Optimizing Performance Using Query Plan Cache - QPC

To increase the efficiency of query execution, ClustrixDB employs a query plan cache (QPC) to store compiled versions of user queries. Two tables that are used in conjunction with cached query processing can also help identify problematic queries. They are system.qpc_queries and clustrix_statd.qpc_history. Together they allow us to isolate poorly performing queries and determine when a problem began. Additionally, the qpc_history table allows us to see if competing activity on a cluster may have contributed to a problem.

Using system.qpc_queries

Contents of Table

This table contains information regarding the queries run most recently on each node of a cluster. The full text of the query statement is available for tuning, if necessary.

Frequently Used Columns

These selected columns are used most frequently from system.qpc_queries:

Column	Description
nodeid	The ID of the node where the plan is cached. Each node maintains its own query cache. In many cases, queries are evenly distributed across the nodes, but in some cases a single client or process is responsible for an inordinate amount of load on a single node. In this case, you can filter on nodeid to list only its entries.
database	The database from which the query was executed.
statement	The text of the original query used for compilation is stored here. The query itself is parameterized when cached, meaning that the constant values supplied in the query (i.e. the N in WHERE id = N) are not included in the cached plan. When the QPC is utilized, the new parameters are substituted into the compiled query appropriately.
query_id	Unique identifier assigned to the query.
plan_id	Unique identifier assigned to differentiate multiple plans generated for the same query.
exec_count	Number of times this query plan has been executed since the plan was created.
exec_ms	Amount of CPU time (milliseconds) that has been spent executing this query plan since the plan was created. This figure does not include query compilation time.

Useful Query for qpc_queries

Locating and Tuning Resource Intensive Queries

The qpc_queries table can be helpful to locate queries that are using the most CPU resources. From there, one can determine if such queries can be tuned to impact their overall performance. For example, to find the top three queries on the cluster that have used the most cumulative CPU time, issue the following query: (runtime_ns is also a viable column to assess in lieu of exec_ms in this example.)

```
sql> SELECT nodeid, exec_count, exec_ms, exec_ms/exec_count as avg_ms, left(statement,100)
     FROM system.qpc_queries
     ORDER BY exec_ms desc
     LIMIT 3;
+--------+------------+----------+------------
+------------------------------------------------------------------------------------------------------+
| nodeid | exec_count | exec_ms  | avg_ms     | left(statement,
100)                                                                                    |
+--------+------------+----------+------------
+------------------------------------------------------------------------------------------------------+
|      3 |         65 |  2226593 | 34255.2769 | SELECT count(*) FROM files WHERE service_id = 9788   AND
mod_time > TIMESTAMPADD(MINUTE,-15,NOW()) |
|      2 |     606833 |  2087750 |     3.4404 | UPDATE files SET mod_time=now() WHERE id =
7617351                                                 |
|      1 |    1309577 |  1422334 |     1.0861 | SELECT * FROM files WHERE id =
12684388                                                                |
+--------+------------+----------+------------
+------------------------------------------------------------------------------------------------------+
3 rows in set (0.01 sec)
```

From the results, we can see that the first query averages almost 35 seconds per execution and is a candidate for investigation. The EXPLAIN and SHOW CREATE TABLE output reveal that no indexes are available or being used by that select (index_scan 1 := files.__base_files):

```
sql> EXPLAIN SELECT count(*)
     FROM files
     WHERE service_id = 9788
     AND mod_time > TIMESTAMPADD(MINUTE,-15,NOW());
+-----------------------------------------------------------------------------------------------------------
---------+-----------+-----------+
|
Operation
| Est. Cost | Est. Rows |
+-----------------------------------------------------------------------------------------------------------
---------+-----------+-----------+
| row_count
"expr1"
|     31.84 |      1.00 |
|
stream_combine
|     31.76 |      0.82 |
|     compute expr0 := param
(0)                                                                                          |     10.23 |
0.27 |
|       filter isnotnull(param
(0))                                                                                       |     10.22 |
0.27 |
|         filter (1.mod_time > add_time_interval(current_timestamp(), param(2), param(1))) and (1.service_id =
param(3)) |     10.22 |      0.30 |
|           index_scan 1 := files.
__base_files                                                                        |     10.20 |      0.34
|
+-----------------------------------------------------------------------------------------------------------
---------+-----------+-----------+
6 rows in set (0.01 sec)

sql> SHOW CREATE TABLE files\G
*************************** 1. row ***************************
       Table: files
Create Table: CREATE TABLE `files` (
  `service_id` int(11),
  `mod_time` datetime
) CHARACTER SET utf8 /*$ REPLICAS=2 SLICES=3 */
1 row in set (0.00 sec)
```

To improve performance, we create a compound, non-unique index on service_id and mod_time by issuing the following command:

```
sql> ALTER TABLE files ADD INDEX service_mod (service_id, mod_time);
Query OK, 0 rows affected (0.82 sec)
```

After adding this index, the average execution time for this query dropped from 35 seconds to 3 milliseconds.

Using clustrix_statd.qpc_history for Historical Analysis

Contents of Table

Summarized statistics for queries from the past seven days are retained in the table clustrix_statd.qpc_history. For each query tracked, data is collected for query duration, the number of executions and the cumulative number of rows read, written, or updated. The statistics are then used to rank the queries relative to other queries running during the same period.

Statistics Gathering Rules

The queries summarized and tracked in qpc_history follow these two important rules:

1. Time Frames. The table contains data for one week (7 days) and statistics are retained in two distinct time frames. (Statistics older than seven days are purged.)
    a. Stats for the most recent (rolling) 24 hours are tracked every five minutes. Current day statistics for the 100 most active queries are collected, summarized, ranked, and stored in qpc_history every five minutes. That translates to information saved on nearly 30K query executions for the most recent 24 hours. (60 mins/5 * 24 * 100 = 28,800).
    b. Stats for the preceding six days are compressed into hourly segments. After 24 hours, the data collected from active queries every five minutes is compressed into hourly statistics.
2. Ranking. The entries within each time frame (either five minutes or one hour) are ranked based on how much CPU time the query takes. Those rows with a rank of 1 take more time than those that have been ranked as 100.

Data for every five minutes is capped at the 100 most active queries, so the ranks for each five-minute time slot will range from 1-100. Data that is consolidated and summarized hourly could have more than 100 rows and consequently more than 100 ranks.

Frequently Used Columns

These are the columns of clustrix_statd.qpc_history that are used most frequently:

Column	Description
query_key	Identifier shared between the same query during different tracking periods. Note that changes to the cluster (i.e. improved probability statistics, schema changes, node addition or removal, and a myriad of other conditions) can cause the different query_keys to be assigned to the same statement over time.
timestamp	Time for which the statistics were tracked. For stats of five-minute intervals this column is populated to the second - YYYY-MM-DD HH24:MI:SS. For older, summarized hourly stats, the minutes and seconds are zero - YYYY-MM-DD HH24:00:00.
database	The database from which the query was executed.
statement	This column contains a truncated version of the query that was executed (the first 8,196 of 65,535 characters). While this may not be directly useable for tuning, this column should contain enough information to allow for query recognition.
exec_count	Total number of executions for this query during the statistics gathering period.
exec_ms	Total time (milliseconds) that the query took to execute during the time period.
avg_exec_ms	Average execution time for the query (milliseconds). Queries with long avg_exec_ms times are potential candidates for optimization.
min_exec_ms	Minimum execution time for the query (milliseconds). These minimums and maximums are potentially more meaningful for hourly statistics than for five-minute summaries.
max_exec_ms	Maximum execution time for the query (milliseconds). Queries for which max_exec_ms significantly exceeds avg_exec_ms may indicate resource contention or blocking issues.
rows_read	Total number of rows read.
avg_rows_read	Average rows read per query execution. Queries that process a lot of rows may be candidates for optimization.
rank	Computer generated rank that compares numerous characteristics to determine a query’s impact on the cluster relative to other queries running during the same time. For example, trying to speed up queries that are ranked in the top 10 will likely have a bigger overall impact than optimizing queries that affect fewer overall system resources.
is_rollup	This column identifies the type of statistic. If 0, the statistic has been gathered during a five-minute interval. If 1, the statistic has been consolidated from multiple five-minute time frames to reflect summarized statistics for one hour.

Useful Queries for qpc_history

These samples show ways in which the qpc_history table can be used to identify potential performance issues. You will likely not use any of these examples verbatim, but they will provide some perspective relative to the information available from this table. Many samples below exclude the clustrix_statd database, which is the database containing qpc_history.

Frequently Run Queries

This statement will locate the 100 queries that have run most frequently within the past 24 hours. By including rank, we can see how the query compares to other frequently run queries. Frequently run queries, especially those with high execution times, may contribute to slow cluster performance.

```
sql> SELECT  query_key,
             min(rank),
             max(rank),
             database,
             left(statement,100),
             sum(exec_count) as calc_exec_count,
             round(avg(avg_rows_read)) as calc_avg_rows_read,
             round(avg(avg_exec_ms)) as calc_avg_exec_ms
     FROM    clustrix_statd.qpc_history
     WHERE   timestamp BETWEEN (now() - interval 24 hour) AND now()
     AND     database !='clustrix_statd'
     GROUP BY query_key
     ORDER BY calc_exec_count     DESC,
              calc_avg_rows_read  DESC
     LIMIT 100;
```

Queries Reading a Lot of Rows

This statement will locate 100 queries that have returned the most number of rows during the past 24 hours. Even a slight improvement of a high-volume query could improve performance.

```
sql> SELECT query_key,
            min(rank),
            max(rank),
            database,
            left(statement,100),
            sum(exec_count) as calc_exec_count,
            round(avg(avg_rows_read)) as calc_avg_rows_read,
            round(avg(avg_exec_ms)) as calc_avg_exec_ms
     FROM   clustrix_statd.qpc_history
     WHERE  timestamp BETWEEN (now() - interval 24 hour) AND now()
     AND    database !='clustrix_statd'
     GROUP BY query_key
     ORDER BY calc_avg_rows_read DESC,
              calc_exec_count DESC
     LIMIT 100;
```

Long Running Queries

This query shows the 100 queries that have taken the longest to run within the past 24 hours. Long-running queries that affect few rows may be underperforming.

```
sql> SELECT query_key,
            min(rank),
            max(rank),
            database,
            left(statement,100),
            sum(exec_count) as calc_exec_count,
            round(avg(avg_rows_read)) as calc_avg_rows_read,
            round(avg(avg_exec_ms)) as calc_avg_exec_ms
     FROM   clustrix_statd.qpc_history
     WHERE  timestamp BETWEEN (now() - interval 24 hour) AND now()
     AND    database !='clustrix_statd'
     GROUP BY query_key
     ORDER BY calc_avg_exec_ms  DESC,
              calc_exec_count   DESC
     LIMIT 100;
```

Top-Ranked Queries

Look at the 100 most recent queries that have used the most resources and thus have been ranked in the top 3. The LIMIT combined with the ORDER BY of this sample means the top-ranked queries shown will be from within the past 24 hours. To include additional days, expand the limit or target hourly rollups: (is_rollup = 1).

```
sql> SELECT timestamp,
            rank,
            database,
            left(statement,100),
            exec_count,
            avg_rows_read,
            avg_exec_ms
     FROM   clustrix_statd.qpc_history
     WHERE  rank < 4
     ORDER BY timestamp DESC, rank ASC
     LIMIT 100;
```

Some Helpful Tips and Tricks When Using qpc_history

- If you have an idea of when a problem began, search for a specific time frame within qpc_history.
- To isolate activity performed on a given table, search for a specific table name in the statement column. For example: WHERE UPPER(statement) like '%YOUR_TABLE_NAME%'
- Look at rows_written if you're investigating statements that write to the database.
- If a query seems to be running slower than normal, compare the avg_exec_ms of that query from a time when it was running fine and investigate what else was running during the same time period. Sometimes poor performance is the result of contention or blocking issues.
- Researching qpc_history in conjunction with the query logs is often helpful.