PostgreSQL Maintenance

Situations

There are multiple situations that can occur in a database. Maintenance is necessary in order to prevent them.

- Fragmented / bloated tables & indexes
- TransactionID (XID) Wrap-Around
- Out of Date query plans
- Unbalanced Indexes
- Index Utilization

Fragmentation / Bloat

- A side-effect of the PostgreSQL MVCC system is that ‘dead’ space will be left in the table and indexes after UPDATE and DELETE statements
- If maintained properly (and the workload permits) this can be reused by other updates
- Takes up disk space, unnecessarily
- Bloat can cause performance problems
  - Can be felt in INSERT / UPDATE / DELETE and SELECT

Finding Bloat

- Estimates are usually good enough to determine the troublesome tables
  - Table Bloat Query (https://wiki.postgresql.org/wiki/Show_database_bloat)
  - pgstattuple_approx

- More exact methods are available, but rarely does the extra overhead justify to precision
  - pgstattuple

Finding Bloat (cont.)

```
CREATE EXTENSION pgstattuple;
SELECT * FROM pgstattuple_approx('pgbench_accounts');
```

```
-[ RECORD 1 ]--------+-----------------
table_len            | 135340032
scanned_percent      | 45
approx_tuple_count   | 999981
approx_tuple_len     | 127414265
approx_tuple_percent | 94.1438117880746
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 2693908
approx_free_percent  | 1.99047389023818
```

What is xid Wraparound?

- PostgreSQL's MVCC transaction semantics compares transaction ID (XID) numbers
- A row version with an insertion XID greater than the current transaction's XID is "in the future" and is not be visible to the current transaction
- XIDs are limited in size (32 bits) so an instance that runs for a long time (more than 2 billion transactions) would suffer transaction ID wraparound
- The XID counter wraps around to the starting point (integer 4 < 9.6, freeze bit in 9.6+), so transactions that were in the past appear to be in the future

Finding xid Wraparound

- Find troublesome tables early so they can be handled during planned maintenance periods instead of PostgreSQL automatically doing it.

```
SELECT relname, age(relfrozenxid) as xid_age
  FROM pg_class c, pg_namespace n
 WHERE c.relnamespace = n.oid
   AND n.nspname = 'public'
   AND relkind = 'r';
```

Data Statistics

- PostgreSQL uses a cost-based optimizer to generate query plans in order to execute on them.

- Each operation in a query plan is given a cost, this is provided by table and index statistics. Each method of executing the query (hash join vs. merge join, index scan vs. sequential scan, etc…) are compared and the lowest-cost option is selected.

Age of Statistics

- Depending on the turnover of the data in a table, the statistics may get stale very quickly
- The meaning of age is very dependent on the application

```
SELECT relname, last_analyze,
       last_autoanalyze
  FROM pg_stat_user_tables;
```

Updating table statistics

- These can be updated manually by running the `ANALYZE` command.

```
Command:     ANALYZE
Description: collect statistics about a database
Syntax:
ANALYZE [ VERBOSE ] [ table [ ( column [, ...] ) ] ]
```

Vacuum

- Vacuum marks ‘dead’ space left by MVCC in blocks (tables and indexes) as available for re-use
- UPDATEs can take advantage of the available space
- Vacuum itself does not reclaim disk space
  - That is done with either vacuum full (offline), cluster (offline) or ‘pg_repack’ (online)
- Vacuum scans a whole table

Vacuum Modes / Options

- Default, vacuums the table and all associated index, looks for free space and marks it in the FSM:

```
VACUUM [table];
```

- Same as above, but, performs a statistical analysis as well:

```
VACUUM ANALYZE [table];
```

- Same as above, but, initiates freeze operation. Resets xmin to ‘2’:

```
VACUUM ANALYZE FREEZE [table];
```

Vacuum Modes / Options

Same as above, but, prints vacuum stats:

```
VACUUM ANALYZE VERBOSE [table];
```

WARNING: Fully locks the table and performs space reclamation / compaction:

```
VACUUM FULL ANALYZE VERBOSE [table]
```

pg_repack

- Allows for an online rebuild of the table
- Requires server side extension and client side utility

```
tpch=> CREATE EXTENSION pg_repack;
CREATE EXTENSION
```

- The -k (no superuser check) option is required for RDS

```
[~]$ pg_repack -k -t customer -h pg11xxx.rds.amazonaws.com \
               -U postgres -d tpch
INFO: repacking table "public.customer"
```

Autovacuum

- Vacuum can be run manually, but it is recommended to run the autovacuum daemon (default) to handle vacuuming automatically

- Autovacuum has a launcher process (utility process) that manages autovacuum workers

- Autovacuum workflow
  - Wake-up
  - Look for a table that has hit certain thresholds
  - Vacuum that table
  - Sleep

Autovacuum

Capable of all modes (except FULL)

Configurable number of background workers:

`autovacuum_max_workers`

Autovacuum can be throttled:

`autovacuum_vacuum_cost_delay`

Control when a FREEZE takes places:

`autovacuum_freeze_max_age`

Autovacuum Starvation

- Autovacuum works based on thresholds
  - Number of changes to a table
  - pg_stat_user_tables
    - n_tup_upd, n_tup_del
- If a table is receiving ultra-high volume changes, it would effectively hit the top of the ‘hit-list’ for autovacuum each time it runs
- This effectively drops your ‘max_workers’ by one

Dealing with starvation

- Modify global thresholds
  - postgresql.conf
  - Typically not preferred, especially if only 1 or 2 tables are causing issues
- Modify thresholds, per-table
  - ALTER TABLE…
- Increase max_workers
  - By the number of nasty tables you have
- Disable autovacuum, run an outside job for vacuum

```
psql -d gnb -c “VACUUM VERBOSE pgbench_branches”
```

Per Table Thresholds

- The default values controlling autovacuum are adequate for most tables

- Some need more aggressive settings

```
ALTER TABLE foo SET (autovacuum_vacuum_scale_factor = 0.05);
```

Routine Reindexing

- In some situations it is worthwhile to rebuild indexes periodically with the REINDEX command.

- Index pages that have become completely empty are reclaimed for reuse, but the potential for bloat is not indefinite.

- REINDEX will lock the table

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

Minimize Reindex Locking

```
cat index_build.sql
-------------------------------------------------
CREATE INDEX CONCURRENTLY people_lName_idx_new
ON people (id, lname);

BEGIN;
DROP INDEX people_lName_idx;
ALTER INDEX people_lName_idx_new
RENAME TO people_lName_idx;
COMMIT;
-------------------------------------------------

psql -d postgres -U postgres -f index_build.sql
```

Unused Indexes

- Indexes add overhead for every INSERT and UPDATE

- If the index does not enforce a constraint and is not used, it should be removed.

```
SELECT relname,indexrelid, idx_scan
   FROM pg_stat_user_indexes
  WHERE idx_scan = 0;
```

Summary

- Bloat
- XID Wraparound
- Gathering Statistics
- Vacuum
- Indexes