Performance analysis at full power

Julien Rouhaud

pgconf.eu 2019

Oct. 16th 2019
Who am I

- Julien Rouhaud, from France
  - Working with PostgreSQL since 2008
  - DBA, consulting, developer
- Author of HypoPG and other tools
- Some contributions to PostgreSQL
Why this talk
My own experience

-Based on my experience as database administrator
  (subset of) Existing (or new) facilities I find most useful
  Open source
  For performance analysis!

- There are many other facilities availables and other approaches
  Sometime complementary (some info are only available in the logs, pgBadger is so useful)
Why this talk
PostgreSQL’s moving fast

- PostgreSQL changes
  - New features for better performance
  - New bottlenecks
  - New performance counters
- Lot of metrics available on the OS side
  - top, perf, iostat...
- PostgreSQL’s core statistics
  - some metrics available
  - Cumulated statistics
  - No underlying system metrics
  - but extensible, there are tools to help!
PostgreSQL statistics
How it works

- Some in core, some in contrib, some in external extensions
- Almost all of them **are cumulated counters over time**
- Usually store information in shared memory
- Accessible with views or Set Returning Functions
**PostgreSQL statistics**

**List of in-core views**

```sql
select viewname from pg_views where viewname ~ '^pg_stat_';
```

<table>
<thead>
<tr>
<th>viewname</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_stat_bgwriter</td>
</tr>
<tr>
<td>pg_stat_progress_vacuum</td>
</tr>
<tr>
<td>pg_stat_progress_cluster</td>
</tr>
<tr>
<td>pg_stat_progress_create_index</td>
</tr>
<tr>
<td>pg_stat_all_tables</td>
</tr>
<tr>
<td>pg_stat_xact_all_tables</td>
</tr>
<tr>
<td>pg_stat_sys_tables</td>
</tr>
<tr>
<td>pg_stat_xact_sys_tables</td>
</tr>
<tr>
<td>pg_stat_user_tables</td>
</tr>
<tr>
<td>pg_stat_xact_user_tables</td>
</tr>
<tr>
<td>pg_stat_all_indexes</td>
</tr>
<tr>
<td>pg_stat_sys_indexes</td>
</tr>
<tr>
<td>pg_stat_user_indexes</td>
</tr>
<tr>
<td>pg_stat_activity</td>
</tr>
<tr>
<td>pg_stat_replication</td>
</tr>
<tr>
<td>pg_stat_wal_receiver</td>
</tr>
<tr>
<td>pg_stat_subscription</td>
</tr>
<tr>
<td>pg_stat_ssl</td>
</tr>
<tr>
<td>pg_stat_gssapi</td>
</tr>
<tr>
<td>pg_stat_database</td>
</tr>
<tr>
<td>pg_stat_database_conflicts</td>
</tr>
<tr>
<td>pg_stat_user_functions</td>
</tr>
<tr>
<td>pg_stat_xact_user_functions</td>
</tr>
<tr>
<td>pg_stat_archiver</td>
</tr>
</tbody>
</table>

(24 rows)
PostgreSQL statistics
The limits

- No historisation done by PostgreSQL
- You know the cumulated counters since the last reset
- Are those counters always increasing the same way?
- What happened yesterday between 9AM and 2PM?
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkpoints_timed</td>
<td>1214</td>
</tr>
<tr>
<td>checkpoints_req</td>
<td>84</td>
</tr>
<tr>
<td>checkpoint_write_time</td>
<td>4534682</td>
</tr>
<tr>
<td>checkpoint_sync_time</td>
<td>34732</td>
</tr>
<tr>
<td>buffers_checkpoint</td>
<td>236104</td>
</tr>
<tr>
<td>buffers_clean</td>
<td>204069</td>
</tr>
<tr>
<td>maxwritten_clean</td>
<td>523</td>
</tr>
<tr>
<td>buffers_backend</td>
<td>594294</td>
</tr>
<tr>
<td>buffers_backend_fsync</td>
<td>0</td>
</tr>
<tr>
<td>buffers_alloc</td>
<td>5484743</td>
</tr>
<tr>
<td>stats_reset</td>
<td>2019-07-04 21:51:48.554982+02</td>
</tr>
</tbody>
</table>
Get all metrics every few minutes, and store it somewhere
You can do that manually with cron or custom script
Or use PoWA
- Extensible infrastructure to historize multiple data sources
- Optional background worker for a self contained solution
- Optional daemon for more complex setup
- Custom UI to visualize and analyze metrics
PostgreSQL statistics
Time visualisation

Checkpointing activity
- Buffers alloc
- Buffers checkpoint
- Sync time
- Write time

Background writer
- Maxwritten clean
- Buffers clean
Official contrib

Global view of what’s happening on your server

Query normalization, based on object identifiers

Cumulate many statistics per `queryid, userid, dbid`

- cumulated runtime and number of execution
- min, max, mean time
- shared/local buffers access (hit, read, dirtied, written)
- temps files
- IO timing (depending on `track_io_timing`)
What can we learn?

- Most frequent queries
- Slowest queries
- Queries generating most amount of temporary files
- Per-query hit-ratio
- Queries requiring more work_mem
- …
pg_stat_statements
Query example

```
SELECT round(total_time::numeric/calls, 2) AS avg_time, mean_time,
       rows/(calls) AS avg_rows,
       shared_blks_hit * 100 / (shared_blks_hit+shared_blks_read) AS hit_ratio,
FROM pg_stat_statements s JOIN pg_database d ON d.oid = s.dbid
WHERE datname = 'bench' AND (shared_blks_hit+shared_blks_read) > 0
ORDER BY total_time / calls DESC;
```

<table>
<thead>
<tr>
<th>avg_time</th>
<th>mean_time</th>
<th>avgrows</th>
<th>hitratio</th>
<th>query</th>
</tr>
</thead>
<tbody>
<tr>
<td>385.43</td>
<td>385.43</td>
<td>1</td>
<td>48</td>
<td>UPDATE pgbench_accounts SET abalance = abalance + 2796 WHERE aid = 1334587</td>
</tr>
<tr>
<td>212.77</td>
<td>212.77</td>
<td>1</td>
<td>48</td>
<td>SELECT abalance FROM pgbench_accounts WHERE aid = $1</td>
</tr>
<tr>
<td>0.38</td>
<td>0.38</td>
<td>1</td>
<td>67</td>
<td>UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2</td>
</tr>
<tr>
<td>0.05</td>
<td>0.05</td>
<td>1</td>
<td>75</td>
<td>UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2</td>
</tr>
</tbody>
</table>

(4 rows)
pg_stat_statements
Over time

- **Query runtime per second (kind of SQL load)**

- **global, per-database or per-query**
And general consumption over a specific interval

<table>
<thead>
<tr>
<th>Database</th>
<th>#Calls</th>
<th>Runtime</th>
<th>Avg runtime</th>
<th>Blocks read</th>
<th>Blocks hit</th>
<th>Blocks dirtied</th>
<th>Blocks written</th>
<th>Temp Blocks written</th>
<th>I/O time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bench</td>
<td>43,167.00</td>
<td>14 s 574 ms</td>
<td>340 μs</td>
<td>109.61 M</td>
<td>912.08 M</td>
<td>66.73 M</td>
<td>60.00 K</td>
<td>0 B</td>
<td>13 s 283 ms</td>
</tr>
<tr>
<td>powa</td>
<td>1,571.00</td>
<td>6 s 406 ms</td>
<td>4 ms 80 μs</td>
<td>40.00 K</td>
<td>11.84 M</td>
<td>8.00 K</td>
<td>0 B</td>
<td>0 B</td>
<td>1 ms 280 μs</td>
</tr>
<tr>
<td>tpc</td>
<td>379.00</td>
<td>5 min 51 s</td>
<td>928 ms 450 μs</td>
<td>3.99 G</td>
<td>88.00 G</td>
<td>694.34 M</td>
<td>521.17 M</td>
<td>661.95 M</td>
<td>37 s 116 ms</td>
</tr>
<tr>
<td>postgres</td>
<td>40.00</td>
<td>580 ms 926 μs</td>
<td>14 ms 520 μs</td>
<td>56.00 K</td>
<td>5.91 M</td>
<td>176.00 K</td>
<td>0 B</td>
<td>0 B</td>
<td>2 ms 310 μs</td>
</tr>
<tr>
<td>obvious</td>
<td>25.00</td>
<td>1 s 264 ms</td>
<td>50 ms 560 μs</td>
<td>397.77 M</td>
<td>371.77 M</td>
<td>0 B</td>
<td>0 B</td>
<td>0 B</td>
<td>920 ms 150 μs</td>
</tr>
</tbody>
</table>

Drill-down approach to investigate performance issues
## pg_stat_statements

### Identify slow queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Execution</th>
<th>I/O Time</th>
<th>Blocks</th>
<th>Temp blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT pg_sleep(s1)</td>
<td>88.00</td>
<td>9 min 48 s</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT count(*) FROM commandes cmd JOIN lignes_commandes lc ON lc.num...</td>
<td>16.00</td>
<td>2 min 19 s</td>
<td>0.25 G 479.01 M</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM pieces_fournisseurs WHERE cout_piece &gt;= s1</td>
<td>10.00</td>
<td>49 s 642 ms</td>
<td>2 min 27 s</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT numero_commande, etat_commande FROM commandes WHERE client_id =...</td>
<td>16.00</td>
<td>80 s 960 ms</td>
<td>1 min 37 s</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM clients cl JOIN contacts co ON co.client_id = cl.contac...</td>
<td>16.00</td>
<td>29 s 461 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT co.nom FROM clients cl JOIN contacts co ON co.client_id = cl.contac...</td>
<td>16.00</td>
<td>17 s 796 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM pays p JOIN contacts con ON con.code_pays = p.code...</td>
<td>16.00</td>
<td>10 s 702 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>ALTER TABLE ONLY public.lignes_commandes ADD CONSTRAINT lignes_command...</td>
<td>1.00</td>
<td>4 s 587 ms</td>
<td>16 ms 971 μs</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>COPY public.lignes_commandes (numero_commande, piece_id, fournisseur_id,...</td>
<td>1.00</td>
<td>4 s 528 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM commandes WHERE date_commande BETWEEN $1</td>
<td></td>
<td>$2...</td>
<td>16.00</td>
<td>2 s 181 ms</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM pays p JOIN contacts ON con.code_pays = p.code...</td>
<td>16.00</td>
<td>2 s 144 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT con.nom</td>
<td></td>
<td>$1</td>
<td></td>
<td>code_pays</td>
</tr>
<tr>
<td>SELECT nom FROM contacts cl JOIN pays p ON p.code_pays = c.code_pays whl...</td>
<td>16.00</td>
<td>1 s 235 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>COPY public.pieces (piece_id, nom, fabricant, marque, type_piece, taille...</td>
<td>1.00</td>
<td>1 s 225 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>COPY public.pieces_fournisseurs (piece_id, fournisseur_id, quantite_disponible...</td>
<td>1.00</td>
<td>1 s 34 ms</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM commandes WHERE date_commande BETWEEN $1</td>
<td></td>
<td>$2...</td>
<td>16.00</td>
<td>903 ms 89 μs</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM pieces_fournisseurs WHERE quantite_disponible &lt; $...</td>
<td>10.00</td>
<td>690 ms 725 μs</td>
<td>0</td>
<td>0 B 0 B</td>
</tr>
</tbody>
</table>

Julien Rouhaud

Performance analysis at full power
**pg_stat_kcache**

Kernel metrics

- [github.com/powa-team/pg_stat_kcache](https://github.com/powa-team/pg_stat_kcache)
- Wrapper around `get_rusage(2)`
- Gives access to kernel metrics, aggregated per `(queryid, dbid, userid)`:  
  - Physical disk reads and writes  
  - User and system CPU  
  - Context switches, page faults
What can we learn?

- "Real" hit-ratio: shared_buffers vs OS cache vs Disk access
- CPU intensive queries
- Too high number of active queries
pg_stat_kcache
Examples - per database

Blocks (On database tpc)

System resources (events per sec)
pg_stat_kcache
Examples - per query

CPU Time repartition

- CPU other time / Query time
- CPU system time / Query time
- CPU user time / Query time

Performance analysis at full power
pg_wait_sampling
Wait events monitoring

- [github.com/postgrespro/pg_wait_sampling/](github.com/postgrespro/pg_wait_sampling/)
- Developed by Postgres Professional
- Efficient high frequency sampling of *wait events*
- Default period is 10ms, customisable
- Aggregated per queryid, dbid
- For 9.6+ only, when Wait Events were introduced
What can we learn?

- Low level bottlenecks that can’t be seen at SQL level
  - Costly parts of a query execution
  - Lightweight locks contention (Buffer mapping, WAL write lock...)
  - IPC, IO and other events
Per database:

Wait Events (per second)

- IO
- Timeout
- IPC
- Extension
- Client
- Activity
- Buffer pin
- Lock
- Lightweight Lock
**pg_wait_sampling**

**Examples**

Per query:

```sql
SELECT count(*) FROM commandes cmd
JOIN lignes_commandes lc ON lc.numero_commande = cmd.numero_commande
WHERE cmd.client_id = $1
```

- **# of execution**: 20
- **Total runtime**: 10 min 5 s 396 ms
- **Hit ratio**: 22.49%

**Wait Events**

**Wait events summary**

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event</th>
<th># of events</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO</td>
<td>DataFileRead</td>
<td>52,872.00</td>
</tr>
<tr>
<td>IPC</td>
<td>BgWorkerShutdown</td>
<td>1,504.00</td>
</tr>
<tr>
<td>IPC</td>
<td>ExecuteGather</td>
<td>48.00</td>
</tr>
<tr>
<td>IPC</td>
<td>ParallelFinish</td>
<td>14.00</td>
</tr>
<tr>
<td>IPC</td>
<td>Hash/Build/Hashing/Inner</td>
<td>8.00</td>
</tr>
</tbody>
</table>

Julien Rouhaud - Performance analysis at full power
**pg_qualstats**
Statistics on predicates

- github.com/powa-team/pg_qualstats
- Gather statistics on predicates (WHERE / JOIN clauses)
  - Number of underlying query executions
  - Number of predicate’s operator execution
  - Selectivity
  - Sequential scan or index scan
- Per queryid, userid, dbid
- Sampled to avoid overhead (default is 1 / max_connections)
Detect missing indexes
Differentiate most executed, most/least filtering, most frequent constants
Detect possible partial indexes
If sampled over time, avoid suggesting indexes for night batches
pg_qualstats
Constant distribution

Most executed values

- 'returned':::text
- 'shipped':::text
- Others

Julien Rouhaud
Performance analysis at full power
Index suggestion

- Possible indexes for attributes present in `WHERE pieces_fournisseurs.quantite_disponible < ? AND pieces_fournisseurs.cout_piece >= ?`:
  - With access method `btree`
    - **Attribute**
      - `pieces_fournisseurs.cout_piece`
    - **Data distribution**
      - approximately **1000** distinct values
    - **Attribute**
      - `pieces_fournisseurs.quantite_disponible`
    - **Data distribution**
      - approximately **9985** distinct values
HypoPG
Hypothetical indexes

- github.com/HypoPG/hypopg
- Hypothetical indexes, aka. "What if this index existed?"
- Create "fake" indexes instantly, without any resource consumption
- EXPLAIN can use such index
SELECT id, dt FROM command WHERE state = ?

# of execution: 20
Total runtime: 16 s 571 ms
Hit ratio: 100.0%

Query detail    PG Cache   IO    System resources  Wait Events    Predicates

Predicates used by this query

Predicate
WHERE command.state = ?
Avg filter_ratio (excluding index)
99.90%
Execution count (excluding index)
258,500,000.00

Index suggestion
- Possible indexes for attributes present in WHERE command.state = ?:
  - With access method btree
    - Attribute
      - command.state
  - Data distribution
    - approximately 2 distinct values
  - With access method btree

The following indexes would be used:
CREATE INDEX ON "public"."command"(state)

EXPLAIN plan without suggested indexes:
Seq Scan on command (cost=0.00..1986.00 rows=110 width=12)
  Filter: (state = 'returned'::text)
Query cost gain factor with hypothetical index: 99.41%

EXPLAIN plan with suggested index
Index Scan using <27940>btree_command_state on command
(cost=0.04..11.67 rows=110 width=12)
  Index Cond: (state = 'returned'::text)
Get all executed queries on the given time interval
Get all interesting predicates (seq scan, filtering at least 30%)
Get information about indexing capabilities (operators, datatype, opclass)
Analyze and suggest indexes to optimize all queries with the least amount of indexes
Check with HypoPG that indexes would be used
pg_qualstats + HypoPG

Global index suggestion

<table>
<thead>
<tr>
<th>Index</th>
<th>Used by</th>
<th># Queries boosted</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE INDEX ON public.commandes USING btree(client_id, date_commande)</td>
<td>WHERE commandes.client_id = ? AND commandes.date_commande &gt;= ? AND</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>commandes.date_commande &lt;= ?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE commandes.client_id = ?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE commandes.date_commande &lt;= ? AND commandes.date_commande &gt;= ?</td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX ON public.pieces_fournisseurs USING btree(cout_piece, quantite_disponible)</td>
<td>WHERE pieces_fournisseurs.quantite_disponible &lt; ? AND</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>pieces_fournisseurs.cout_piece &gt;= ?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE pieces_fournisseurs.cout_piece &gt;= ?</td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX ON public.clients USING btree(solde, client_id)</td>
<td>WHERE clients.client_id = ? AND clients.solde &gt; ?</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>WHERE clients.solde &gt; ?</td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX ON public.commandes USING btree(date_commande)</td>
<td>WHERE commandes.date_commande &lt;= ? AND commandes.date_commande &gt;= ?</td>
<td>2</td>
</tr>
</tbody>
</table>

Hypothetical index creation error

<table>
<thead>
<tr>
<th>Query</th>
<th>Index used</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT COUNT(*) FROM pieces_fournisseurs WHERE quantite_disponible &lt; 2117::integer AND cout_piece &gt;= 976::numeric</td>
<td>✔</td>
<td>45.7%</td>
</tr>
<tr>
<td>SELECT co.nom FROM clients cl JOIN contacts co ON co.contact_id = cl.contact_id WHERE cl.solde &gt; 448::numeric</td>
<td>✔</td>
<td>17.11%</td>
</tr>
<tr>
<td>SELECT count(*) FROM commandes cmd JOIN lignes_commandes lc ON lc.numero_commande = cmd.numero_commande WHERE cmd.client_id = 4180::integer</td>
<td>✔</td>
<td>16.91%</td>
</tr>
<tr>
<td>SELECT numero_commande, etat_commande FROM commandes WHERE client_id = 4180::integer</td>
<td>✔</td>
<td>99.74%</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM pieces_fournisseurs WHERE cout_piece &gt;= 977::numeric</td>
<td>✔</td>
<td>35.63%</td>
</tr>
<tr>
<td>SELECT COUNT(*) FROM commandes WHERE client_id = 13598::integer AND priorite_commande LIKE '3-3%'::text</td>
<td>✔</td>
<td>99.75%</td>
</tr>
</tbody>
</table>

Julien Rouhaud

Performance analysis at full power
pg_track_settings
History of configuration changes

- [ ] github.com/rjuju/pg_track_settings/
- SQL only extension
- detect and store the settings changed since last call
- both global and object specific (eg. ALTER DATABASE SET)
- and also postgres restart
What changed since yesterday?

```sql
# SELECT * FROM pg_track_settings_diff(now() - interval '1 day', now());
```

<table>
<thead>
<tr>
<th>name</th>
<th>from_setting</th>
<th>from_exists</th>
<th>to_setting</th>
<th>to_exists</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkpoint_segments</td>
<td>30</td>
<td>t</td>
<td>35</td>
<td>t</td>
</tr>
</tbody>
</table>

(1 row)
What’s the full history for a specific setting?

```sql
# SELECT * FROM pg_track_settings_log('checkpoint_segments');

<table>
<thead>
<tr>
<th>ts</th>
<th>name</th>
<th>setting_exists</th>
<th>setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>2015-01-25 01:01:42.58+01</td>
<td>checkpoint_segments</td>
<td>t</td>
<td>35</td>
</tr>
<tr>
<td>2015-01-25 01:00:37.44+01</td>
<td>checkpoint_segments</td>
<td>t</td>
<td>30</td>
</tr>
</tbody>
</table>
```

(2 rows)
What was the configuration like at a specific timestamp?

```sql
# SELECT * FROM pg_track_settings('2015-01-25 01:01:00');

<table>
<thead>
<tr>
<th>name</th>
<th>setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkpoint_completion_target</td>
<td>0.9</td>
</tr>
<tr>
<td>checkpoint_segments</td>
<td>30</td>
</tr>
<tr>
<td>checkpoint_timeout</td>
<td>300</td>
</tr>
</tbody>
</table>
```

Julien Rouhaud
Performance analysis at full power
Available in PoWA, filtered by database if applicable
Demo

dev-powa.anayrat.info

(not credential required, just click connect)
A lot of tools are there to help
Can be used alone or together
Or even integrated in your own solution
Questions?

- rjuju.github.io
- @rjuju123
- powateam (pg12 compatible)