

Architecture Vacuum

Postgres
PROFESSIONAL



15

Copyright

© Postgres Professional, 2023

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

Cover photo by Oleg Bartunov (Phu monastery and Bhrikuti peak, Nepal)

Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Agenda



Routine tasks that require periodic execution

Vacuum and analysis

Table and index bloating

Full vacuum and rebuilding of indexes

Vacuuming pages to remove MVCC historical data

- dead row versions are vacuumed out of tables

- index entries referring to dead versions are vacuumed from indexes

MVCC makes it possible to effectively implement snapshot isolation, but as a result, old versions of rows accumulate in table pages, and links to these versions accumulate in index pages. Historical versions are needed for some time so that transactions can work with their data snapshots. But over time, a row version will no longer have any snapshot that needs it. Such a row version is called "dead".

The vacuum procedure cleans out dead row versions from table pages, as well as unnecessary index entries that reference such versions.

If historical data is not vacuumed in a timely manner, tables and indexes will bloat uncontrollably and the search for current row versions in them will slow down.

<https://postgrespro.com/docs/postgresql/15/routine-vacuuming>

Updating the visibility map

tracks pages where all row versions are visible in all snapshots

used to optimize vacuuming and speed up index access

exists only for tables

In addition to this main task, vacuuming also performs other instance maintenance tasks. Vacuuming updates the visibility map and the free space map. This is service information that is stored alongside the main data.

The visibility map shows pages that contain only the current row versions visible in all data snapshots. In other words, these are pages that have not changed for long enough to be cleared out of outdated row versions.

The visibility map has several uses:

- Vacuuming optimization.
The marked pages cannot contain dead row versions, so they can be skipped during vacuuming.
- Index Only access speedup.
Versioning information is stored only for tables, but not for indexes (that's why indexes don't have visibility maps). After getting a reference to a row version from an index, you usually need to read the table page to check its visibility. But if the index itself already has all the necessary columns, and at the same time the page is marked in the visibility map, then reading the table page can be skipped.

If the visibility map isn't updated regularly, index access can slow down. This is described in more detail in the course "Query Performance Tuning" (QPT).

Updating the free space map

- tracks the free space in the pages after vacuuming
- used when inserting new row versions
- exists for both tables and indexes

The visibility map tracks available free space within pages. This space is constantly changing, decreasing when new row versions are added and increasing when they are removed.

When inserting new row versions, the map helps to quickly find a suitable page to record the data into. The free space map has a complex tree-like structure designed to improve search speed.

Indexes can have free space maps as well. However, since index entries are inserted into specific positions within an index, the map only tracks empty pages, which form when all index entries are deleted from them. These pages are excluded from the index and later can be included again into another part of the index.

Updating statistics

- used by the query optimizer
- calculated based on a random sample

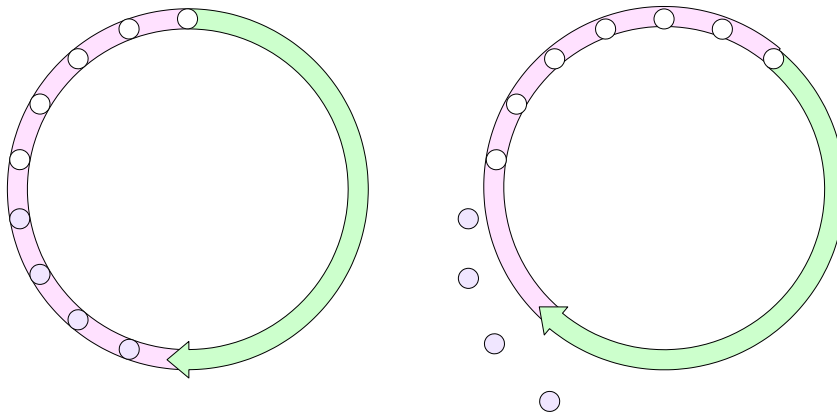
The query optimizer requires statistical information about the data it's working with, such as the number of rows in tables and the distribution of data in columns. The process of collecting the statistics is called analysis.

For analysis, a random sample of data of a certain size is read from the table. This way, the system can quickly collect statistics even on very large tables. The result is not accurate, but it is not expected to be. The data constantly changes, so it is impossible to maintain absolutely accurate statistics all of the time anyway. It is sufficient to keep it relatively up-to-date and relatively close to the truth.

If statistics are not updated regularly, they will no longer represent the data accurately, leading to the optimizer proposing inefficient execution plans. Because of this, queries may start executing orders of magnitude slower than they could.

Freezing

prevents the consequences of the 32-bit transaction counter overflow



7

As already mentioned, PostgreSQL orders events by transaction ID. The counter has 32 bits allocated for it, and sooner or later it will overflow (this is known as *transaction ID wraparound*). But when the counter resets to zero, the order of transactions will be disrupted.

This is why the transaction ID scope is looped. For each transaction, half of the IDs are considered to be in the future, half in the past.

Sufficiently old row versions are marked as frozen. This means that they were created so far in the past that their transaction ID no longer means anything and can be reused.

To avoid scanning extra pages, the visibility map has a bit that marks the pages where all row versions are frozen.

Without regular freezing, the server may end up with no available transaction IDs for new transactions. This is an emergency: the server stops, all active transactions drop, and the administrator has to manually restart the server and do the freezing.

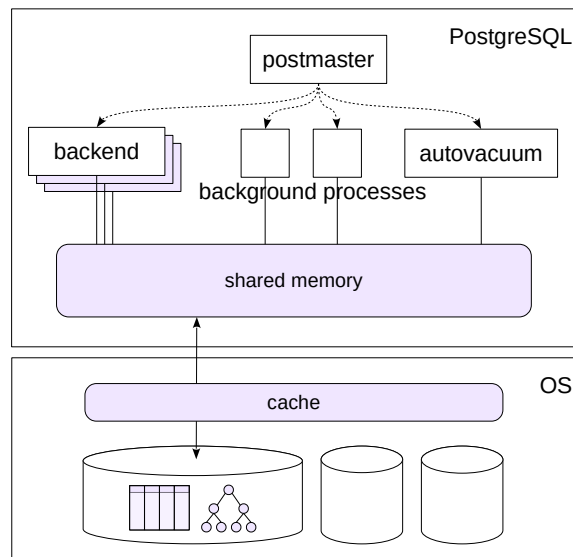
Autovacuum

Autovacuum launcher

background process
periodically launches
worker processes

Autovacuum worker

vacuums tables of a
separate database that
require attention



8

All the maintenance tasks discussed above are taken care of by the autovacuum process. It dynamically reacts to the frequency of table updates, and the more active the changes, the more often the table will be vacuumed.

The autovacuum launcher process runs in the background, scheduling the vacuuming work and launching the required number of parallel autovacuum workers.

Vacuuming works page-by-page, so it doesn't block other transactions. It still does require additional resources from the I/O system.

Autovacuum will stop when either of the two parameters *autovacuum* or *track_counts* is switched off. It may seem that disabling autovacuum can increase system performance by eliminating "unnecessary" I/O operations, but it cannot. Failure to vacuum will lead to the consequences described above: uncontrolled bloating, slower query processing, and the risk of an emergency server shutdown. Ultimately, this will lead to a complete system paralysis.

Autovacuuming is absolutely necessary. There is a large number of configuration parameters that allow tweaking the autovacuum process. They are discussed in detail in the course DBA2 "Configuration and monitoring".

Manual vacuuming



Vacuuming

<code>VACUUM [table, ...]</code>	vacuuming specific tables
<code>VACUUM</code>	vacuuming the entire database
<code>\$ vacuumdb</code>	wrapper for the OS

Analysis

<code>ANALYZE</code>
<code>\$ vacuumdb --analyze-only</code>

Vacuum and analysis

<code>VACUUM ANALYZE</code>
<code>\$ vacuumdb --analyze</code>

9

If necessary, vacuuming and analysis can be started manually using the following commands:

`VACUUM` (vacuuming only), `ANALYZE` (analysis only), and `VACUUM ANALYZE` (both vacuuming and analysis).

Autovacuum is different from running scheduled vacuum and analysis in that it reacts to the frequency of data changes. Running it on a schedule too often will create unnecessary load on the system. On the other hand, if you vacuum too rarely, and data is changed often, the files may have time to bloat significantly between vacuums.

<https://postgrespro.com/docs/postgresql/15/sql-vacuum>

<https://postgrespro.com/docs/postgresql/15/sql-analyze>

Vacuum

Let's create a table and turn autovacuum off so that we can vacuum manually:

```
=> CREATE TABLE bloat(  
  id integer GENERATED ALWAYS AS IDENTITY,  
  d timestampz  
) WITH (autovacuum_enabled = off);
```

CREATE TABLE

Fill the table with data and build an index:

```
=> INSERT INTO bloat(d)  
  SELECT current_timestamp FROM generate_series(1,100000);
```

INSERT 0 100000

```
=> CREATE INDEX ON bloat(d);
```

CREATE INDEX

All table rows have only one row version.

Now, let's update a row:

```
=> UPDATE bloat SET d = d + interval '1 day' WHERE id = 1;
```

UPDATE 1

Run vacuum and have it tell us what it is doing:

```
=> VACUUM (verbose) bloat;
```

```
INFO:  vacuuming "arch_vacuum_overview.public.bloat"  
INFO:  finished vacuuming "arch_vacuum_overview.public.bloat": index scans: 0  
pages: 0 removed, 541 remain, 541 scanned (100.00% of total)  
tuples: 1 removed, 100000 remain, 0 are dead but not yet removable  
removable cutoff: 750, which was 0 XIDs old when operation ended  
new relfrozenxid: 747, which is 1 XIDs ahead of previous value  
index scan bypassed: 1 pages from table (0.18% of total) have 1 dead item identifiers  
avg read rate: 2.228 MB/s, avg write rate: 4.456 MB/s  
buffer usage: 1086 hits, 2 misses, 4 dirtied  
WAL usage: 542 records, 1 full page images, 40350 bytes  
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s  
VACUUM
```

Here we can see that vacuum has scanned all the table pages and removed one row version.

Vacuuming does not reduce the size of tables and indexes

the “holes” in the pages are used for new data,
but never returned to the operating system

Causes of bloating

- incorrect autovacuum configuration
- massive changes of data
- long-running transactions

Negative consequences

- inefficient disk space use
- slower sequential table scan
- less efficient index access

Vacuuming cleans out outdated row versions from pages. This creates spots of free space in the pages, which is then used to store new data. But the free space is not returned to the operating system, so, from the point of view of the OS, the size of the data files does not decrease.

In the case of indexes (B-trees), it is complicated by the fact that if there is not enough space in the page to place an index entry, the page is split into two. The resulting pages are never merged again, even if all index entries are removed from them.

If autovacuuming is configured correctly, the data files grow by a certain constant amount due to updates between vacuumings. But if a large amount of data is being changed at the same time, or there are active long transactions (keeping old data snapshots active and not allowing you to vacuum out old row versions), vacuuming will not be able to free up the space in time. As a result, the tables and indexes may continue to grow in size.

File size bloating leads not only to disk space overuse (including for backups), but also to a decrease in performance.

https://wiki.postgresql.org/wiki/Show__database__bloat

<https://postgrespro.com/docs/postgresql/15/pgstattuple>

Table and index bloating estimation

There are several ways to assess bloating and its impact:

- Queries to the system catalog
- The pgstattuple extension

```
=> CREATE EXTENSION pgstattuple;
```

```
CREATE EXTENSION
```

The extension helps monitor the state of a table:

```
=> SELECT * FROM pgstattuple('bloat') \gx
```

```
-[ RECORD 1 ]-----+-----
table_len      | 4431872
tuple_count    | 100000
tuple_len      | 4000000
tuple_percent  | 90.26
dead_tuple_count | 0
dead_tuple_len | 0
dead_tuple_percent | 0
free_space     | 16720
free_percent   | 0.38
```

- tuple_percent is data payload (never 100% because of overhead).

Same for an index:

```
=> SELECT * FROM pgstatindex('bloat_d_idx') \gx
```

```
-[ RECORD 1 ]-----+-----
version        | 4
tree_level     | 1
index_size     | 712704
root_block_no  | 3
internal_pages | 1
leaf_pages     | 85
empty_pages    | 0
deleted_pages  | 0
avg_leaf_density | 89.17
leaf_fragmentation | 0
```

- leaf_pages is the number of leaf pages in the index,
- avg_leaf_density is how full they are,
- leaf_fragmentation is how ordered they are on disk.

Let's update half the rows at once:

```
=> UPDATE bloat SET d = d + interval '1 day' WHERE id % 2 = 0;
```

```
UPDATE 50000
```

Check the table again:

```
=> SELECT * FROM pgstattuple('bloat') \gx
```

```
-[ RECORD 1 ]-----+-----
table_len      | 6643712
tuple_count    | 100000
tuple_len      | 4000000
tuple_percent  | 60.21
dead_tuple_count | 50000
dead_tuple_len | 2000000
dead_tuple_percent | 30.1
free_space     | 21000
free_percent   | 0.32
```

The density went down.

To avoid scanning the whole table, pgstattuple can display approximate information:

```
=> SELECT * FROM pgstattuple_approx('bloat') \gx
```

```

-[ RECORD 1 ]-----+-----
table_len      | 6643712
scanned_percent | 100
approx_tuple_count | 100000
approx_tuple_len | 4000000
approx_tuple_percent | 60.207305795314426
dead_tuple_count | 50000
dead_tuple_len   | 2000000
dead_tuple_percent | 30.103652897657213
approx_free_space | 21000
approx_free_percent | 0.31608835542540076

```

Check the index:

```
=> SELECT * FROM pgstatindex('bloat_d_idx') \gx
```

```

-[ RECORD 1 ]-----+-----
version        | 4
tree_level     | 1
index_size     | 1040384
root_block_no  | 3
internal_pages | 1
leaf_pages     | 125
empty_pages    | 0
deleted_pages  | 0
avg_leaf_density | 90.95
leaf_fragmentation | 0

```

Leaf page density remained the same, but the number of pages has increased.

Rebuilding objects

Full vacuum

```
VACUUM FULL
```

```
$ vacuumdb --full
```

completely rebuilds the contents of tables and indexes

locks the table completely

Rebuilding indexes

```
REINDEX
```

rebuilds indexes

locks the index completely and locks

the associated table for write operations

In order to reduce the physical size of bloated tables and indexes, a *full vacuum* is required.

The VACUUM FULL command completely rewrites the contents of the table and its indexes, minimizing the space occupied. However, this process requires an exclusive table lock and therefore cannot be executed in parallel with other transactions.

<https://postgrespro.com/docs/postgresql/15/sql-vacuum>

You can rebuild an index or several indexes without touching the table. This is done by the REINDEX command. It locks the table for writing (reading is still available), so transactions trying to change the table will be blocked.

<https://postgrespro.com/docs/postgresql/15/sql-reindex>

If prolonged exclusive locking is undesirable, you can consider `pg_repack`, a third-party extension (https://github.com/reorg/pg_repack) that allows you to rebuild tables and their indexes on the fly.

Non-blocking index rebuilding

`REINDEX ... CONCURRENTLY`

rebuilds indexes without blocking tables for writing

takes longer and may fail

not transactional

does not work for system indexes

does not work for indexes associated with exclusion constraints

In PostgreSQL 12 and higher, the `REINDEX ... CONCURRENTLY` command can work without blocking the table for writing. However, non-blocking rebuilding takes longer and may fail (due to deadlocks). In this case, the index will need to be rebuilt again.

Non-blocking index rebuilding has a number of limitations: it cannot be performed inside a transaction, and it cannot rebuild system indexes and indexes with exclusion constraints (`EXCLUDE`).

Rebuilding objects

You can rebuild an index with the REINDEX command with the CONCURRENTLY keyword. It rebuilds the index without stopping the system.

```
=> REINDEX TABLE CONCURRENTLY bloat;
```

REINDEX

Check the index again:

```
=> SELECT * FROM pgstatindex('bloat_d_idx') \gx
```

```
-[ RECORD 1 ]-----+-----
version      | 4
tree_level   | 1
index_size   | 712704
root_block_no | 3
internal_pages | 1
leaf_pages   | 85
empty_pages  | 0
deleted_pages | 0
avg_leaf_density | 89.17
leaf_fragmentation | 0
```

The density is back to nominal.

To rebuild a table together with its indexes, you can use VACUUM FULL. However, unlike REINDEX .. CONCURRENTLY, it does lock the table.

```
=> VACUUM FULL bloat;
```

VACUUM

```
=> SELECT * FROM pgstattuple('bloat') \gx
```

```
-[ RECORD 1 ]-----+-----
table_len      | 4431872
tuple_count    | 100000
tuple_len      | 4000000
tuple_percent  | 90.26
dead_tuple_count | 0
dead_tuple_len | 0
dead_tuple_percent | 0
free_space     | 16724
free_percent   | 0.38
```

The density has increased, the freed up space is returned to the operating system.

Row versions accumulate, so periodic vacuuming is necessary

Vacuuming serves multiple goals:

- updating visibility maps and free space maps
- collecting statistics for the planner
- freezing old row versions

Autovacuuming is necessary, but requires configuration

Full vacuum may be necessary to combat bloating

1. Disable autovacuuming and make sure it doesn't work.
2. In a new database, create a table with one numeric column and an index for this table. Insert 100,000 random numbers into the table.
3. Change half of the table rows several times. Write down the size of the table and the index each time.
4. Run a full vacuum.
5. Repeat step 4, running a regular vacuum each time you change the values. Compare the results.
6. Turn autovacuuming back on.

Task 1. Set the *autovacuum* parameter to off and reload the configuration files.

Task 3. Use the functions `pg_table_size (table_name)` and `pg_indexes_size (table_name)`. For more information about calculating the sizes of various objects, see the module "Data organization".

Task 6. Set the *autovacuum* parameter back to on (or reset this parameter with the RESET command), then reload the configuration files.

Task 1. Switching autovacuum off

```
=> SELECT pid, backend_start, backend_type
FROM pg_stat_activity
WHERE backend_type = 'autovacuum launcher';

 pid |          backend_start          | backend_type
-----+-----+-----
17300 | 2023-05-04 19:50:35.804474+03 | autovacuum launcher
(1 row)
```

```
=> ALTER SYSTEM SET autovacuum = off;
```

ALTER SYSTEM

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

```
=> SELECT pid, backend_start, backend_type
FROM pg_stat_activity
WHERE backend_type = 'autovacuum launcher';

 pid | backend_start | backend_type
-----+-----+-----
(0 rows)
```

Task 2. Database, table and index

Create a database, a table and an index:

```
=> CREATE DATABASE admin_maintenance;
```

CREATE DATABASE

```
=> \c admin_maintenance
```

You are now connected to database "admin_maintenance" as user "student".

```
=> CREATE TABLE t(n numeric);
```

CREATE TABLE

```
=> CREATE INDEX t_n on t(n);
```

CREATE INDEX

Insert rows:

```
=> INSERT INTO t SELECT random() FROM generate_series(1,100000);
```

INSERT 0 100000

Store a query to calculate the size of the table and the index as a psql variable so that we can use it later:

```
=> \set SIZE 'SELECT pg_size_pretty(pg_table_size(''t'')) table_size, pg_size_pretty(pg_indexes_size(''t'')) index_size\g (footer=off)'
=> :SIZE
```

```
table_size | index_size
-----+-----
4360 kB    | 4536 kB
```

Task 3. Changing rows without vacuuming

```
=> UPDATE t SET n=n WHERE n < 0.5;
```

UPDATE 49969

```
=> :SIZE
```

```
table_size | index_size
-----+-----
6520 kB    | 6808 kB
```

```
=> UPDATE t SET n=n WHERE n < 0.5;
```

UPDATE 49969

```
=> :SIZE
```

```
table_size | index_size
-----+-----
8680 kB    | 7568 kB
```

```
=> UPDATE t SET n=n WHERE n < 0.5;
```

UPDATE 49969

```

=> :SIZE

table_size | index_size
-----+-----
11 MB      | 11 MB

```

The table and index sizes keep growing.

Task 4. Full vacuum

```

=> VACUUM FULL t;

VACUUM

=> :SIZE

table_size | index_size
-----+-----
4336 kB    | 3104 kB

```

The table size is almost back to where it was initially, and the index size decreased (building an index for a large data set is more efficient than keeping adding data to the index row by row).

Task 5. Changing rows with vacuuming

```

=> UPDATE t SET n=n WHERE n < 0.5;

UPDATE 49969

=> VACUUM t;

VACUUM

=> :SIZE

table_size | index_size
-----+-----
6528 kB    | 4648 kB

=> UPDATE t SET n=n WHERE n < 0.5;

UPDATE 49969

=> VACUUM t;

VACUUM

=> :SIZE

table_size | index_size
-----+-----
6528 kB    | 4648 kB

=> UPDATE t SET n=n WHERE n < 0.5;

UPDATE 49969

=> VACUUM t;

VACUUM

=> :SIZE

table_size | index_size
-----+-----
6528 kB    | 4648 kB

```

The size increased once and then stabilized.

The example demonstrates how removing (or changing) large chunks of data should be done in multiple transactions, if possible. This will allow autovacuum to clean up unneeded row versions in time, avoiding table bloating.

Task 6. Turn autovacuum back on

```

=> ALTER SYSTEM RESET autovacuum;

ALTER SYSTEM

=> SELECT pg_reload_conf();

pg_reload_conf
-----
t
(1 row)

```