed in seconds since the last vacuum was performed on each table in the database, providing us with valuable information on when tables were last cleaned up and optimized for performance.

4. **get_stats_oldest_transaction_timestamp:** This metric highlights the duration for which the longest-running transaction has been opened in the database. While this value should ideally be close to zero, an increase may indicate potential issues, such as unclosed maintenance connections, that require prompt attention to ensure optimal database performance.

5. **get_stats_index_hit_rates:** This metric offers insights into the usage of indexes versus sequential scan through the table, helping us understand the effectiveness of our indexing strategies. With a clear understanding of the data and index usage, we can make informed decisions on when high or low index usage is desirable for optimal database performance.

6. **get_stats_table_bloat:** This metric reveals the amount of wasted space in the database table due to the MVCC process. By identifying obsolete data that is marked as free but not truly deleted, this metric helps us identify and address table bloat issues, and optimize the database storage. The current implementation of this metric may be resource-intensive and can be disabled if necessary to avoid performance issues.

2.2.2 Database Cluster (Global) Metrics

1. **get_stats_client_connections:** This insightful metric provides visibility into the current number of connections open to the database, including the actual metrics for gathering connections. It's worth noting that having an excessive number of open connections, typically over a hundred, can be detrimental.

2. **get_stats_lock_statistics:** This crucial metric sheds light on the locks that queries are waiting on, as well as the locks that have been granted. Extended wait times for locks can indicate potential issues like heavy lock contention, and warrant further investigation.

3. **get_stats_heap_hit_statistics:** These valuable metric offers insights into the reads hitting the memory buffers and disk (or disk caches) in our cluster. Additionally, it calculates the heap hit ratio based on these values. It's important to note that the read amounts refer to blocks read, not actual read queries. Comparing these values with the transaction rate can provide a solid understanding of the reading activity on the database.

4. **get_stats_replication_delays:** This critical metric provides information on the replication delay in bytes between the master and each slave. In the synchronous replication state, the replication delay is zero. Monitoring this metric can help identify any potential replication lag issues that may impact data consistency and integrity.

5. **get_stats_wal_file_amount:** This metric offers visibility into the number of files in the database cluster's Write-Ahead Log (WAL) log directory (pg_wal or pg_xlog). A sudden increase in the WAL file amount may indicate issues with the WAL archiving process, which could result in disk filling up and ultimately crashing the database cluster.

6. **get_xid_remaining_ratio, get_multixact_remaining_ratio, get_multixact_members_remaining_ratio:** These insightful metrics provide the remaining percentage of transaction IDs ("xid"), multixact IDs ("mxid"), and multixact members available for use by PostgreSQL before exhaustion. Monitoring these metrics can help ensure that the vacuuming process is functioning as intended, maintaining optimal performance for the PostgreSQL instance.

7. **get_multixact_members_per_mxid:** This metric provides information on the number of multixact members per multixact ID. A higher number may indicate a faster depletion of multixact members (as seen in the **get_multixact_members_usage_ratio**). This metric can be useful in identifying potential multixact ID exhaustion issues and optimizing multixact usage in the PostgreSQL instance.

2.2.3 Real time query analysis

querypilotX presents an intricate and comprehensive overview of real-time query monitoring metrics for PostgreSQL, empowering database administrators and developers with valuable insights into query performance and resource utilization.

1. One of the key metrics monitored by querypilotX is the query duration, which reveals the time duration for which a query has been running. This enables users to swiftly identify queries that may be running longer than expected, potentially impacting performance.

2. Furthermore, querypilotX provides in-depth metrics for CPU usage, memory usage, disk I/O, and network activity for each active query. These metrics offer visibility into resource utilization, enabling users to pinpoint performance bottlenecks and identify queries that may be consuming excessive resources.

3. CPU usage is represented as a percentage of CPU time utilized by each active query, helping users identify queries that may be disproportionately using CPU resources, thereby affecting performance.

4. Memory usage is depicted as the amount of memory being utilized by each active query, along with the percentage of total memory being used by PostgreSQL processes overall. This aids in identifying queries that may be consuming excessive memory resources, potentially causing performance issues or memory leaks.

5. querypilotX also provides insights into disk I/O and network activity, showcasing the number of bytes being read and written by each active query, along with the rate of data transfer. This assists in identifying queries that may be causing excessive disk I/O or network traffic, potentially impacting performance.

6. In summary, querypilotX's real-time query monitoring metrics offer a robust and invaluable tool for monitoring PostgreSQL

databases, enabling users to proactively identify and resolve performance issues in real-time.

2.3 Analysis

After collecting the metrics related to Read query throughput and performance, write query throughput and performance, replication and reliability, and resource utilization from PostgreSQL's statistics collector and other native sources, the next step is to analyze these metrics and utilize them for effective alerting purposes. This will enable us to proactively detect and address any potential issues, ensuring optimal performance, availability, and reliability of the database environment.

2.3.1 Read query throughput and performance

Analyzing the metrics related to read query throughput and performance can provide insights into the efficiency of the database in handling read queries.

- **Sequential scans vs. index scans:** If the database is regularly performing more sequential scans over time, its performance could be improved by creating an index on data that is frequently accessed.Optimizing the usage of sequential scans and index scans can lead to improved query performance, reduced query execution times, and overall better database performance.

- **rows fetched and rows returned:** PostgreSQL tracks tup_returned as the number of rows read or scanned during query execution, regardless of whether those rows were actually returned to the client. On the other hand, tup_fetched, or "rows fetched", is the metric that counts the number of rows that contained data which was actually needed to execute the query.By monitoring and analyzing the ratio of rows fetched to rows returned, we can gain insights into how efficiently the database is processing read queries. If there is a significant discrepancy between these metrics, further analysis may be required to identify and optimize queries or table structures that are causing unnecessary row scans. This can help improve the performance of read queries and ensure efficient database operations.

2.3.2 Write query throughput and performance

- **Rows inserted, updated, and deleted:** Monitoring the number of rows inserted, updated, and deleted can help in giving an idea of what types of write queries the database is serving. For instance, if we observe a high rate of updated and deleted rows, it raises a red flag and prompts us to closely monitor the number of dead rows in the database. If the number of dead rows is consistently rising, it can negatively impact query performance and slow down the database. By regularly monitoring and analyzing the number of rows inserted, updated, and deleted, as well as the presence of dead rows, we can proactively identify and address any issues related to write queries.

- **Concurrent operations:** PostgreSQL's statistics collector provides valuable insights into concurrent operations, allowing us to assess the performance and efficiency of the database in handling multiple operations simultaneously. Metrics such as the number of concurrent connections, lock contention, and wait events provide crucial information on the database's ability to handle concurrent queries without experiencing performance degradation or bottlenecks. By monitoring these concurrent operation metrics, we can identify any potential issues related to contention for resources, such as locks, which can impact the performance and responsiveness of the database

2.3.3 Resource utilization

Similar to any other sophisticated database system, PostgreSQL utilizes various system resources such as CPU, disk, memory, and network bandwidth to perform its operations efficiently. By diligently monitoring these critical system-level metrics, we can ensure that PostgreSQL has the necessary resources at its disposal to swiftly respond to queries and effectively update data across its tables and indexes. Additionally, PostgreSQL also maintains its own comprehensive set of metrics to monitor its internal resource utilization, encompassing aspects such as connections, shared buffer usage, and disk utilization. In the following sections, we have explained these metrics in greater detail which enables us to better understand and optimize the performance of PostgreSQL.

- **Connections:** When a client requests a connection to the PostgreSQL primary server process, a new process is forked to handle that connection. PostgreSQL establishes a connection limit that determines the maximum number of concurrent connections that can be opened to the backend at any given time. It's important to note that the actual maximum number of connections may be further constrained by the limits set by the operating system.

  In scenarios with high concurrency, where there is a large number of concurrent connections, implementing a connection pool such as PgBouncer can be beneficial. The connection pool acts as an intermediary between the applications and the PostgreSQL backends, effectively distributing the direct connections to the primary server.

  If we notice that the number of active connections consistently approaches the maximum connection limit, it could be an indication that applications are creating new connections for each request instead of reusing existing connections. This may be due to long-running queries or inefficient connection management. Implementing a connection pool can help mitigate this issue by ensuring that idle connections are consistently reused instead of burdening the primary server with frequent opening and closing of connections.

  To further optimize connection management, we set the value for idle_in_transaction_session_timeout. This parameter instructs PostgreSQL to automatically close connections that remain idle for a specified period of time. By default, this value is set to 0, which means that this feature is disabled. Adjusting this parameter can help improve resource utilization and connection management in PostgreSQL.

- **Shared Buffer Usage:** When PostgreSQL accesses data, it first checks if the data is already present in the shared buffer cache or the operating system (OS) cache to avoid disk I/O. If the data is not cached, it needs to be retrieved from disk, and it is then cached both in the OS cache and the database's shared buffer cache to optimize future queries. This means that some data may be cached in multiple places simultaneously. To optimize this caching process, PostgreSQL recommends limiting the shared_buffers parameter, which controls the size of the shared buffer cache, to around 25 percent of the available OS memory.

  It's important to note that pg_statio, which provides statistics related to the shared buffer cache, does not account for hits in the OS cache. Therefore, if the cache hit rate appears low based solely on pg_statio statistics, it's crucial to remember that it does not capture hits in the OS cache. To gain a comprehensive understanding of PostgreSQL's actual memory usage, it's recommended to supplement pg_statio statistics with metrics related to I/O and memory utilization from the OS kernel. This holistic approach will provide a more accurate picture of PostgreSQL's performance and resource utilization.

- **Disk and index usage:** PostgreSQL has built-in mechanisms to collect statistics that provide insights into the size of tables and indexes over time. These statistics are invaluable for tracking changes in query performance as tables and indexes grow in size. As data volume increases, query execution times may also lengthen, and indexes may require more disk space. This may necessitate scaling up the disk space of the instance, partitioning data to optimize storage, or reevaluating the indexing strategy.

  Furthermore, monitoring table and index size can help identify unexpected growth, which may indicate issues with VACUUM operations not running as intended. Properly functioning VACUUMs are crucial for maintaining optimal performance in PostgreSQL. Regular monitoring and analysis of table and index size statistics can provide valuable insights into the health of the database and help identify potential areas for improvement to ensure efficient query performance and resource utilization. Tracking replication delay over time can provide valuable insights into the consistency of data updates across replica servers. A low replication delay indicates that the standby or replica servers are keeping up with the updates from the primary server in near real-time, ensuring that the data is replicated accurately and promptly. On the other hand, a high replication delay can indicate potential issues, such as network latency, resource constraints, or configuration problems, that may affect the ability of the replica servers to catch up with the updates from the primary server.

2.3.4 Replication and reliability

- **WAL replication in PostgreSQL:** PostgreSQL ensures data integrity and performance by utilizing a write-ahead log (WAL) mechanism for handling data writes and updates. When a transaction is executed, PostgreSQL records the transaction in the WAL, rather than immediately updating the actual data page/block on disk. This approach allows for efficient data updates without sacrificing data reliability. After logging the transaction in the WAL, PostgreSQL checks if the affected data block is already in memory. If the block is in memory, it is updated in memory, and the block is marked as a "dirty page" to indicate that it has been modified. This optimization helps to minimize disk I/O and improve performance, as updates can be handled in memory without immediately writing them to disk.

- **Replication delay:** Monitoring replication delay is an essential part of PostgreSQL replication and reliability monitoring. Replication delay is measured as the time difference between the last write-ahead log (WAL) update received from the primary server and the last WAL update applied or replayed on disk on the standby or replica server. Collecting and graphing replication delay metrics over time can help identify trends and patterns, allowing for proactive monitoring and troubleshooting. For example, sudden spikes in replication delay may indicate a network issue or a heavy load on the replica server, while a persistent increase in replication delay may indicate a need for tuning or optimization in the replica server configuration.

3 Results/Screenshots

We have made three tools to provide different kinds of analytics, Following are the details for the same.

3.1 Python CLI's app integrated with Grafana

Python is used to connect to the database and fetch different metrics. These metrics are then streamed at a UDP socket, which can be used by users to integrate with different monitoring tools, currently, we have integrated it with Grafana: A multi-platform open-source analytics and interactive visualization web application.

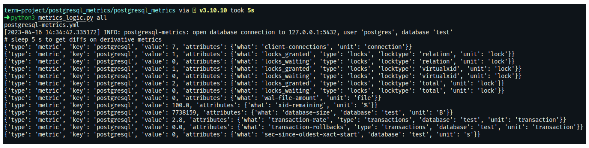

Figure 1: Raw metric streamed over UDP socket

Figure 2: Grafana integrated dashboard

3.2 Python GUI App

We have made an app that provides real-time query analysis and system resource usage in GUI.

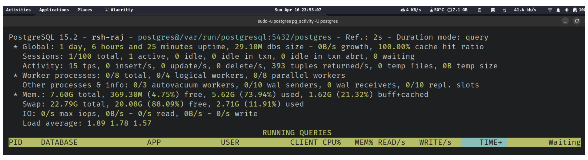

Figure 3: System resource usage

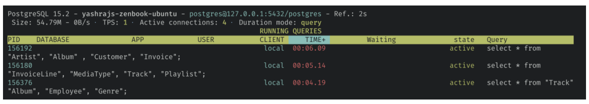

Figure 4: Live query analysis

3.3 GO CLI app to provide in-depth table and overall database analysis

This app provides two reports, whole DB analysis, and individual table analysis. Here is the sample output run on our HMS project's database. Click here to view the reports: data

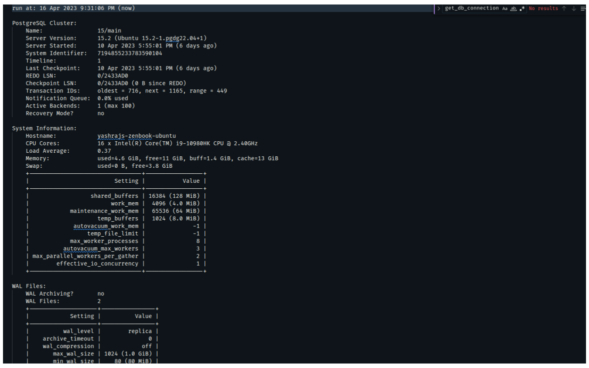

Figure 5: detailed in-depth analysis

References

- datadoghq.com/blog/postgresql-monitoring-tools
  Collecting metrics with PostgreSQL monitoring tools

- datadoghq.com/blog/postgresql-monitoring-tools
  Collecting metrics with PostgreSQL monitoring tools

- datadoghq.com/blog/postgresql-monitor
  Key metrics for PostgreSQL monitoring

- wiki.postgresql.org/wiki/Monitoring
  Monitoring

- postgresql.org/docs/current/monitoring
  Monitoring Database Activity

- docs.digitalocean.com/products/databases/postgresql
  How to Monitor PostgreSQL Database Performance

- severalnines.com/blog/key-things-monitor-postgresql-analyzing-your-workload
  Key Things to Monitor in PostgreSQL – Analyzing Your Workload

- fatdba.com/2021/03/24/how-to-monitor-your-postgresql-database-using-grafana
  How to monitor your PostgreSQL database using Grafana, Prometheus postgres_exporter

- grafana.com/solutions/postgresql/monitor
  Monitor PostgreSQL easily with Grafana