Query Store

Last updated by | Lisa Liu | Nov 6, 2020 at 10:35 AM PST

Query Store

Monday, October 22, 2018
5:07 PM

Query Data Store (QDS) is a component that captures and stores data about queries including various runtime stats and plans, and exposes the result as views. Other component Performance/ Workload Insights (WI) and Database Tuning Advisor (DTA) use QDS to provide insights and recommendations.

The counters/ metrics collection is mainly done by the engine. PGQDS will collect the following **per query**:

- I/O metrics – local and shared block reads, writes, dirtied and hit; temp block reads and writes
- Execution time – query execution time
- [Future] CPU metrics – none right now
- [Future] Memory metrics – none right now
- [Future] Network metrics – none right now
- [Future] Wait/ block data - none right now

Notes:

- The corresponding server-level metrics (i.e. not query-level metrics) will separately flow into Kusto - some of them already do; this is out of scope of QDS, QDS concerns per only.

1. Postgresql Extension

1. EXTENSION: COMPARISON WITH PG_STAT_STATEMENTS

For V1, the core QDS module will be implemented as a shared PostgreSQL extension **pg_qds**. The starting point for the code is the existing PG extension pg_stat_statements (pgss does not aggregate data by time windows (it collects stats for a query for all of time) and requires a server restart (on load and unload) since it's a shared module.

Differences between pgss and pg_qds:

Timeframe	Description
March	• Aggregate within a time window – with configurable (default 15 minutes) interval. • Store data into database tables. Build indexes on tables. Serve data via views built on top of the tables. • Copyrighted to Microsoft. • General code quality improvements. • Store un-scrubbed raw query along with scrubbed query.
Future	• Use local memory instead of shared memory, so server restart not required to use extension. • C++ extension.

1. EXTENSION: CUSTOMER EXPERIENCE

In Private Preview, the extension will be enabled for all - I.e. already loaded as a shared preload library and whitelisted for all customers in all regions.

- Customer has to do CREATE EXTENSION once – this will create the necessary schema for QDS
- Customer has to turn the extension ON (via config option) and reload the config

This will result in QDS collecting and storing data from thereon.

1. EXTENSION: HOSTING - SHARED PRELOAD LIBRARY

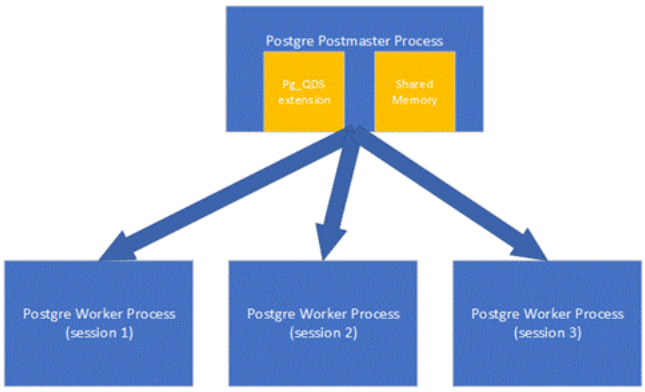


Figure 2. PostGreSQL process layout

In v1.0, the extension will be loaded as a dll in the Postmaster process space since it's a shared preload library. In the future when we make this a session-preload extension, it wi the worker process space. The extension will be a shared preload library (since it uses shared memory) – so will require a server restart on load or unload, but it will be enabled a

for Private Preview customers, and enabled and turned off for all other customers.

The decision to keep it enabled for all customers in Private Preview is so that the customers that will use QDS don't require a server restart when they decide to start using.

1. EXTENSION: ON/OFF CONFIG SWITCH

It will be possible to turn the extension on or off using a config option. This extra switch is being introduced because QDS will be shipped and enabled by default for all customers Preview, but will be exercised only by select Private Preview customers – these customers will be instructed to turn it on via the config switch.

Extension is loaded	On/Off Config	Does QDS record queries?	Behind the scenes
Yes	**On**	**Yes**	
No	**On**	No	
Yes	**Off**	No	Pg_qds functions will be called – but the function will check the config value and return since it's "off"
No	**Off**	No	

Process to enable qds after extension is loaded:

Run following query in pgadmin to set the configuration variable

ALTER SYSTEM SET pg_qds.turned_on = on

Reload the configuration variable by running this query in pgadmin

SELECT pg_reload_conf();

Process to disable qds:

Run following query in pgadmin to set the configuration variable

ALTER SYSTEM SET pg_qds.turned_on = off

Reload the configuration variable by running this query in pgadmin

SELECT pg_reload_conf();

Users do not need to restart the server for those operations.

User has to be superuser to run Alter SYSTEM SET clause. Proposed: to update azure CLI to help allow user to toggle this switch for the Private Preview. Later we also need to upd server parameters blade to allow user to update those configurations

1. EXTENSION: AGGREGATION DETAILS

QDS aggregates prior to query writes – this is with the understanding that a lot of queries in a standard OLTP system are similar – so, recording once instead of multiple times bot essence and reduces storage needs.

We rely on one pg background worker diagram below) to wake up periodically and dump tracked queries statistics into qds tables, then clean the old statistics from a shared-mem table. Through this way, we got the query statistics aggregation for the previous window. Every time window (default 15 minutes, minimum granularity 1 minute), queries with th structure are clustered into the same bucket and their statistics are collected as one; the complete raw query for the very first query in this bucket is noted and later written to th

E.g.:

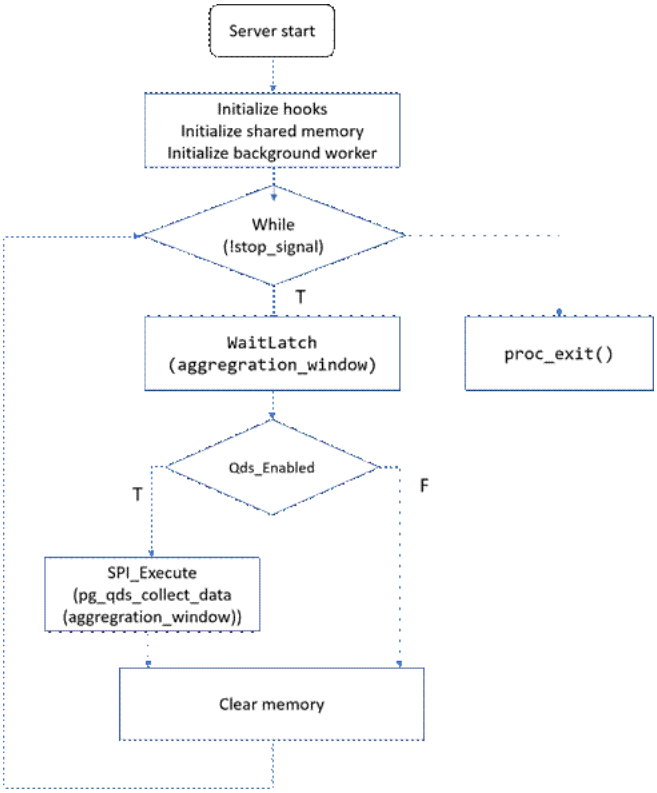
Query1: select * from foo where v = 'bar1' (local_read_blocks = 5, time: t1)

Query2: select * from foo where v = 'bar2' (local_read_blocks = 6, time: t1+1 min)

Query1 and Query2 are clustered into the same bucket since they have the same general structure (pgss makes this determination). There will be one entry in pg_qds tables for t with local_read_blocks = 11; Query1's full text will be stored but not Query2's.

The aggregates are cached in a shared memory section reserved by pg_qds until they are flushed.

Below is a flowchart for the background worker.



1. EXTENSION: QUERY NORMALIZATION

Query normalization is implemented by fingerprinting queries, selectively ignoring extraneous information (constants) from the query. After this Jumble process, the queryjumble hashed into a 32 bit integer as the queryId. The constant is replaced by a '?' symbol in the query.

1. EXTENSION: CONFIGURATION OPTIONS

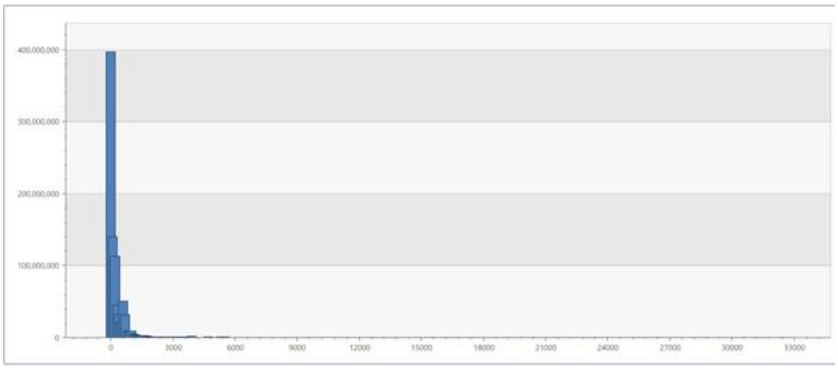
The pg_qds extension will be loaded for all customers; Private Preview customers will be instructed to turn the extension ON. Configuration options for V1:

For all the configs below, user can reload the configuration settings to make the change take effect.

Config Option	Default	Acceptable values	Changing requires server restart	Description
pg_qds.**turned_on**	off	on, off	no	Whether QDS is enabled or not
pg_qds.aggregration_window_in_minutes	15 mins	1 min – 60 [mins	yes	The aggregation window
pg_qds.max_queries_number	5000	10 – INT_MAX	no	Maximum # of queries to be tracked by QDS
pg_qds.max_query_length	6000	100 – TBD	no	Maximum length of the query, if longer than this length, query truncated.

If trying to set the value out of the allowed range, user will get an error like "ERROR: 10 is outside the valid range for parameter "pg_qds.aggregration_window_in_minutes" (60 .. 3600)", and d will be set.

pg_qds.max_query_length: We did a query length distribution analyze for the query length in North Europe (plot is attached below), and found the threshould value at 3000 would be a good thre tracking majority queries at full length. For queries longer than this threshould, they will get truncated for to avoid slow text storage and holding the lock too long.



1. EXTENSION: MAX-SIZE

Implemented for the max #rows cached in shared memory prior to flushing.

[Max #rows in table not implemented yet]

When the max number of queries has been recorded, every new query triggers a "Garbage Collection" the same way it happens for pgss. This GC finds the most unpopular query c by call count and deletes it from the table[yun add the description on the delete of the most unpopular query]

The popularity of one query in pgss is close to the total time (singlequery duration * execute count) of that query get executed, with some decay factor multiplied to accommodate the old verse ne

1. INSTALLATION

1. FRESH INSTALL

Current QDS extension installation is the same as any other extension need to be loaded via shared_preload_libraries. In general user can follow those steps:

2. make sure 'pg_qds' is added to shared_preload_libraries setting in postgres.conf (we will make this change for user during the rollout)
3. (optional) update configuration options for qds
4. create extension by run 'CREATE EXTENSION pg_qds'
5. turn on pg_qds by the on/off switch mentioned previously
1. ALTER SYSTEM SET `pg_qds.enabled`= on
2. Then reload configuration by running 'SELECT pg_reload_conf();'

1. UPGRADE STORY

Future QDS upgrades might come with:

- Engine changes to record more metrics – will require server restart
- Extension updates – will require server restart
- Schema updates – breaking schema updates will have to be accompanied by data migration capability

1. PG VERSION SUPPORT

For the Private Preview, we are supporting PG 9.6. Later we will support PG 9.5, PG 10 and any other new version.

1. QDS VERSIONINdG

The Private Preview version of pg_qds will be 0.1. The intention is to be fully backward-compatible with respect to functionality, schema, data, config – so the first version will exp configs and the backend is designed to be more flexible than the pgss schema. Specific backend details appear later in this document.

1. PUBLISHING TO MDS

Node agent will collect data periodically and emit only compliant columns to MDS table for internal team troubleshooting purpose. Fields except query_sql_text will be pushed t are standard headers from MonDmPgSqlStatStatements (pulling form pgss).

Introduce a new event called MonDmPgSqlQueryStore. It basically has all data except query text (GDPR) plus standard server metadata info.

- View collection definition:

<query TableName ="MonDmPgSqlQueryStore" Scope="Server" FrequencyInSeconds="3600" TimeoutInSeconds="120" Owner="Monitoring" Mds="true" Disabled="false" ConfigPackageName="SQL.Config" OptInFeatureSwitch="EnableQueryDataStoreExtension" ShouldQuerySystemTable="true">

SELECT runtime_stats_entry_id, user_id, db_id, query_id, plan_id, start_time, end_time, calls, total_time, min_time, max_time, mean_time, stddev_time, rows, shared_blks_h 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_reac blk_write_time

FROM query_store.qs_view

-- remember to update the where clause when FrequencyInSeconds get updated

-- look back 55 seconds further to compensate QS data latency.

Where end_time >= current_timestamp - '3655 seconds'::INTERVAL;

</query>

- Window is 1hour 3600 seconds
- Look back 55 seconds more to accomdate any data latency. Since minimum aggregation window for QS is 1 mins, the change of duplicated data is very small.

- MDS Table

<DirectoryWatchItem

eventName="MonDmPgSqlQueryStore"

storeType="!STORETYPECENTRALBOND!"

duration="PT1H"

startTimeDeltaInSeconds="!STARTTIMEDELTA!"

account="WASDMon"

priority="Low">

<Directory><![CDATA[Concat("", GetEnvironmentVariable("MONITORING_RELATIVEDIRECTORY"), "\Public\MonDmPgSqlQueryStore")]]></Directory>

</DirectoryWatchItem>

1. PRIVATE PREVIEW ROLLOUT PLAN

We have two options for the private preview rollout. The one highlighted in bold is the proposed method.
1 - Private build and deploy to specific customer.
Pro: Do not need to have code fully complete.
Con:
Any further updates will reset the feature. We need to ping the server to block the update.
Same user cannot have other feature (eg dta) tested.
Concern: right now we have a lot of hotfix rollout everytime, it's a bad idea to block certain user.

2 - Global availability then guide specific customers to turn on the extension.
The extension will be shipped to all customers, it will be turned on only for Private Preview customers.
Pro: it will not block further updates or other private feature tests.

Option 2 is the POR.

1. Backend database design

Query Data Store has three main types of data structures in the backend: a shared-memory hash table to store the aggregated query counters (updated when a new query is issu engine through engine hooks), files transiently to store the query texts and internal tables as persistent storage for the periodical dumps.

The statistics are gathered from the engine through several hooks during the query life cycle (as for pg_stat_statements):

- post_parse_analyze_hook
- ExecutorStart_hook
- ExecutorRun_hook
- ExecutorFinish_hook
- ExecutorEnd_hook
- ProcessUtility_hook

For the periodical flush, a background worker (created when the extension is installed) is woken up from idle state (through a latch mechanism) and dump the content of shared m permanent tables in a normalized schema. This dump is executed using a pg function installed at extension creation (**pg_qds_collect_data(interval_window)**) through the Server Interface (SPI). Finally, after the periodical dump the shared memory content is cleaned.

In order to reset pg_qds extension, cleaning up permanent tables another function is exposed (**pg_qds_reset()**).

The information gathered by QDS is exposed through **pg_qds_view**.

2. TABLES

For March time frame, the permanent tables created on QDS schema are:

- **query_store_query_text**: stores one row per unique query statement text in the system. This starts at the first character of the first token and ends at the last character of t (constraint to the length specified in the global configuration file). The primary key for this table is query_id.
- **query_store_runtime_stats**: stores the runtime statistics gathered for each query. The primary key for this table is (user_id, dbid, query_id)
- **query_store_runtime_stats_interval**: stores information of start_time and end_time for each triggered interval by the background worker. The primary key for this table is a
- We might create a separate Counters table so more counters can be added later easily without a schema revision.

Tables	Description
msdb.query_store_query_text	Information about captured query texts.
msdb.query_context_settings*	Different runtime combinations of semantics-affecting context settings (SET options that influence plan shape, language ID, ...)
msdb.query_store_query	Unique combination of query text and context settings
msdb.query_store_plan*	Information about plans SQL Server uses to execute queries in the system.
msdb.query_store_runtime_stats_interval	Aggregation intervals (time windows) created in Query Store.
msdb.query_store_runtime_stats	Runtime statistics for executed query plans, aggregated on per-interval basis
msdb.database_query_store_options*	QueryStore config options

3. VIEWS

For march time frame, the information gathered from qds will be exposed through a unique view ***pg_qds_view,*** with the following columns:

Name	Type	Description
userid	oid	OID of user who executed the statement
dbid	oid	OID of database in which the statement was executed
queryid	bigint	Internal hash code, computed from the statement's parse tree
query	text	Text of a representative statement
calls	bigint	Number of times executed
start_time	datetime	Start time of the interval for this recording
end_time	datetime	End time of the interval for this recording
total_time	double precision	Total time spent in the statement, in milliseconds
rows	bigint	Total number of rows retrieved or affected by the statement
shared_blks_hit	bigint	Total number of shared block cache hits by the statement
shared_blks_read	bigint	Total number of shared blocks read by the statement
shared_blks_dirtied	bigint	Total number of shared blocks dirtied by the statement
shared_blks_written	bigint	Total number of shared blocks written by the statement
local_blks_hit	bigint	Total number of local block cache hits by the statement
local_blks_read	bigint	Total number of local blocks read by the statement
local_blks_dirtied	bigint	Total number of local blocks dirtied by the statement
local_blks_written	bigint	Total number of local blocks written by the statement
temp_blks_read	bigint	Total number of temp blocks read by the statement
temp_blks_written	bigint	Total number of temp blocks written by the statement
blk_read_time	double precision	Total time the statement spent reading blocks, in milliseconds (if track_io_timing is enabled, otherwise zero)
blk_write_time	double precision	Total time the statement spent writing blocks, in milliseconds (if track_io_timing is enabled, otherwise zero)

Created with Microsoft OneNote 2016.

How good have you found this content?