Abstract

- PostgreSQL adoption is exploding and the move to the cloud is fueling it
- The difference between kicking things off and scaling in production
- The four areas of focus for scaling PostgreSQL
  - Query & SQL Optimization
  - Performance Features
  - Architectural Improvements
  - Parameter Tuning
2022 DBMS Market Snapshot

Top 3 Cloud Revenue ($B)

- AWS: $23
- Microsoft: $9
- Google: $8

80% of Cloud Market

DBMS Subsegments ($B)

- RDBMS: $71
- Non-RDBMS: $19
- Pre-RDBMS: $1

Top 10 (Top 5 = 81% of Market) Fastest Growing

<table>
<thead>
<tr>
<th>Rank</th>
<th>Company</th>
<th>Revenue ($)</th>
<th>Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AWS</td>
<td>23,023</td>
<td>83.2%</td>
</tr>
<tr>
<td>2</td>
<td>Snowflake</td>
<td>21,970</td>
<td>81.5%</td>
</tr>
<tr>
<td>3</td>
<td>Cockroach</td>
<td>18,875</td>
<td>76.3%</td>
</tr>
<tr>
<td>4</td>
<td>Databricks</td>
<td>7,618</td>
<td>58.3%</td>
</tr>
<tr>
<td>5</td>
<td>Enterprise</td>
<td>4,587</td>
<td>49.3%</td>
</tr>
<tr>
<td>6</td>
<td>SAP</td>
<td>3,611</td>
<td>48.7%</td>
</tr>
<tr>
<td>7</td>
<td>Alibaba</td>
<td>1,939</td>
<td>46.2%</td>
</tr>
<tr>
<td>8</td>
<td>Google</td>
<td>1,251</td>
<td>41.3%</td>
</tr>
<tr>
<td>9</td>
<td>Tencent</td>
<td>1,223</td>
<td>40.7%</td>
</tr>
<tr>
<td>10</td>
<td>Redis</td>
<td>1,205</td>
<td>40.8%</td>
</tr>
<tr>
<td></td>
<td>MongoDB</td>
<td>1,205</td>
<td></td>
</tr>
</tbody>
</table>

$91B 14.4% Growth

$11B Gain

$50.3B 55% of Market

98% of 2022 Growth
So - what do you do when you need to scale your database in the cloud?
Scale by Credit Card!
Well, not really ... You are only delaying the inevitable
You tested your application here ...
... and this is what production looks like
There is no magic button or setting ...
Scaling PostgreSQL
A Developer’s Guide

- Query & SQL Optimization
- Performance Features
- Architectural Improvements
- Parameter Tuning
Query & SQL Optimization
pg_stat_statements is your friend

- PostgreSQL extension, included in distribution and off by default
- Logs statistics about SQL statements
- Easy stats to watch out for
  - Long running (mean_exec_time)
  - Most frequent (calls)
  - Standard deviation in execution time (stddev_exec_time)
  - I/O intensive (blk_read_time, blk_write_time)
Explain plan is your friend

Node: What is happening in this step? Feed result to parent Node.

Relation: What is it happening on? Table or result of child Node?

Cost: Relatively how expensive is this step?

Modifier: Tweak result before handoff.

Rows: How many rows will be returned by this Node.

Loops: How many times will this step be executed.
Watch out for locks!

Session 1

BEGIN;

UPDATE foo SET ... WHERE id = 1;
UPDATE foo SET ... WHERE id = 2;
UPDATE foo SET ... WHERE id = 3;
COMMIT;

Session 2

UPDATE foo SET ... WHERE id = 1;
(waits)
Performance Features
Indexes

- B-Tree - default index
- Hash - equality checks
- Composite - multi column
- Partial - conditional index on subset of data
- Covering - includes an additional column
- BRIN (block range index) - space efficient for sorted tables
Indexes - Not a one-size-fits-all!

- You need all or most of the data any ways
- Your workload is WRITE or UPDATE heavy with little READs
- ‘Over’ indexing can cause data bloat
- Your table is too small
Many performance features ‘just work’

A few examples …

- Parallel queries
- Heap-Only Tuples (HOT)
- Incremental sort
- Autovacuum
Architectural Improvements
Load Balancing
Load Balancing

Single Node SELECTs

transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 25
number of threads: 1
maximum number of tries: 1
duration: 60 s
number of transactions actually processed: 19139
number of failed transactions: 0 (0.000%)
latency average = 67.215 ms
initial connection time = 8620.897 ms
\( \text{tps} = 371.939402 \) (without initial connection time)

Load Balanced 3-node Cluster

transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 25
number of threads: 1
maximum number of tries: 1
duration: 60 s
number of transactions actually processed: 24885
number of failed transactions: 0 (0.000%)
latency average = 51.449 ms
initial connection time = 8896.110 ms
\( \text{tps} = 485.918972 \) (without initial connection time)
Partitioning

Application

Jan  Feb  Mar
Apr  May  Jun
Jul  Aug  Sep
Oct  Nov  Dec

Application

Q1

Q2

Q3

Q4
Partitioning

select * from foo where month = 'Aug'

select * from foo where month = 'Aug'
Parameter Tuning
Easily tuned database parameters

- **Most defaults are good enough!**
  - **shared_buffers**
    - Cache for frequently accessed data
    - Default is 128MB
    - Recommended is between 25% and 40% of system memory
  - **wal_buffers**
    - Shared memory not yet written to disk
    - Default is 3% of shared_buffers
    - A value of up to 16MB can improve performance in high concurrency commits
  - **work_mem**
    - Memory available for a query operation
    - Default is 4MB
    - High I/O activity for a query is an indicator that an increase in work_mem can help
    - Each parallel operation is allowed to use memory up to this value
Easily tuned database parameters

- **maintenance_work_mem**
  - Memory used by maintenance operations like VACUUM and ANALYZE
  - Default is 64MB
  - Higher values can improve maintenance performance
  - Each worker is allowed to use up to this value

- **effective_cache_size**
  - Value of effective disk cache to be used by query planner
  - Not an allocation!
  - Default is 4GB
  - Higher values encourage index scans

- **random_page_cost**
  - Value of non-sequential disk page access cost
  - Not an allocation!
  - Default is 4.0
  - Lower values encourage index scans
Conclusion

Database performance involves a lot of variables. Optimize how data is accessed before scaling by credit card!
Questions?

LinkedIn: pg_umair

KEEP CALM AND USE POSTGRES