Authors: Egor Rogov, Pavel Luzanov

Feedback
Send feedback, comments and suggestions to:
edu@postgrespro.ru

Topics

Vacuuming and its tasks

Indexes monitoring and rebuilding

There are many more maintenance tasks, which are considered in other lessons (for example, setting up of message log or using backups).

Vacuuming

Vacuum options

Updating statistics

Updating visibility map and free space map

Transaction ID wraparound and freeze

Cleaning (vacuum) can be executed in different ways and, besides the actual cleaning of pages from dead row versions, it also performs a number of other important tasks.

Vacuum

VACUUM

`VACUUM` *`table`*	single table (and its indexes)
`VACUUM`	all tables in a database
`$ vacuumdb`	wrapper for use in the OS

runs concurrently with other transactions

frequent vacuuming loads I/O subsystem

rare vacuuming leads to high bloat

Vacuuming is needed to get rid of old (dead) tuples which are created when you change or delete rows, as required by the multiversion concurrency control (the «Architecture» module). Vacuum cleans up data pages of such tuples.

Vacuum works concurrently with other transactions, processing files page by page. Because of this it does not reduce the size of the files (except for the case when several pages at the end of the file are completely free). Instead, space is freed up inside the data pages, which can later be used to insert new tuples.

Vacuuming can be started manually with the VACUUM command, but how often shall we do it? Running it too often will create unnecessary load on the system. But running vacuum too rare, with a large volume of changes, the files may grow significantly in size (became *bloated*).

https://postgrespro.com/docs/postgresql/11/sql-vacuum

Vacuum

Autovacuum background process

- autovacuum launcher
  autovacuum worker
- frequency is adjusted automatically to the frequency of changes
- configured by the server parameters

Therefore, the main vacuuming method is a special autovacuum background process. It dynamically reacts to the rate of changes: the more active the changes, the more often the table will be processed.

There is an `autovacuum launcher` daemon in the system, which schedules the vacuuming and starts up the required number of `autovacuum worker` processes running in parallel.

Properly configured autovacuum process should manage to process the tables in time, avoiding unnecessary bloat.

Vacuum

VACUUM FULL

`VACUUM FULL` *`table`*	single table (and its indexes)
`VACUUM FULL`	all tables in a database
`$ vacuumdb --full`	wrapper for use in the OS

completely rewrites the table and all its indexes,
reducing file size and minimizing occupied space

acquires an exclusive lock on the table for the duration of operation

`TRUNCATE` works similarly

With a well-tuned vacuum process, the data files grow by some amount due to updates between sequential vacuuming. If the files have grown significantly (for example, due to massive unplanned changes that autovacuum did not manage to process in time), this can lead not only to overrun disk space, but also to slow down the system. Then, to shrink the files, a so-called full vacuum is required.

The VACUUM FULL command completely rewrites the contents of tables and indexes, minimizing the space occupied. However, this process acquires an exclusive table lock for the duration of the operation, and therefore cannot be executed concurrently with other transactions.

By the way, the SQL TRUNCATE command works in a similar way. It immediately deletes all rows without requiring vacuum after it, unlike DELETE. But, although TRUNCATE is a transactional command, it acquires an exclusive lock on the table. Thus, TRUNCATE will be executed only after all transactions that use the table (even for reading) are completed, and until the truncating transaction is completed, all access to the table will be blocked (even for reading).

If prolonged exclusive lock is undesirable, you can consider a third-party `pg_repack` extension https://github.com/reorg/pg_repack, which allows you to rebuild the table and its indexes on the fly. It still acquires an exclusive lock, but for a very short period of time.

Statistics update

Autovacuum background process

automatically updates statistics (analyze) after significant changes

random sampling is used, size is customizable

ANALYZE

```
ANALYZE [table]
$ vacuumdb --analyze-only
```

manually updating statistics

VACUUM

```
VACUUM ANALYZE [table]
$ vacuumdb --analyze
```

vacuuming and updating statistics at once

Fresh statistics is vital for the optimizer to generate adequate plans.

The autovacuum background process, in addition to cleaning up dead tuples, collects and updates statistics on changing data.

If this is not enough, the statistics can be updated manually using an ANALYZE command or together with vacuum by VACUUM ANALYZE.

When collecting statistics, a random sample of data of a certain size is read. This allows PostgreSQL to quickly collect statistics even on very large tables. The result is not 100% accurate, but in most cases there is nothing to worry about. If necessary, the sample size can be increased.

https://postgrespro.com/docs/postgresql/11/sql-analyze

Maps update

Visibility map

- bitmap of pages with all-visible tuples in all snapshots (tracks sufficiently-long-unchanged pages)
- visibility is reset after any change on a page, and set automatically by vacuum
- used to optimize vacuum (no need to visit unchanged pages) and index-only scans

Free space map

- tree-like structure to track free space in the pages
- updated when inserting new tuples and by vacuum
- used to quickly look up a suitable page when inserting new tuples

Visibility map and free space map were discussed in the «Data Organization» module.

Recall that the visibility map (VM) tracks pages that contain only those row versions (tuples) that are known to be visible to all transactions. In other words, there should not be two versions of the same row in the page. These are the pages that hasn't been changed for a long time for vacuum to clean up all dead tuples from them.

VM is used to optimize vacuum operation: vacuum does not need to visit tracked pages, since there is nothing to clean in them. Also VM speeds up index-only scans.

When any data changes in the page, the visibility bit of this page is reset. The bit is set automatically when possible during vacuum.

https://postgrespro.com/docs/postgresql/11/storage-vm

A free space map tracks pages that are not completely filled with data.

This a tree-like structure that allows to quickly look up a suitable page when inserting (and updating, because updating is interpreted as deleting and inserting) new rows.

The free space map is updated when inserting new tuples (free space decreases) and by the vacuum removing dead tuples from the pages (free space increases).

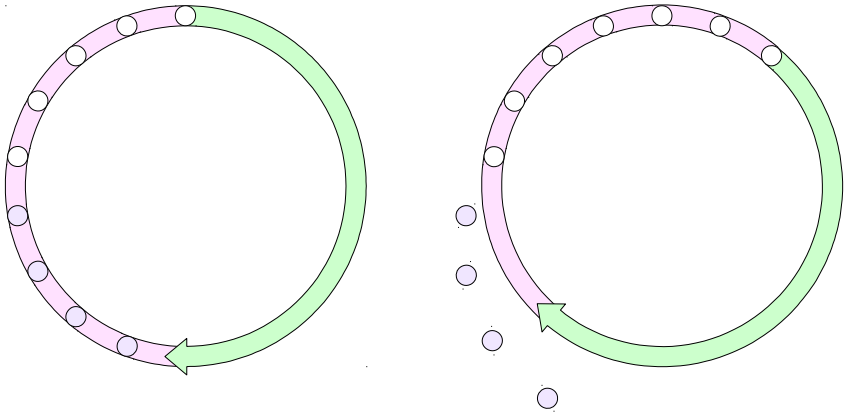
https://postgrespro.com/docs/postgresql/11/storage-fsm

Wraparound and freeze

32-bit transaction number

space of numbers is looped

old row versions are «frozen» by autovacuum



9

As was discussed in the «Architecture» module, PostgreSQL uses transaction ids to order operations. Transactoin counter is a 32-bits integer, and in a heavily loaded system it can overflow. But if the counter is reset to zero (which is called a wraparound), then the ordering of operations will be broken.

To prevent such a situation, the space of numbers is looped. For any given transaction, it is considered that one half of the numbers are behind (in the past), and the other half are ahead (in the future).

Over time we can come across a very old transaction id in `xmin` field of a tuple, and we would see the tuple in the future instead of the past. Therefore, tuples that are already visible in each snapshot are periodically marked as «frozen». The multiversion visibility rules considers such tuples to be infinitely old, and no longer looks at the transaction ids.

Freezing is automatically performed by the autovacuum process. Previously, PostgreSQL had to periodically completely scan tables in search of not-yet-frozen tuples. In PostgreSQL 9.6 this process was optimized: a *freeze map* is added to the visibility map — an extra bit that tracks those pages that contain only frozen tuples.

If the server stumbles over a non-frozen transaction in the «future» half of the transaction numbers, it forcibly stops; all running transactions fails. After that, the administrator must manually start the server and manually perform vacuum to be able to continue operation.

https://postgrespro.com/docs/postgresql/11/routine-vacuuming.html#vacuum-for-wraparound

Indexes

Index monitoring

Rebuilding indexes

The status of the indexes should be monitored and - when appropriate - rebuild indexes or delete unnecessary ones.

Index monitoring

Excessive indexes

- not used indexes (monitoring pg_stat_all_indexes.idx_scan), duplicating or intersecting indexes (queries to pg_index)
- problems: change overhead, disk space
- solution: removing unnecessary indexes (carefully)

Index bloat

- monitoring pg_relation_size()
- problems: disk space, reduced performance
- solution: rebuilding indexes

Indexes are quite complex structures compared to tables. Under heavy load of rows deletions and insertions, an index may unreasonably increase in size.

For example, if we are talking about B-trees, an index page must contains values in a specified order. When there is no space on a page to insert a new value, the page is splitted in two, which can never be merged again, even if all the rows on these pages are deleted. The vacant space is certainly available for insertion of new rows, but the size of the index won't reduce.

The administrator must look after the indexes and rebuild them if necessary.

https://wiki.postgresql.org/wiki/Show_database_bloat

https://postgrespro.com/docs/postgresql/11/pgstattuple

Rebuilding indexes

REINDEX

`REINDEX INDEX` *index*	rebuild one index
`REINDEX TABLE` *table*	rebuild all indexes on the table
`REINDEX DATABASE` *database*	rebuild all indexes in the database
`REINDEX SYSTEM`	rebuild all indexes

rebuilds the index, minimizing the space occupied

acquires an exclusive lock on the index

VACUUM FULL

rebuilds indexes along with the table

acquires an exclusive lock on the table and the index

Rebuilding can be done with the `REINDEX` command. It creates a new copy of the index and replaces the old one with it. However, it acquires an exclusive lock on the index during its operations, and also blocks writes to the table. So almost any concurrent work with the table will be suspended.

Indexes are also rebuilt when performing `VACUUM FULL`, and also with exclusive locking. In fact, both commands work in the same way, but `REINDEX` processes only indexes, leaving the table unchanged.

https://postgrespro.com/docs/postgresql/11/sql-reindex

Rebuilding indexes

Rebuild concurrently

```
CREATE INDEX new ON ... CONCURRENTLY;
DROP INDEX old;
```

there can be several indexes on the same columns

some index types do not support concurrent build

concurrent index build is not transactional

concurrent index build may fail

Indexes supporting integrity constraints

```
CREATE UNIQUE INDEX new ON ... CONCURRENTLY;
ALTER TABLE ...
  DROP CONSTRAINT old_constraint,
  ADD CONSTRAINT new_constraint [UNIQUE|PRIMARY KEY]
    USING INDEX new;
```


An alternative way is to manually rebuild the index in several steps.

First, a new index is created. Normal index build blocks concurrent writes to the table, so the index should be built in concurrent mode (keyword `CONCURRENTLY`).

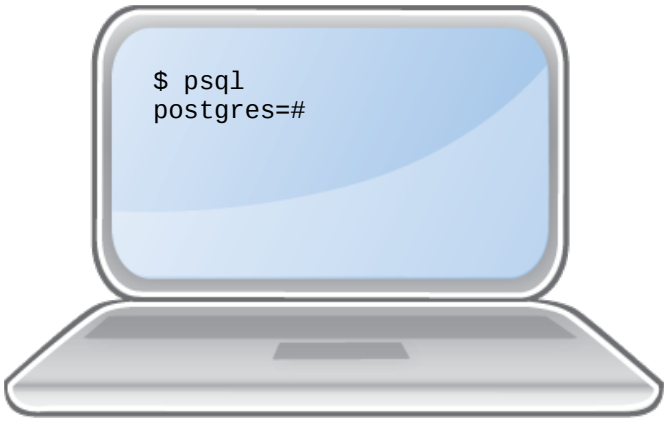
However, concurrent built takes more time to complete and may fail (due to internal deadlocks). In the latter case, the index has to be deleted and rebuilt again. Also note that concurrent built cannot be performed within a transaction.

Then the old index can be deleted. Or, if the old index supports an integrity constraint, this constraint is deleted (along with the old index) and a new constraint is created.

https://postgrespro.com/docs/postgresql/11/sql-createindex

https://postgrespro.com/docs/postgresql/11/sql-altertable

Demonstration



Summary

The vacuum process is responsible for many tasks:

- cleaning up dead tuples
- collecting statistics for the planner
- updating visibility map and free space map
- freezing tuples

Autovacuum should be enabled and requires configuration

Indexes require monitoring and (sometimes) rebuilding

Practice

1. Disable the autovacuum process and make sure it doesn't work.
2. In a new database create a table with one numeric column and an index on this table.
3. Insert into the table 100,000 random numbers.
4. Several times change half the rows of the table, controlling at each step the size of the table and the index.
5. Perform full vacuum of the table.
6. Repeat step 4, running vacuum (normal, not full) after each change. Compare the results.
7. Re-enable the autovacuum process.

1. Set the *autovacuum* parameter to `off` and signal the server to re-read the settings.

7. Set the *autovacuum* parameter to `on` and signal the server to re-read the settings.