Monitoring and Debugging PostgreSQL

- Ashutosh Sharma | 2018.02.22

Agenda

- Database Logs
- The PostgreSQL Statistics Collector
- Statistics Views
- Wait Events
- Monitoring query performance using pg_stat_statements
- Useful debugging tools

Database Logs

- Where Are My Database Logs?

```
test=> show log_destination;
log_destination
----------------
stderr
(1 row)
```

- log_destination
  - stderr
  - syslog
  - csvlog
  - eventlog

Database Logs

- Stderr

```
test=> show logging_collector
logging_collector
---------------------
on
(1 row)

test=> show log_directory;
log_directory
---------------
log
(1 row)
```

Database Logs

- Stderr (Continued)

```
test=> show data_directory;
data_directory
------------------------------
/home/ashu/pgsql/tmp/pgsql/data
(1 row)


test=> show log_filename;
log_filename
------------------------------
postgresql-%y-%m-%d_%h%m%s.log
(1 row)
```

Database Logs

- What Information Do Logfiles Contain?
  - General LOG information
  - Error messages
  - Hints …

Database Logs

- General Log Information

  LOG: database system was interrupted; last known up at 2018-02-02 12:17:28 IST

  LOG: database system was not properly shut down; automatic recovery in progress

  LOG: server process (PID 66538) was terminated by signal 11: Segmentation fault
  DETAIL: Failed process was running: INSERT INTO tab1 VALUES(1);

- Errors

  ERROR: relation "tab1" does not exist at character 13

  FATAL: role "hacker" does not exist

  PANIC: invalid index offnum: 50
  STATEMENT: INSERT INTO tab1 VALUES (10000);

Database Logs

- Hints

LOG: checkpoints are occurring too frequently
HINT: Consider increasing the configuration parameter "checkpoint_segments".

ERROR: function non_existing_function() does not exist at character 8
HINT: No function matches the given name and argument types. You might need to add explicit type casts.
STATEMENT: select non_existing_function();

Database Logs

- How to log slow running SQL statements, query waiting duration, amount of temporary files generated…

- Login to the database as a superuser and then execute the following SQL statements:

```
test=# ALTER SYSTEM SET log_min_duration_statement to '20s';

test=# ALTER SYSTEM SET log_lock_waits to 'on';

test=# ALTER SYSTEM SET log_temp_files to '0';

test=# SELECT pg_reload_conf();
pg_reload_conf
----------------
t
(1 row)
```

Agenda

- Database Logs
- The PostgreSQL Statistics Collector
- Statistics Views
- Wait Events
- Monitoring query performance using pg_stat_statements
- Useful debugging tools

The PostgreSQL Statistics Collector

- As the name indicates, postgres statistics collector process collects statistics about the database.
- It's an optional process with the default value as on and it's behavior is dependent on a set of track parameters, which guides the stats collector about which metrics it needs to collect from the running instance.
- Each individual processes transmit new statistical counts to the collector process just before going to idle state, the collector process then collects the stats sent by backend process and writes the data into some stats file which can be read via number of views.

The PostgreSQL Statistics Collector (Continued)

- Track parameters associated with Statistics Collector

  - track_activities : enables monitoring of the current command being executed by any backend process , on by default.

  - track_activity_query_size : decides the space in terms of bytes reserved to store user query, 1024 is the default value.

  - track_counts : allows the stats collector process to collect all the base table and index table access related statistics and store them into the pg_stat_tmp location in the form of db_<database_oid>.stat or globals.stat, on by default.

The PostgreSQL Statistics Collector (Continued)

- track_io_timing : enables monitoring of disk blocks read and write time i.e. the time spent on disk blocks read/write operations by each backend process, off by default.

- track_functions : controls tracking of metrics about the user level functions, default value is none meaning that it won't be tracking any type of user functions, can be set to pl, C, all..

The PostgreSQL Statistics Collector (Continued)

- BEGIN;
- SET track_io_timing = ON;
- EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tab1;

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------
Seq Scan on tab1  (cost=0.00..607.08 rows=40008 width=330) (actual time=8.318..38.126 rows=40009 loops=1)
   Buffers: shared read=207
   I/O Timings: read=30.927
 Planning time: 161.577 ms
 Execution time: 42.104 ms
```

- COMMIT;

➔ TakeAway : Out of 42ms of time spent on query execution, ~32ms were spent on I/O operation.

The PostgreSQL Statistics Collector (Continued)

Now, What if the previous query is executed once more...

- BEGIN;
- SET track_io_timing = ON;
- EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tab1;

```
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------
---
Seq Scan on tab1  (cost=0.00..607.08 rows=40008 width=330) (actual
time=0.004..7.504 rows=40009 loops=1)
   Buffers: shared hit=207
 Planning time: 0.367 ms
Execution time: 11.478 ms
```

- COMMIT;

➔ TakeAway : The query is just about 31ms faster and well, the reason is very obvious, the data got cached this time..

Agenda

- Database Logs
- The PostgreSQL Statistics Collector
- Statistics Views
- Wait Events
- Monitoring query performance using pg_stat_statements
- Useful debugging tools

Statistics view

- pg_stat_database
  - The database-level statistics are saved in the pg_stat_database view.
  - It contains one row for each database, showing database-wide statistics.
  - It shows the informations such as the number of backend processes currently connected to a database, number of transactions committed or rollback in a particular database, number of data blocks read from disk or the total time spent in disk read or write activities.
  - For details on the layout of pg_stat_database statistics view, have a look at the documentation about pg_stat_database

Statistics view (Continued)

- How To Make Sense Of Data In Pg_stat_database ?

- Getting statistics like the cachehit ratio, dml statistics, transaction statistics etc. for a particular database ...
  - CREATE VIEW get_db_stats AS SELECT datname, round((blks_hit::float / (blks_read+blks_hit+1) * 100)::numeric, 2) as cachehitratio, xact_commit, xact_rollback, tup_fetched, tup_inserted, tup_updated, tup_deleted FROM pg_stat_database WHERE datname NOT IN ('template0', 'template1')ORDER BY cachehitratio desc;

```
postgres=# select * from get_db_stats;
------------------+-----------
datname           | postgres
cachehitratio     | 94.32
xact_commit       | 22
xact_rollback     | 2
tup_fetched       | 1236
tup_inserted      | 150
tup_updated       | 10
tup_deleted       | 10
```

Statistics view (Continued)

- How To Make Sense Of Data In Pg_stat_database ?

- Finding the total number of temp files generated in a database...
  - SELECT temp_files, temp_bytes FROM pg_stat_database WHERE datname = current_database();
  - SHOW work_mem;
  - SET work_mem to 'some_higher_value';

- Monitoring database loads...
  - SELECT numbackends , xact_commit , xact_rollback, blks_read + blks_hit as total_buffer_read FROM pg_stat_database where datname NOT IN ('template0', 'template1') order by xact_commit desc;

Statistics view (Continued)

- pg_stat_all_tables
  - The pg_stat_all_tables view contains one row for each table (which includes system table or a user table or may be TOAST table) in the current database, showing statistics about accesses to that specific table.
  - The pg_stat_user_tables and pg_stat_sys_tables views contain the same information as pg_stat_all_tables, but are restricted to only user and system tables respectively.
  - For details on the layout of pg_stat_all_tables statistics view, have a look at the documentation about pg_stat_all_tables

Statistics view (Continued)

- How To Make Use Of Data In Pg_stat_all_tables ?
  - Finding top 10 most read tables in the database

SELECT relname, idx_tup_fetch + seq_tup_read as TotalReads FROM pg_stat_all_tables WHERE idx_tup_fetch + seq_tup_read != 0 order by TotalReads desc LIMIT 10;

```
   relname     | totalreads
---------------+------------
 pg_class      |      27637
 pg_attribute  |        692
 pg_opclass    |        386
 pg_index      |        259
 pg_database   |        210
 pg_operator   |        148
 pg_proc       |         89
 pg_amproc     |         77
 pg_amop       |         56
 pg_type       |         47
(10 rows)
```

Statistics view (Continued)

- How To Make Use Of Data In Pg_stat_all_tables ?

- Autovacuum monitoring
  - SELECT schemaname, relname, last_autovacuum, last_autoanalyze FROM pg_stat_all_tables WHERE relname='tab1';

- Checking for the dead tuples count to see if a table needs to be manually VACUUMED or not..
  - SELECT relname, last_vacuum, n_dead_tup, last_analyze FROM pg_stat_all_tables where relname='tab1';

- Finding the ratio of index scan to seq scan on a table.
  - SELECT sum(idx_scan)/(sum(idx_scan) + sum(seq_scan)) as idx_scan_ratio FROM pg_stat_all_tables WHERE schemaname='public';

Statistics view (Continued)

- pg_stat_activity
  - The pg_stat_activity view shows what activity is currently happening on your PostgreSQL database server.
  - It contains one row per server process and shows some very useful informations like the current state of a running backend process, the query that the client process is currently running, query start time or transaction start time, the wait event on which the client is currently waiting and so on...
  - In short, pg_stat_activity basically provides a way to get a snapshot of what every client on the server is currently doing.
  - For details on the layout of pg_stat_activity statistics view, have a look at the documentation about pg_stat_activity

Statistics view (Continued)

- Querying Pg_stat_activity For ...
- Finding out how many queries are currently being executed on your database
  - CREATE VIEW get_active_sessions AS SELECT datname, count(*) AS open, count(*) FILTER (WHERE state= 'active') AS active, count(*) FILTER (WHERE state = 'idle') AS idle, count(*) FILTER (WHERE state ='idle in transaction') AS idle_in_trans FROM pg_stat_activity GROUP BY datname;

```
postgres=# select * from get_active_sessions;
-[ RECORD 1 ]-+---------
datname       | postgres
open          | 2
active        | 1
idle          | 1
idle_in_trans | 0
```

Statistics view (Continued)

- Querying Pg_stat_activity For ...
- Finding and killing long running idle database connections
  - CREATE VIEW kill_idle_sessions AS SELECT pg_terminate_backend(pid) FROM pg_stat_activity WHERE datname = 'postgres' AND pid <> pg_backend_pid() AND state in ('idle', 'idle in transaction', 'idle in transaction (aborted)', 'disabled') AND state_change < current_timestamp - INTERVAL '5' DAY;

postgres=# select * from kill_idle_sessions;

```
-[ RECORD 1 ]--------+--
pg_terminate_backend | t
```

postgres=# select * from get_active_sessions;

```
-[ RECORD 1 ]-+---------
datname       | postgres
open          | 1
active        | 1
idle          | 0
idle_in_trans | 0
```

Statistics view (Continued)

- Querying Pg_stat_activity For ...

- Detecting long running queries or transactions...

  If you want to find out a query running for very long time, say more than 2 hours on PostgreSQL, you can run the following command,

  - SELECT pid, datname, usename, client_addr, now() - query_start as "runtime", query_start, wait_event_type, wait_event, state, query FROM pg_stat_activity WHERE now() - query_start > '2 hours'::interval ORDER BY runtime DESC;

- Wait Event Monitoring for a long running queries

  SELECT pid, now() - query_start as "runtime", wait_event_type, wait_event, state, query FROM pg_stat_activity WHERE now() - query_start > '5 hours'::interval ORDER BY runtime DESC;

Statistics view (Continued)

- Querying Pg_stat_activity For ...

- Finding blocked sessions
  - SELECT datname, usename, application_name, now()-backend_start AS "Session duration", pid, query FROM pg_stat_activity WHERE state='active' AND wait_event IS NOT NULL;

- Figuring out where queries come from ...
  - SELECT application_name, client_addr, client_hostname, client_port from pg_stat_activity;

Agenda

- Database Logs
- The PostgreSQL Statistics Collector
- Statistics Views
- Wait Events
- Monitoring query performance using pg_stat_statements
- Useful debugging tools

Wait Events

- Wait events report where the backend is waiting
- Two columns in pg_stat_activity
  - wait_event_type
  - wait_event
- wait_event_type - The type of event for which the backend is waiting, if any; otherwise NULL.
- wait_event - Wait event name if backend is currently waiting, otherwise NULL. Some of the possible values for wait_event_type and wait_events are listed in the following slides

Wait Events (Continued)

- wait_event_type
  - LWLock : The backend is waiting for a lightweight lock.
  - Lock : The backend is waiting for a heavyweight lock.
  - Client : The backend is in idle state waiting on a socket for the query from user.
  - Acitivity : The auxiallary processes are waiting for some activity to happen.
  - BufferPin : The server is waiting to acquire a buffer pin where no other processes has acquired pin on the same buffer.
  - IO : The server process is waiting for an IO to complete.
  - Timeout : The server process is waiting for a timeout to expire.
  - IPC : The server process is waiting for some activity from another process in the server.

Wait Events (Continued)

- wait_event
  - WALWriteLock : Waiting for the WAL Buffers to be written into a disk.
  - CheckPointLock : Waiting to perform a checkpoint.
  - ShmemIndexLock : Waiting to find or allocate a space in shared memory
  - clog : Waiting for I/O on a clog buffer.
  - buffer_io : Waiting for I/O on a data page to complete.
  - transactionid : Waiting for a transaction to finish.
  - Relation : Waiting to acquire lock on a relation.
  - tuple : Waiting to acquire lock on a tuple.

Wait Events (Continued)

synchronous_commit = on;
pgbench -c 20 -j -T 2 300 pgbench

test=# select pid, wait_event, wait_event_type, state from pg_stat_activity WHERE wait_event is not NULL;

```
  pid    |  wait_event    | wait_event_type | state
---------+----------------+-----------------+--------
 25632   | transactionid  | Lock            | active
 25633   | WALWriteLock   | LWLockNamed     | active
 25635   | WALWriteLock   | LWLockNamed     | active
 25636   | transactionid  | Lock            | active
 25638   | transactionid  | Lock            | active
 25640   | WALWriteLock   | LWLockNamed     | active
 25642   | WALWriteLock   | LWLockNamed     | active
.........
```

- Most of the backend sessions are waiting on a WALWriteLock i.e. for WALBuffers to be flushed.

Wait Events (Continued)

synchronous_commit = off;
pgbench -c 20 -j -T 2 300 pgbench

test=# SELECT pid, wait_event, wait_event_type, state, query from pg_stat_activity WHERE wait_event is not NULL;

```
  pid  |  wait_event   | wait_event_type | state
-------+---------------+-----------------+--------
 26201 | transactionid | Lock            | active
 26203 | transactionid | Lock            | active
 26204 | transactionid | Lock            | active
(3 rows)
```

- No more contention on WALWriteLock...

Agenda

- Database Logs
- The PostgreSQL Statistics Collector
- Statistics Views
- Wait Events
- Monitoring query performance using pg_stat_statements
- Useful debugging tools

pg_stat_statements

- pg_stat_statements is an extension module that tracks the execution statistics of all SQL statements executed by a server and stores them in a pg_stat_statements table (which is basically a hash table).
- It's a module that needs to be loaded and is not available in the default configuration. It can be loaded by adding pg_stat_statements to shared_preload_libraries in postgresql.conf.
- Whenever any SQL query is executed by a server, pg_stat_statements adds an entry for that query in the hash table where all the statistics about the query execution are stored.
- When user queries pg_stat_statements view, it fetches the stats from the hash table.

pg_stat_statements (Continued)

- Track Parameters Associated With Pg_stat_statements
  - pg_stat_statements.max : pg_stat_statements.max is the maximum number of statements tracked by the pg_stat_statements module (i.e., the maximum number of rows in the pg_stat_statements table)
  - pg_stat_statements.track : pg_stat_statements.track specifies the statements that can be tracked by pg_stat_statements module. It can be only top level statement or all the statements including the nested statements or none.
  - pg_stat_statements.track_utility : pg_stat_statements.track_utility controls whether utility commands (other than SELECT, INSERT, UPDATE, DELETE) are tracked by the module.
  - pg_stat_statements.save : pg_stat_statements.save specifies whether to save statement statistics across server shutdowns. If it is off then statistics are not saved at shutdown nor reloaded at server start. The default value is on.

pg_stat_statements (Continued)

```
postgres=# select * from pg_stat_statements where calls > 2;
-[ RECORD 1 ]-------+---------------------------
userid              | 10
dbid                | 12323
queryid             | -2116979465729057784
query               | insert into tab1 values($1)
calls               | 3
total_time          | 0.8577
min_time            | 0.0415
max_time            | 0.7474
mean_time           | 0.2859
stddev_time         | 0.32652004532647
rows                | 3
shared_blks_hit     | 2
shared_blks_read    | 1
shared_blks_dirtied | 2
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_dirtied  | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
blk_read_time       | 0
blk_write_time      | 0
```

pg_stat_statements (Continued)

- Query + No. Of Calls + Avg Time

queryid	6669079817886550995
query	select * from tab1 where a=$1
calls	6
total_time	1.30983

- Avg. Shared Buffer Hit Ratio

shared_blks_hit	6
shared_blks_read	1

hit_rate = shared_blks_hit / (shared_blks_hit + shared_blks_read) : 85%

pg_stat_statements (Continued)

- Time Spent Reading Or Writing To Disk (in Ms)

blk_read_time	10.549
blk_write_time	441.661

- Monitoring Query Performance Using Pg_stat_statements
  - SELECT substring(query, 1, 50) AS short_query, round(total_time::numeric, 2) AS total_time, calls, round(mean_time::numeric, 2) AS mean, round((100 * total_time / sum(total_time::numeric) OVER ())::numeric, 2) AS percentage_cpu, shared_blks_hit, shared_blks_read FROM pg_stat_statements ORDER BY total_time DESC LIMIT 10;

pg_stat_statements (Continued)

- Monitoring Query Performance Using Pg_stat_statements

short_query	total_time	calls	mean	percentage_cpu	shared_blks_hit	shared_blks_read
SELECT abalance FROM pgbench_accounts WHERE aid =	18300.01	1099212	0.02	85.83	4207679	195121
copy pgbench_accounts from stdin	1543.69	1	1543.69	7.24	6	16394
alter table pgbench_accounts add primary key (aid)	585.10	1	585.10	2.74	2175	14315
vacuum analyze pgbench_accounts	445.29	1	445.29	2.09	20564	28700
CREATE DATABASE test	291.67	1	291.67	1.37	41	18
create extension pg_stat_statements	36.86	1	36.86	0.17	643	103
create extension pg_stat_statements	25.24	1	25.24	0.12	643	99
vacuum analyze pgbench_branches	17.55	1	17.55	0.08	179	12
vacuum pgbench_branches	11.59	1	11.59	0.05	36	1
alter table pgbench_branches add primary key (bid)	7.08	1	7.08	0.03	114	19

(10 rows)

Agenda

- Database Logs
- The PostgreSQL Statistics Collector
- Statistics Views
- Wait Events
- Monitoring query performance using pg_stat_statements
- Useful debugging tools

Useful debugging Tools

- pageinspect
  - pageinspect is an extension module in postgres that provides functions to inspect the contents of database pages at low level which can be used for debugging.
  - It includes various user exposed functions that can be used to view the contents of heap and different index pages.
  - It is particularly useful in understanding the changes happening at page level when various actions are performed on a relation.

Useful debugging Tools (Continued)

- pageinspect
  - CREATE TABLE tab1(a int4 primary key);
  - SELECT txid_current();
  - INSERT INTO tab1 VALUES(10);
  - CREATE EXTENSION pageinspect;

    test=# SELECT lp, lp_len, t_xmin, t_xmax, lp_off FROM heap_page_items(get_raw_page('tab1', 0));

```
lp  | lp_len | t_xmin | t_xmax | lp_off
----+--------+--------+--------+--------
  1 |     28 |    599 |      0 |   8160
(1 row)
```

Useful debugging Tools (Continued)

- pageinspect
  - SELECT txid_current();
  - UPDATE tab1 SET a=20 WHERE a=10;
  - COMMIT;

  test=# SELECT lp, lp_len, t_xmin, t_xmax, lp_off FROM heap_page_items(get_raw_page('tab1', 0));

```
 lp | lp_len | t_xmin | t_xmax | lp_off
----+--------+--------+--------+--------
  1 |     28 |    599 |    601 |   8160
  2 |     28 |    601 |      0 |   8128
(2 rows)
```

Useful debugging Tools (Continued)

- pageinspect

test=# SELECT * FROM bt_page_items('tab1_pkey', 1);

```
 itemoffset | ctid  | itemlen | nulls | vars |          data
------------+-------+---------+-------+------+------------------------
          1 | (0,1) |      16 | f     | f    | 0a 00 00 00 00 00 00 00
          2 | (0,2) |      16 | f     | f    | 14 00 00 00 00 00 00 00
(2 rows)
```

- VACUUM tab1;

test=# SELECT lp, lp_len, t_xmin, t_xmax, lp_off FROM heap_page_items(get_raw_page('tab1', 0));

```
 lp | lp_len | t_xmin | t_xmax | lp_off
----+--------+--------+--------+--------
  1 |      0 |        |        |      0
  2 |     28 |    601 |      0 |   8160
(2 rows)
```

Useful debugging Tools (Continued)

- pageinspect

```
test=# SELECT * FROM bt_page_items('tab1_pkey', 1);

itemoffset  | ctid  | itemlen | nulls | vars |          data
------------+-------+---------+-------+------+------------------------
          1 | (0,2) |      16 | f     | f    | 14 00 00 00 00 00 00 00
(1 row)
```

- pgstattuple
  - pgstattuple is another extension module in postgres that provides table-level statistics.
  - This contrib module is particularly useful in identifying the tables which have bloated and how much bloat is there.
  - Like pageinspect, this module also provides a set of functions that can be used to identify the bloated tables in postgres.

Useful debugging Tools (Continued)

- pgstattuple
  - Now, Let us generate some bloat artificially and see how this module can be helpful in identifying it.

    pgbench -i -s10 test

Client-1:
test=# \dt+ pgbench_accounts

```
                        List of relations
 Schema |       Name       | Type  | Owner |  Size  | Description
--------+------------------+-------+-------+--------+-------------
 public | pgbench_accounts | table | ashu  | 128 MB |
(1 row)
```

test=# begin; set default_transaction_isolation TO 'repeatable read';

test=# SELECT * FROM pgbench_accounts LIMIT 1;

Useful debugging Tools (Continued)

- pgstattuple

Client-2:
pgbench --no-vacuum --client=2 --jobs=4 --transactions=100000 --protocol=prepared test

Client-1:
test=# \dt+ pgbench_accounts

```
                        List of relations
 Schema |       Name       | Type  | Owner |  Size   | Description
--------+------------------+-------+-------+---------+-------------
 public | pgbench_accounts | table | ashu  | 256 MB |
(1 row)
```

  - From above data, it seems that the test table “pgbench_accounts” has indeed doubled in size. Now, let's perform VACUUM ANALYZE on this table and verify if it has really bloated using pgstattuple module.

Useful debugging Tools (Continued)

- pgstattuple

Client-1:

- VACUUM ANALYZE pgbench_accounts;
- CREATE EXTENSION pgstattuple;

test=# select table_len, scanned_percent, approx_free_space, approx_free_percent from pgstattuple_approx('pgbench_accounts');

```
 table_len | scanned_percent | approx_free_space | approx_free_percent
-----------+-----------------+-------------------+---------------------
 268607488 |               0 |         131164800 |    48.8314011558754
(1 row)
```

- Likewise, there are several other functions available in pgstattuple module which can be used for finding various informations. For eg. there is a function named pgstattuple(), which returns a relation's physical length, percentage of "dead" tuples, and other infos.

Thanks!