6-MAR-25

Tantor: PostgreSQL 16 Performance tuning

Practices

Oleg Ivanov

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Table of contents

Tantor: PostgreSQL 16 Performance Tuning	page
Practice for Chapter 1	
Part 1. Standard pgbench test	5
Part 2. Using pgbench with your own script	8
Part 3. Using the sysbench utility	14
Part 4. Using the HammerDB application	15
Part 5. Using the Go-TPC application	22
Practice for Chapter 2	
Part 1. Launching a Huge Pages Instance	27
Part 2. Changing the oom_score value	29
Part 3. Unloading long lines with the pg_dump utility	31
Part 4. Out of Memory	32
Part 5. Enabling swap	37
Part 6. Page cache	39
Practice for Chapter 3	
Part 1. Standard pgbench test	42
Part 2. Binding processes to a processor core	43
Part 3. Switching execution contexts	44
Part 4. Monitoring CPU load	46
Part 5. Collecting statistics into a file and viewing it with the atop utility	50
Part 6. Linux Time Source	51
Part 7. Network connections	54
Part 8. Replacing the scheduling policy and checking the scheduler operation	55
Practice for Chapter 4	
Part 1. Disk subsystem parameters	58
Part 2. Installing packages in Astralinux	61
Part 3. Working with SSD and testing disk performance with the fio utility	64
Part 4. Testing the ext4 fastcommit journal	67
Part 5. Removing the limit on the number of open files	73
Part 6. Example of command files for testing	75
Part 7. Example of creating programs for testing	79
Practice for Chapter 5	
Part 1. Object locks	82
Part 2. Monitoring the server process memory	84
Part 3. Temporary tables and files	91
Part 4. Impact of configuration parameters on shared memory	96
Part 5. max_connections parameter and performance	99
Part 6. Buffer Cache Size and Buffer Release	103
Practice for Chapter 6	
Part 1. Free space map	106
Part 2. Changing the order of columns	110
Part 3. Table Block Contents	119
Part 4. Aligning fields in table rows	121
Part 5. Aligning Rows in Table Blocks	123
Part 6. Storing empty (NULL) values in table rows	126
Part 7. Number of rows in a table block	128
Practice for Chapter 7	
Part 1. Access methods	133
Part 2. Using Indexes with Integrity Constraints	138
Part 3. Characteristics of btree indexes	141
Part 4. Navigating the btree index structure	145
Part 5. Deduplication in btree indexes	149
Part 6. Indexes in descending order	152
Part 7. Covering indices and Index Only Scan	154

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Part 8. Partial indices	155
Part 9. Studying the structure of the btree index	156
Part 10. Cleaning index blocks during its scanning	168
Part 11. Slow query execution on replica due to lack of index block cleaning	173
Part 12. Determining the number of deleted lines	177
Part 13. Search by btree index structure	179
Practice for Chapter 8	
Part 1. TOAST	183
Part 2. TOAST Table Structure	189
Part 3. UPDATE vs. INSERT Efficiency	194
Part 4. HOT cleanup	197
Part 5. Monitoring HOT cleanup	204
Part 6. Small Data Types	207
Part 7. Storage of variable-length data types	210
Part 8. Data type for the primary key column	215
Part 9. Data types for storing dates and times	217
Part 10. Data types float and real	219
Practice for Chapter 9	
Part 1. Launching an instance in a docker container	223
Part 2. Shared Instance Memory	232
Part 3. Local memory of the server process	238
Part 4. Logging process memory to the diagnostic log	245
Part 5. Deadlocks during testing	246
Part 6. Multitransactions	248
Part 7. Test example	252
Practice for Chapter 10	
Part 1. Extending pg_buffercache	256
Part 2. Buffer rings	259
Part 3. pg_prewarm extension	263
Part 4. The bgwriter background writing process	265
Practice for Chapter 11	
Part 1. Setting the frequency of checkpoints	270
Part 2. Delay in instance startup. recovery_init_sync_method parameter	276
Part 3. Checkpoint duration	283
Part 4. Duration of the final checkpoint when stopping the instance	287
Part 5. Duration of checkpoint after instance crash	288
Part 6. Checkpoint on request	290
Practice for Chapter 12	
Part 1. Vacuum command parameters	296
Part 2: Vacuum Observation	299
Part 3. Extension for viewing visibility map and freezing pg_visibility	302
Part 4. Interval between autovacuum cycles, autovacuum_naptime parameter	304
Part 5. Comparison of vacuum version 17 with previous versions	306
Part 6. Number of index scans during vacuuming	311
Part 7. Autovacuum logging	316
Part 8. pgststtuple extension	319
Part 9: Index Processing Condition: 2% of Rows	320
Practice for Chapter 13	
Part 1. Reading vacuum and autovacuum messages	324
Part 2. Reading checkpoint messages	325
Part 3. Reading pg_waldump checkpoint messages	329
Part 4. Size of the PGDATA/pg_wal directory	332
Practice for Chapter 14	
Part 1. I/O statistics in the pg_stat_io view	339
Part 2. Running fsyncs when checkpointer is stopped	344

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Part 3. I/O Performance Testing	350
Part 4. Choosing the size of temp_buffers when working with temporary tables using pg_stat_io	354
Part 5. Example of analysis of statistics of vacuum command operation with a ring	358
Part 6. bgwriter operation and statistics comparison in pg_stat_bgwriter and pg_stat_io views	362
Part 7. Running bgwriter on 128MB and 1GB buffer cache	366
Part 8. Using the pg_walinspect extension	371
Part 9. Monitoring Blocks	376
Practice for Chapter 15	
Part 1. Installing the pg_stat_kcache extension	380
Part 2. Using the pg_stat_kcache extension	384
Part 3. Performance when using direct i/o	390
Practice for Chapter 16	
Using the pg_wait_sampling extension	392

Copyright

The textbook, practical assignments, presentations (hereinafter referred to as documents) are intended for educational purposes.

The documents are protected by copyright and intellectual property laws.

You may copy and print documents for personal use for self-study purposes, as well as for training in training centers and educational institutions authorized by Tantor Labs LLC. Training centers and educational institutions authorized by Tantor Labs LLC may create training courses based on the documents and use the documents in training programs with the written permission of Tantor Labs LLC.

You may not use the documents for training employees or others without permission from Tantor Labs LLC. You may not license or commercially use the documents in whole or in part without permission from Tantor Labs LLC.

For non-commercial use (presentations, reports, articles, books) of information from documents (text, images, commands), keep a link to the documents.

The text of the documents cannot be changed in any way.

The information contained in the documents may be changed without prior notice and we do not guarantee its accuracy. If you find errors, copyright infringement, please inform us about it.

Disclaimer of liability for the content of the document, products and services of third parties:

Tantor Labs, LLC and its affiliates are not responsible for and expressly disclaim any warranties of any kind, including loss of income, whether direct or indirect, special or incidental, arising from the use of the document. Tantor Labs, LLC and its affiliates are not responsible for any losses, costs or damages arising from the use of the information contained in the document or the use of third-party links, products or services.

Copyright © 2025, Tantor Labs LLC

Created by : Oleg Ivanov

Created: 6 March 2025

For training questions, please contact: edu@tantorlabs.ru

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Practice for Chapter 1

Part 1. Standard pgbench test

Virtual machine image for the course: https://disk.yandex.ru/d/APErrktFq-Gamg

1) When running the practices, the instance will be restarted many times. Create a file with a short name to conveniently restart the instance:

```
astra@tantor:~$ su -
Password: root
root@tantor:~# echo "systemctl restart tantor-se-server-16" > /usr/local/bin/restart
root@tantor:~# chmod 755 /usr/local/bin/restart
root@tantor:~# <ctrl+d>
logout
astra@tantor:~$ sudo restart
astra @ tantor :~$
```

You created a batch file and restarted the instance using it.

2) Switch to the postgres operating system user :

```
astra@tantor:~$ su - postgres
Password: postgres
postgres@tantor:~$
```

3) The pgbench utility comes with postgres and uses tables for built-in tests.
Create tables for p g bench tests :

```
postgres@tantor:~$ pgbench -i
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.10 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.43 s (drop tables 0.00 s, create tables 0.02 s, client-side generate 0.28 s, vacuum
0.07 s, primary keys 0.07 s).
```

`pgbench_accounts` table has `100 thousand rows, the other tables` have 1, 0 and 10 rows.

4) Run the test by default for 30 seconds with output of intermediate results at 5 second intervals:

```
postgres@tantor:~$ pgbench -T 30 -P 5
starting vacuum...end.
progress: 5.0 s, 547.2 tps, lat 1.818 ms stddev 0.475, 0 failed
progress: 10.0 s, 551.6 tps, lat 1.805 ms stddev 0.230, 0 failed
progress: 15.0 s, 545.8 tps, lat 1.824 ms stddev 0.255, 0 failed
progress: 20.0 s, 554.2 tps, lat 1.797 ms stddev 0.204, 0 failed
progress: 25.0 s, 532.6 tps, lat 1.870 ms stddev 0.298, 0 failed
progress: 30.0 s, 502.6 tps, lat 1.982 ms stddev 2.024, 0 failed
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
Number of clients: 1
Number of threads: 1
Maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 16171
number of failed transactions: 0 (0.000%)
```

```
latency average = 1.847 ms
stddev latency = 0.849 ms
initial connection time = 3.973 ms
tps = 539 .088500 (without initial connection time)
```

The example shows the result: 539 transactions per second (tps)

5) Check how the database horizon retention affects the TPS of the standard test. Run the test by specifying a large time interval:

```
postgres @ tantor :~$ pgbench - T 10000 - P 15
```

6) Launch the second terminal. In the terminal, run psl and start the transaction:

```
astra@tantor:~$ psql
postgres=# begin;
BEGIN
postgres =*#
```

7) In the window with the running test, pay attention to the tps indicator:

```
progress: 15.0 s, 541.5 tps, lat 1.838 ms stddev 0.242, 0 failed
progress: 30.0 s, 554.7 tps, lat 1.795 ms stddev 0.255, 0 failed
progress: 45.0 s, 545.9 tps, lat 1.824 ms stddev 0.279, 0 failed
```

8) Open a transaction by calling a function that returns the transaction number:

```
postgres=*# select pg_current_xact_id();
pg_current_xact_id
--------------------
974640
(1 row)
postgres =*#
```

9) Check that in the window with the test tps will gradually start to decrease:

```
progress: 60.0 s, 484.6 tps, lat 2.055 ms stddev 0.603, 0 failed
progress: 75.0 s, 395.7 tps, lat 2.517 ms stddev 0.751, 0 failed
progress: 90.0 s, 349.1 tps, lat 2.853 ms stddev 0.793, 0 failed
progress: 105.0 s, 319.5 tps, lat 3.118 ms stddev 0.912, 0 failed
progress: 120.0 s, 308.1 tps, lat 3.237 ms stddev 0.942, 0 failed
progress: 135.0 s, 339.8 tps, lat 2.935 ms stddev 0.507, 0 failed
progress: 150.0 s, 306.3 tps, lat 3.257 ms stddev 0.478, 0 failed
progress: 165.0 s, 277.6 tps, lat 3.595 ms stddev 0.706, 0 failed
progress: 180.0 s, 264.2 tps, lat 3.777 ms stddev 0.673, 0 failed
```

In three minutes, tps will decrease by 1.5-2 times. In an hour, tps will decrease by 100 times. We will not wait for an hour, an hour-long test can be launched at lunchtime.

10) Run a query showing the PIDs of active processes and the duration of their transactions:

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start))
secs, pid, datname database, state from pg_stat_activity where backend_xmin IS
NOT NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc;
age | secs | pid | database | state
--------+------------+--------+----------+--------------------
175455 | 1425.651346 | 255554 | postgres | idle in transaction
1 | 0.001878 | 255547 | postgres | active
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
      1 | 0.001213 | 255626 | postgres | active
```

This is a query that allows you to monitor the horizon of databases. The query returns the number of transactions that have occurred since the transaction was started by the server process to which the row belongs; the duration of the transaction, in the first line the longest transaction and the horizon hold time; the pid of the process, the name of the database in which the transaction is running; the state of the transaction. If the state of the transaction is " `idle in transaction` " this means that the transaction is idle and waiting for a command from the client. An idle open transaction is undesirable and should not exist in a well-written application.

In the example, 1425 seconds have passed and during this time the tps time in the test window has decreased by 7 times:

```
progress: 1425.0 s, 71.6 tps, lat 13.964 ms stddev 1.215, 0 failed
```

11) Complete the transaction holding the horizon in the second terminal window:

```
postgres=*# rollback;
ROLLBACK
postgres=#
```

12) repeat the request in the third terminal window:

```
postgres=# select age(backend_xmin), extract(epoch from (clock_timestamp()-
xact_start)) secs, pid, datname database, state from pg_stat_activity where
backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
 age | secs | pid | database | state
----------+----------+--------+----------+--------
 1 | 0.001050 | 255547 | postgres | active
 1 | 0.001280 | 255626 | postgres | active
```

Two transactions This transactions pgbench test . Their "age" in the number of transactions is 1. They do not hold the horizon.

13) Look in the first terminal window with the test that the tps will return to the original quite quickly:

```
progress: 1905.0 s, 61.3 tps, lat 16.302 ms stddev 1.201, 0 failed
progress: 1920.0 s, 60.4 tps, lat 16.549 ms stddev 1.762, 0 failed
progress: 1935.0 s, 159.5 tps, lat 6.266 ms stddev 6.035, 0 failed
progress: 1950.0 s, 389.1 tps, lat 2.561 ms stddev 0.586, 0 failed
progress: 1965.0 s, 535.5 tps, lat 1.860 ms stddev 0.347, 0 failed
progress: 1980.0 s, 522.1 tps, lat 1.908 ms stddev 0.327, 0 failed
```

You have observed that holding the horizon in the database significantly reduces the number of transactions that the DBMS can service. A single query holds the horizon for the duration of its execution. Long queries should be moved to replicas.

the default transaction isolation level (`READ COMMITTED`) **, a transaction begins to hold a** horizon as soon as it is assigned a real transaction number (`xid`). **The pg_current_xact_id () function** was used to obtain the real number **.** The real number will be assigned when any command that changes data is executed. For example: INSERT, UPDATE, DELETE, CREATE, ALTER, DROP. Transactions at the **repeatable isolation level read** hold the horizon from the moment any command is executed, including SELECT, and hold the horizon until the transaction is completed:

```
postgres=# begin transaction isolation level repeatable read;
BEGIN
postgres =*# select 1;
?column?
----------
1
```

Oleg Ivanov

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
(1 row)
```

Part 2. Using pgbench with your own script

1) Create a table for the test by running the following commands in psql:

```
postgres=# drop table if exists t;
create table t(pk bigserial, c1 text default 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');
insert into t select *, 'a' from generate_series(1, 100000);
alter table t add constraint pk primary key (pk);
DROP TABLE
CREATE TABLE
INSERT 0 100000
ALTER TABLE
```

2) Create three scripts (batch files):

```
postgres=# \q
postgres@tantor:~$ echo "select count(*) from t;" > count1.sql
echo "select count(1) from t;" > count2.sql
echo "select count(pk) from t;" > count3.sql
postgres @ tantor :~$
```

3) Run three tests in turn with the created scripts:

```
postgres@tantor:~$ pgbench -T 30 -P 5 -f count 1 .sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 75.8 tps, lat 13.144 ms stddev 1.410, 0 failed
progress: 10.0 s, 74.2 tps, lat 13.495 ms stddev 1.454, 0 failed
progress: 15.0 s, 75.0 tps, lat 13.304 ms stddev 1.047, 0 failed
progress: 20.0 s, 75.2 tps, lat 13.313 ms stddev 1.267, 0 failed
progress: 25.0 s, 75.6 tps, lat 13.205 ms stddev 1.030, 0 failed
progress: 30.0 s, 76.2 tps, lat 13.125 ms stddev 0.896, 0 failed
transaction type: count1.sql
scaling factor: 1
query mode: simple
Number of clients: 1
Number of threads: 1
Maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 2261
number of failed transactions: 0 (0.000%)
latency average = 13.263 ms
stddev latency = 1.207 ms
initial connection time = 3.788 ms
tps = 75.339419 (without initial connection time)
postgres@tantor:~$ pgbench -T 30 -P 5 -f count 2 .sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 69.4 tps, lat 14.386 ms stddev 0.891, 0 failed
progress: 10.0 s, 67.8 tps, lat 14.707 ms stddev 1.009, 0 failed
progress: 15.0 s, 68.4 tps, lat 14.633 ms stddev 1.208, 0 failed
progress: 20.0 s, 68.4 tps, lat 14.615 ms stddev 0.909, 0 failed
progress: 25.0 s, 68.4 tps, lat 14.596 ms stddev 1.010, 0 failed
progress: 30.0 s, 67.0 tps, lat 14.940 ms stddev 0.720, 0 failed
transaction type: count2.sql
scaling factor: 1
query mode: simple
Number of clients: 1
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
Number of threads: 1
Maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 2048
number of failed transactions: 0 (0.000%)
latency average = 14.645 ms
stddev latency = 0.983 ms
initial connection time = 4.314 ms
tps = 68.242454 (without initial connection time)
postgres@tantor:~$ pgbench -T 30 -P 5 -f count 3 .sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 56.2 tps, lat 17.758 ms stddev 0.983, 0 failed
progress: 10.0 s, 55.2 tps, lat 18.095 ms stddev 0.698, 0 failed
progress: 15.0 s, 55.6 tps, lat 17.968 ms stddev 0.768, 0 failed
progress: 20.0 s, 55.4 tps, lat 18.016 ms stddev 0.765, 0 failed
progress: 25.0 s, 55.8 tps, lat 17.954 ms stddev 0.930, 0 failed
progress: 30.0 s, 55.0 tps, lat 18.125 ms stddev 1.161, 0 failed
transaction type: count3.sql
scaling factor: 1
query mode: simple
Number of clients: 1
Number of threads: 1
Maximum number of tries: 1
duration: 30 s
number of transactions actually processed: 1667
number of failed transactions: 0 (0.000%)
latency average = 17.985 ms
stddev latency = 0.906 ms
initial connection time = 4.129 ms
tps = 55.572129 (without initial connection time)
postgres@tantor:~$
```

4) Analyze the test result: determine which team works faster

5) Run the commands in psql :

```
postgres =# \ timing
Timing is on.
postgres=# select count(pk) from t;
count
--------
100000
(1 row)

Time: 18.249 ms
postgres=# select count(1) from t;
count
--------
100000
(1 row)

Time: 15.223 ms
postgres=# select count(*) from t;
count
--------
100000
(1 row )

Time : 14.535 ms
```

Does the execution time of the commands match the pgbench results? Yes, it does.
Does the command execution time match the latency value? Yes, it does.
`latency average` measured by pgbench with accuracy corresponding to `latency stddev` .

6) turn off measurement time :

```
postgres=# \timing
Timing is off.
```

7) Run the commands:

```
postgres=# explain analyze select count(pk) from t;
QUERY PLAN
--------------------------------------------------------------------------------
Aggregate (cost=1791.00.. 1791.01 rows=1 width=8) (actual time=817.468.. 817.485 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.013.. 403.349 rows=100000 loops=>
Planning Time: 0.047 ms
Execution Time: 817.602 ms
(4 rows)

postgres=# explain analyze select count(1) from t;
QUERY PLAN
--------------------------------------------------------------------------------
Aggregate (cost=1791.00.. 1791.01 rows=1 width=8) (actual time=800.453.. 800.470 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=0) (actual time=0.015.. 399.821 rows=100000 loops=>
Planning Time: 0.053 ms
Execution Time: 800.589 ms
(4 rows)

postgres=# explain analyze select count(*) from t;
QUERY PLAN
--------------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual time=804.634.. 804.650 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=0) (actual time=0.013.. 401.192 rows=100000 loops=>
Planning Time: 0.047 ms
Execution Time: 804.767 ms
(4 rows)

postgres=# explain analyze select count(*) from t;
QUERY PLAN
--------------------------------------------------------------------------------
Aggregate (cost=1791.00.. 1791.01 rows=1 width=8) (actual time=813.472.. 813.489 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=0) (actual time=0.015.. 410.861 rows=100000 loops=>
Planning Time: 0.054 ms
Execution Time: 813.610 ms
(4 rows)
```

explain commands correct? analyze command execution time without explain analyze ? No, it does not match. Command execution time is ~15 ms , command execution time with explain analyze in this example is ~800 ms or less ~300 ms (depending on the type of time source).

explain analyze with default options showed that count(*) took longer to execute than count (1), which is at odds with previous tests. The overhead of timing significantly exceeds the execution time of the commands.

The last two results of the same command `804.767 ms` and `813.610 ms` differ due to delays in accessing buffers in the buffer cache `401.192` and `410.861` .

The execution time of the Aggregate node also diverges from the results of pgbench and the execution of commands without explain analyze:

```
817.485 - 403.349 = 414.136
800.470 - 399.821 = 400.649
804.650 - 401.192 = 403.458
813.489 - 410.861 = 402.628
```

The difference in the results is that count (*) is slower than count (1).

8) Run the commands, disabling the time measurement in the explain command:

```
postgres=# explain (analyze, buffers, timing off ) select count(pk) from t;
QUERY PLAN
----------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual rows=1 loops=1)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
Buffers: shared hit=541
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=8) (actual rows=100000 loops=1)
Buffers: shared hit=541
Planning:
Buffers: shared hit=65
Planning Time: 0.431 ms
Execution Time: 21.388 ms
(8 rows)

postgres=# explain (analyze, buffers, timing off ) select count(1) from t;
QUERY PLAN
---------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual rows=1 loops=1)
Buffers: shared hit=541
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=0) (actual rows=100000 loops=1)
Buffers: shared hit=541
Planning Time: 0.059 ms
Execution Time: 18.952 ms
(6 rows)

postgres=# explain (analyze, buffers, timing off ) select count(*) from t;
QUERY PLAN
-------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual rows=1 loops=1)
Buffers: shared hit=541
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=0) (actual rows=100000 loops=1)
Buffers: shared hit=541
Planning:
Buffers: shared hit=3
Planning Time: 0.058 ms
Execution Time: 16.357 ms
(8 rows )
```

With **the timing off option, the results of the** `explain` command match the results of pgbench tests and query executions without `explain` .

The buffers parameter does not affect the result, it illustrates that all table blocks were in the buffer cache (`shared hit =541`) when executing all three commands.

explain command has a **settings** option that will output a string with the values of parameters that:

a) affect the scheduler

b) have values other than the default values.

If such parameters are missing, the settings parameter has no effect on the output of the explain command.

The features of time measurement, time source settings, and measuring its speed will be discussed in the following chapters of the course.

What parameters "affect the scheduler"? This is declared when adding a configuration parameter by PostgreSQL developers or extension library creators. Example :

```
postgres=# select pg_settings_get_flags('work_mem');
pg_settings_get_flags
-----------------------
{EXPLAIN}
(1 row)
```

If the results contain , it means that the configuration parameter is declared as "affecting the scheduler". There are about 60 such parameters.

Configuration parameters can have the following flags:

`EXPLAI N` (approximately 60 parameters) parameters with this flag are included in EXPLAIN (SETTINGS) commands;

`NO_SHOW_AL L` (there are no parameters with this flag) are excluded from SHOW ALL commands;

`NO_RESET` (3 parameters) do not support RESET commands;

`NO_RESET_AL L` (3 parameters) are excluded from RESET ALL commands;

`NOT _ IN _ SAMPLE` (50 parameters) are not automatically included in the postgresql . conf . sample file from which the `initdb utility` creates the cluster parameter file

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

RUNTIME_COMPUTED (5 parameters) parameter values are not set, but are calculated when the instance starts.

The list of parameters with flags can be obtained using the command:

```
postgres=# select name, pg_settings_get_flags(name) from pg_settings where
array_length(pg_settings_get_flags(name),1)>0;
```

List calculated parameters :

```
postgres=# select name, context, pg_settings_get_flags(name) from pg_settings
where pg_settings_get_flags(name)::text like '%RUNTIME%';
name | context | pg_settings_get_flags
---------------------------------+----------+-------------------------------
data_checksums | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
data_directory_mode | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
shared_memory_size | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
shared_memory_size_in_huge_pages | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
wal_segment_size | internal | {NOT_IN_SAMPLE,RUNTIME_COMPUTED}
(5 rows )
```

List of parameters that are not reset to their original value:

```
postgres=# select name, context, pg_settings_get_flags(name) from pg_settings
where pg_settings_get_flags(name)::text like '%RESET%';
name | context | pg_settings_get_flags
------------------------+---------+-------------------------------------
transaction_deferrable | user | {NO_RESET,NO_RESET_ALL,NOT_IN_SAMPLE}
transaction_isolation | user | {NO_RESET,NO_RESET_ALL,NOT_IN_SAMPLE}
transaction_read_only | user | {NO_RESET,NO_RESET_ALL,NOT_IN_SAMPLE}
(3 rows)
```

9) Run the command generated from the plpgsql test : https://gist.github.com/lukaseder/2611212b23ba40d5f828c69b79214a0e/
used in the article https :// blog . jooq . org / whats - faster - count - or - count 1/ :

```
DO $$
DECLARE
v_ts TIMESTAMP;
v_repeat CONSTANT INT := 100;
rec RECORD;
run INT[];
stmt INT[];
elapsed DECIMAL[];
min_elapsed DECIMAL;
i INT := 1;
BEGIN
FOR r IN 1..5 LOOP
v_ts := clock_timestamp();
FOR i IN 1..v_repeat LOOP
FOR rec IN (
        select count(*) from t
) LOOP
NULL;
END LOOP;
END LOOP;
run[i] := r;
stmt[i] := 1;
elapsed[i] := (EXTRACT(EPOCH FROM CAST(clock_timestamp() AS TIMESTAMP)) -
EXTRACT(EPOCH FROM v_ts));
i := i + 1;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
  v_ts := clock_timestamp();
  FOR i IN 1..v_repeat LOOP
  FOR rec IN (
          select count(1) from t
  ) LOOP
  NULL;
  END LOOP;
  END LOOP;
  run[i] := r;
  stmt[i] := 2;
  elapsed[i] := (EXTRACT(EPOCH FROM CAST(clock_timestamp() AS TIMESTAMP)) -
EXTRACT(EPOCH FROM v_ts));
  i := i + 1;
  v_ts := clock_timestamp();
  FOR i IN 1..v_repeat LOOP
  FOR rec IN (
          select count(pk) from t
  ) LOOP
  NULL;
  END LOOP;
  END LOOP;
  run[i] := r;
  stmt[i] := 3;
  elapsed[i] := (EXTRACT(EPOCH FROM CAST(clock_timestamp() AS TIMESTAMP)) -
EXTRACT(EPOCH FROM v_ts));
  i := i + 1;
  END LOOP;
  SELECT min(t.elapsed)
  INTO min_elapsed
  FROM unnest(elapsed) AS t(elapsed);
  FOR i IN 1..array_length(run, 1) LOOP
  RAISE INFO 'RUN %, Statement %: %', run[i], stmt[i], CAST(elapsed[i] /
min_elapsed AS DECIMAL(10, 5));
  END LOOP;
  END$$;

  INFO: RUN 1, Statement 1: 1.00764
  INFO: RUN 1, Statement 2: 1.12819
  INFO: RUN 1, Statement 3: 1.38278
  INFO: RUN 2, Statement 1: 1.01305
  INFO: RUN 2, Statement 2: 1.12725
  INFO: RUN 2, Statement 3: 1.37496
  INFO: RUN 3, Statement 1: 1.00552
  INFO: RUN 3, Statement 2: 1.13360
  INFO: RUN 3, Statement 3: 1.38406
  INFO: RUN 4, Statement 1: 1.00269
  INFO: RUN 4, Statement 2: 1.13924
  INFO: RUN 4, Statement 3: 1.38148
  INFO: RUN 5, Statement 1: 1.00000
  INFO: RUN 5, Statement 2: 1.14320
  INFO: RUN 5, Statement 3: 1.41113
  DO
```

The result matches the results of the pgbench test.

This was an example of using a plpgsql program to test the execution time of three commands. Using pgbench is simpler and gives more information: the execution time of commands (latency) and the standard deviation of execution time.

Part 3. Using the sysbench utility

1) Create the tables that sysbench uses by running the command:

```
postgres@tantor:~$ sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --pgsql-password=postgres --tables=1 --table_size=100000 /usr/share/sysbench/oltp_read_only.lua prepare
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Creating table 'sbtest1'...
Inserting 100000 records into 'sbtest1'
Creating a secondary index on 'sbtest1'...
```

One table sbtest1 with 100 thousand rows was created.

Sysbench test files are written in lua . The `/ usr / share / sysbench /` directory contains standard tests. `Run` a `read - only test` named `oltp_read_only.lua :`

```
postgres@tantor:~$ sysbench --db-driver=pgsql --pgsql-port=5432 --pgsql-db=postgres --pgsql-user=postgres --pgsql-password=postgres --threads=10 --time=15 --report-interval=5 /usr/share/sysbench/oltp_read_only.lua run
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with the following options:
Number of threads: 10
Report intermediate results every 5 second(s)
Initializing random number generator from current time

Initializing worker threads...
Threads started!

[ 5s ] thds: 10 tps: 584.03 qps: 9364.62 (r/w/o: 8194.56/0.00/1170.05) lat (ms,95%): 28.67 err/s: 0.00 reconn/s: 0.00
[ 10s ] thds: 10 tps: 592.77 qps: 9480.12 (r/w/o: 8294.58/0.00/1185.54) lat (ms,95%): 28.16 err/s: 0.00 reconn/s: 0.00
[ 15s ] thds: 10 tps: 477.21 qps: 7632.70 (r/w/o: 6678.48/0.00/954.21) lat (ms,95%): 51.02 err/s: 0.00 reconn/s: 0.00
SQL statistics:
queries performed:
read: 115934
write: 0
other: 16562
total: 132496
 transactions:                        8281    (550.22 per sec.)
queries: 132496 (8803.46 per sec.)
ignored errors: 0 (0.00 per sec.)
reconnects: 0 (0.00 per sec.)

General statistics:
total time: 15.0492s
total number of events:               8281

Latency (ms):
min: 3.70
avg: 18.12
max: 178.98
95th percentile: 30.81
sum: 150092.73

Threads fairness:
events (avg/stddev): 828.1000/63.26
execution time (avg/stddev): 15.0093/0.01
```

Number of transactions per second `550.22` .

The other indicators are not convenient. The number of transactions `8281` and **`events :`** depend on the duration of the test. Other indicators are difficult to analyze. For example, `Latency 95th percentile` more closely matches reality than `avg` , which is hardly meaningful, since measurements

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

that deviate greatly from the average (`178.98`) should have been excluded from the calculation . Delays can be related to the scheduler or operating system activity. In the example, 10 threads were used, the virtual machine has fewer cores and delays are related to the operating system scheduler. The authors of the utility should have excluded measurements taken immediately after starting the utility from the calculation.

Part 4. Using HammerDB

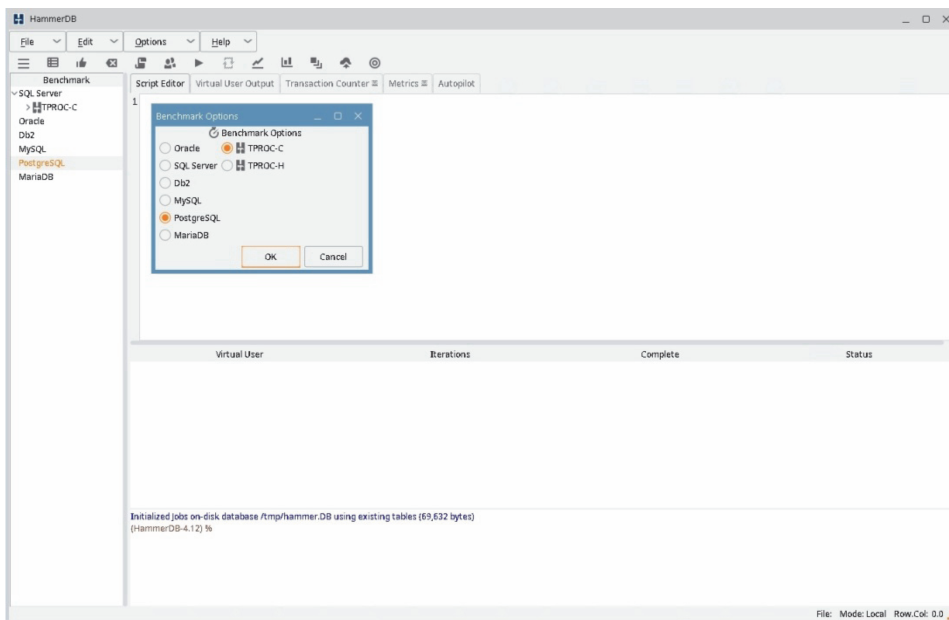
1) Launch terminal as user astra and install HammerDB:

```
astra @tantor:~$ wget https://github.com/TPC-
Council/HammerDB/releases/download/v4.12/HammerDB-4.12-Linux-x64-installer.run
 - 'HammerDB-4.12-Linux-x64-installer.run' saved [12665959/12665959]
astra@tantor:~$ chmod +x HammerDB-4.12-Linux-x64-installer.run
astra@tantor:~$ ./HammerDB-4.12-Linux-x64-installer.run
```

Install to default directory /home/astra/HammerDB-4.12
After installation, the utility will start. The utility can be started again with the command:

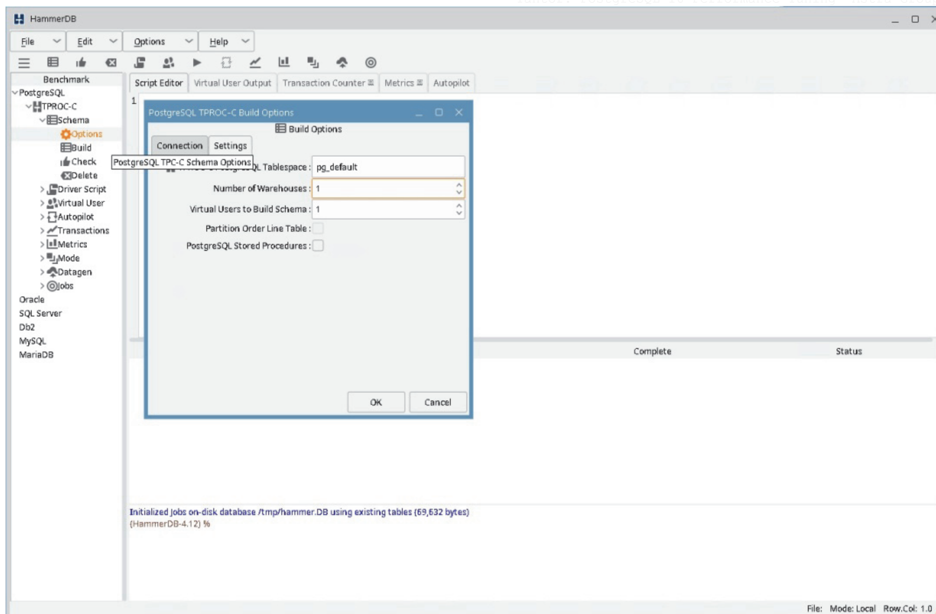
```
astra@tantor:~$ cd /home/astra/HammerDB-4.12
astra @ tantor :~/ HammerDB -4.12$ ./ hammerdb &
```

2) After launching the utility, click on PostgreSQL in its window . In the window that appears, " Benchmark Options " leave the values PostgreSQL and TPROC - C . Click OK in the window. In the confirmation window, also click OK.

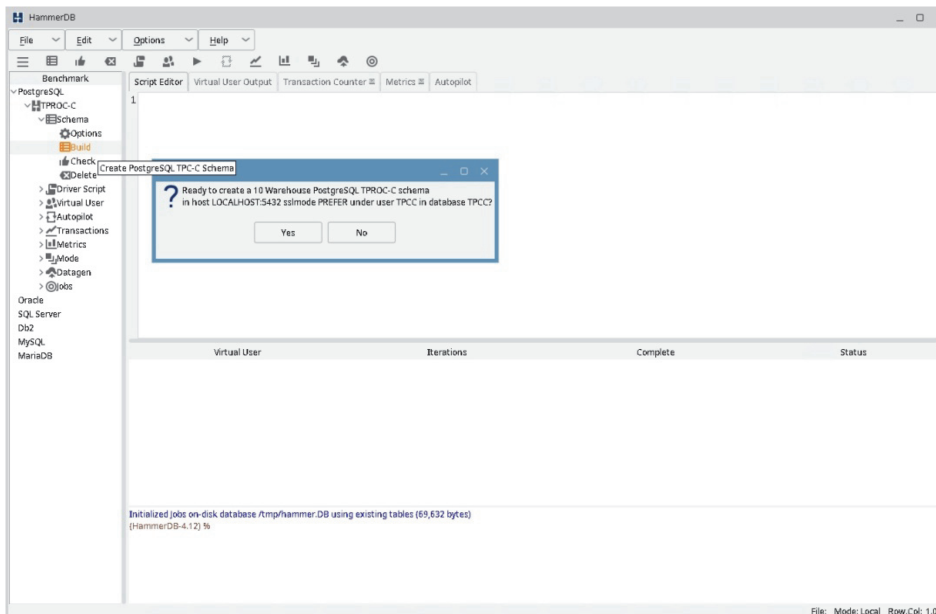


3) Open the menu PostgreSQL->TPROC-C->Schema and click on Options, go to the Settings tab and set Number of warehouses = 2. The default value is 1. For 10 warehouses, 1GB is required. Set the number of Virtual Users to build schema to twice the number of processor cores minus one, for example 7 for 4 cores. Click OK, the setting will be saved. In real testing, a significantly larger number of warehouses will be required. However, a small number of warehouses will allow simulating contention for locks between sessions.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru



on Build in the PostgreSQL -> TPROC - C -> Schema menu . A window will appear warning that the tpcc database will be created and tables with data for the TPC-C test will be created in it.



The creation time is several minutes. You can see the load on the processor while the tables are being created using the top command. To show the load on the cores, press 1. To exit the top utility, press the q key.

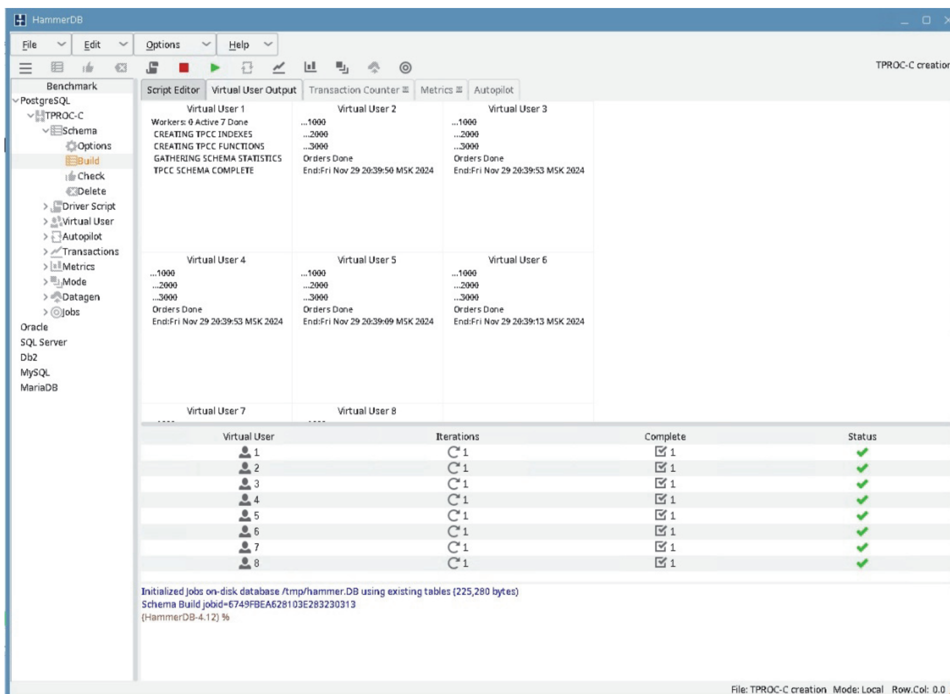
```
astra@tantor:~/HammerDB-4.12$ top
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

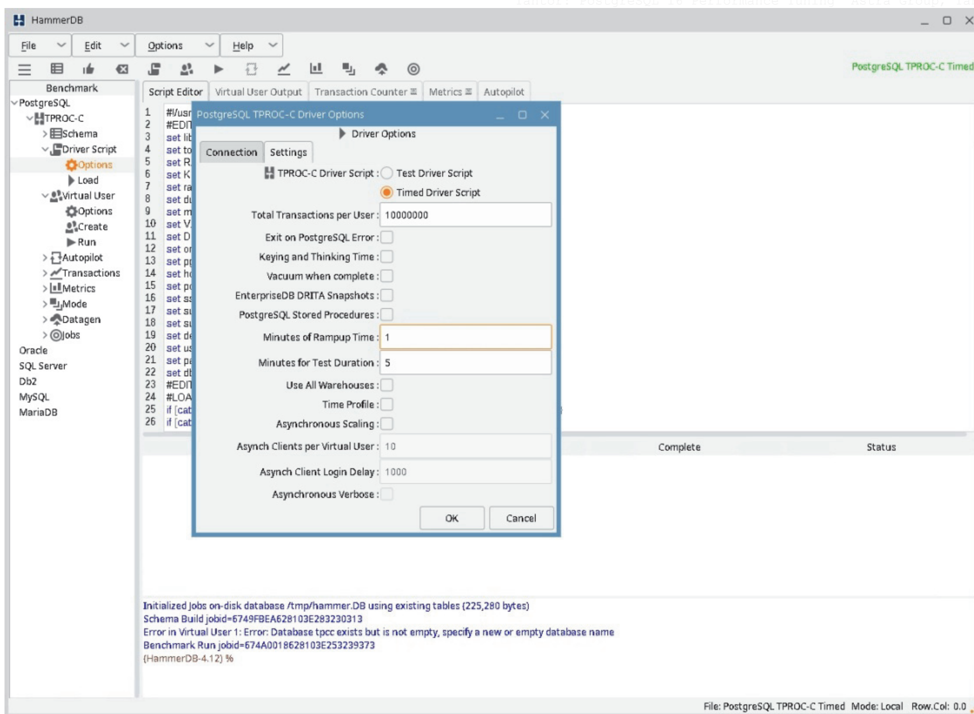

```
top - 20:38:24 up 8 days,  1:25,  3 users,  load average: 3.55, 1.82, 1.07
Tasks: 207 total,   2 running, 205 sleeping,   0 stopped,   0 zombie
%Cpu0  : 89.9 us,  7.7 sy,  0.0 ni,  1.0 id,  1.0 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu1  : 93.3 us,  5.7 sy,  0.0 ni,  0.0 id,  0.7 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu2  : 90.4 us,  4.6 sy,  0.0 ni,  1.7 id,  1.0 wa,  0.0 hi,  2.3 si,  0.0 st
%Cpu3  : 95.7 us,  4.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
MiB Mem :   3926.6 total,    919.1 free,   1342.3 used,   1995.7 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2584.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 290408 astra     20   0  672620 115596  22860 S 356.4   2.9   2:03.44 wish8.6
 290426 postgres  20   0  233720  13780  10112 S   4.6   0.3   0:01.30 postgres
 290432 postgres  20   0  233720  13780  10112 S   4.3   0.3   0:00.99 postgres
 290427 postgres  20   0  233720  13908  10112 S   3.6   0.3   0:01.17 postgres
 290428 postgres  20   0  233720  13780  10112 R   3.3   0.3   0:01.19 postgres
 290430 postgres  20   0  233740  13908  10112 S   3.3   0.3   0:01.12 postgres
 290429 postgres  20   0  233732  13908  10112 S   2.6   0.3   0:01.15 postgres
 290431 postgres  20   0  233740  14036  10112 S   2.6   0.3   0:01.10 postgres
 288832 root      20   0       0      0      0 I   2.3   0.0   0:00.21 kworker/u11:0-flush-8:0
   5620 fly-dm    -2   0  452012 158476  21756 S   1.3   3.9   4:50.41 Xorg
     11 root      20   0       0      0      0 I   1.0   0.0   0:03.22 kworker/u8:0-ext4-rsv-conversion
 234872 postgres  20   0  231448   7760   5120 S   1.0   0.2   0:07.91 postgres
   5904 astra     20   0  257.3g  77476  46188 S   0.7   1.9 180:58.14 fly-start-menu
   5947 astra     20   0   18920   6784   5504 S   0.7   0.2   1:39.10 compton
 234877 postgres  20   0  232028   9552   6528 S   0.7   0.2  34:46.03 postgres
     10 root       0 -20       0      0      0 I   0.3   0.0   0:01.72 kworker/0:0H-kblockd
     16 root      20   0       0      0      0 I   0.3   0.0   1:47.11 rcu_preempt
    136 root       0 -20       0      0      0 I   0.3   0.0   0:12.26 kworker/2:1H-kblockd
    452 root      20   0  239668   8116   6964 S   0.3   0.2   0:25.27 accounts-daemon
   1855 root      20   0 1404656  56632  33008 S   0.3   1.4  13:42.11 syslog-ng
   5834 astra     20   0  151264   2568   2304 S   0.3   0.1  87:30.34 VBoxClient
   5891 astra     20   0  502428  40576  34048 S   0.3   1.0   1:40.80 org_kde_powerde
 234874 postgres  20   0  231312   8016   5376 S   0.3   0.2   0:09.01 postgres
 283231 astra     20   0  616644  81828  65656 S   0.3   2.0   0:01.99 fly-term
 290437 astra     20   0   14196   5632   3456 R   0.3   0.1   0:00.05 top
      1 root      20   0  172812  15116  10124 S   0.0   0.4   0:10.18 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.14 kthreadd
      3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_workqueue_release
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-rcu_g
```

In the top utility window , you can see that the load on four cores is close to 100%. USR/SYS ratio: `% Cpu 0 89.9us + 7.7 sy .`

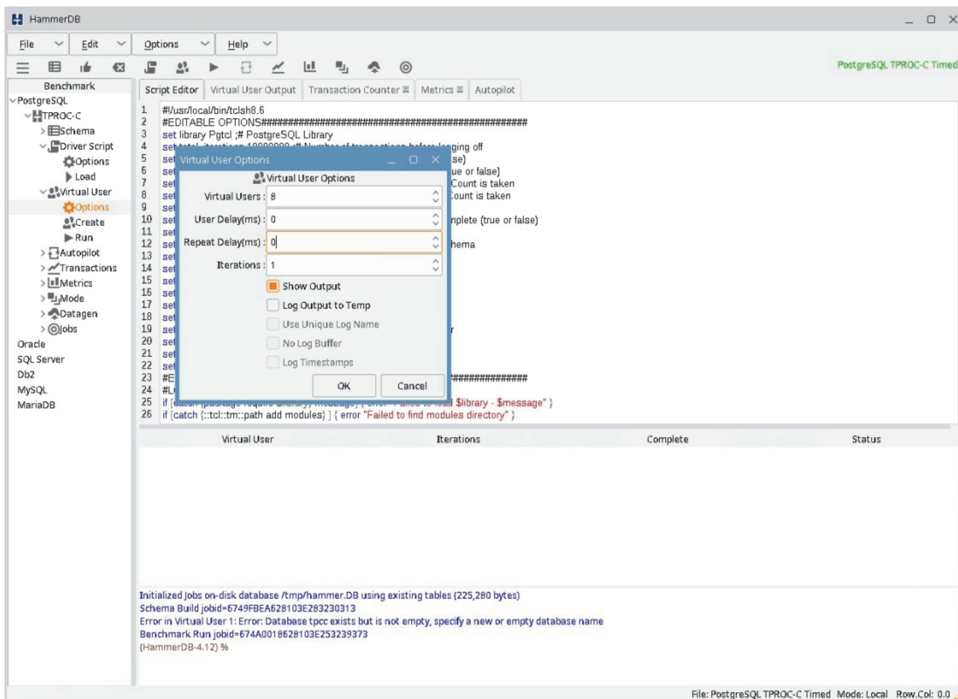
TPCC message will indicate that the tables have been created. SCHEMA COMPLETE in the Virtual window User 1.



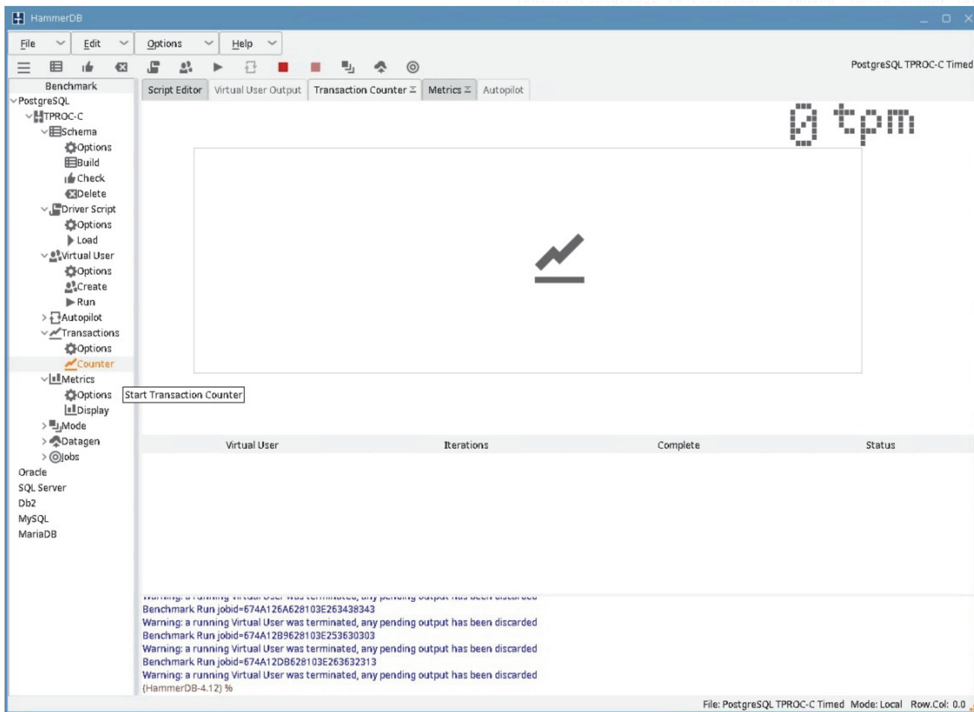
5) Click on the red square on the utility toolbar. Select V menu PostgreSQL->TPROC-C-> Driver Script -> Options. In the window that opens, go to the Settings tab and specify Minutes of Rampup Time 0. This is the "warm-up" time, that is, the gradual increase in the load on the DBMS. We have installed a small number of clients that will not load the DBMS and the delay is not needed. Minutes of Test Duration 5, this is the duration of the test if the test is not stopped earlier.



6) B In the PostgreSQL->TPROC-C-> Virtual User -> Options window , set User Delay(ms)=0, Repeat Delay(ms)=0. For real testing, it is recommended to set the number of users 10 times less than the number of warehouses.

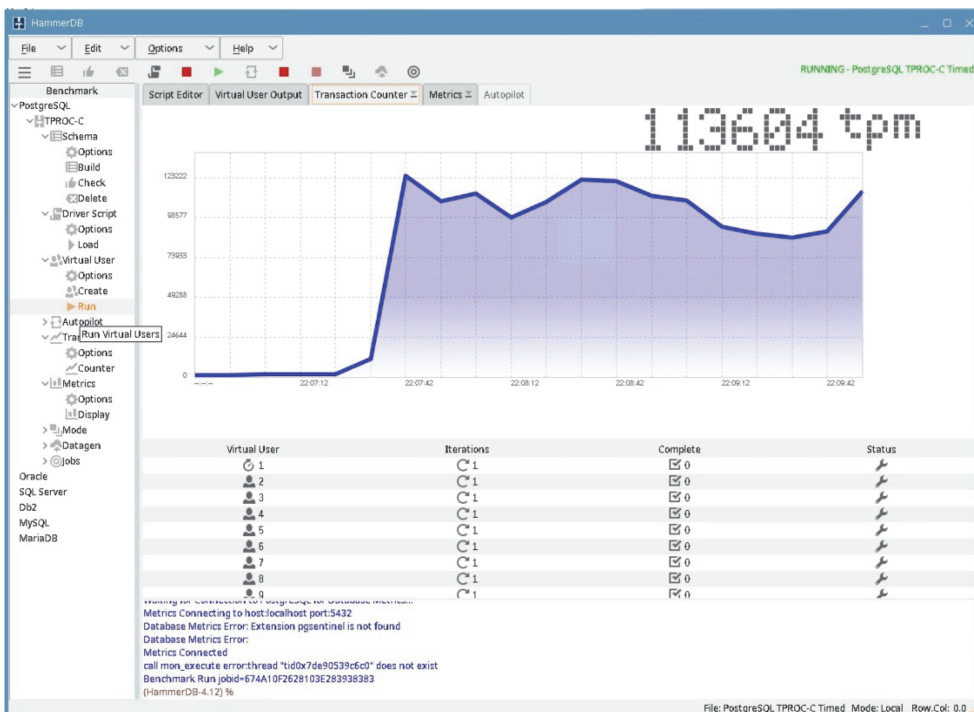


7) Before running the test, you need to start the Transaction Counter to monitor tpm. To do this, click on PostgreSQL -> TPROC - C -> Transactions -> Counter in the menu. The transaction counter will start.



7) Click on PostgreSQL->TPROC-C->Virtual User -> Create and then on PostgreSQL->TPROC-C->Virtual User -> Run

Click on the Transaction Counter tab. A window will appear with a test metric called tpm .



In the example tpm=113604. The tpm metric graph fluctuates, this is not an inaccuracy, but a consequence of the dynamic load corresponding to the test rules. The decrease in tpm by tens of percent occurs due to the retention of the database horizon by autoanalysis. A few minutes after starting the test, tpm can decrease by a `third` . This occurs due to the launch of autoanalysis on the `order_line table` and lasting up to two minutes . The tpm reduction is also affected by the execution of the checkpoint.

This is an example where periodic statistics collection is not only useless, but harmful. If the statistics data does not change after recollections, there is no point in "updating" the statistics.

```
postgres =# \c tpcc
```

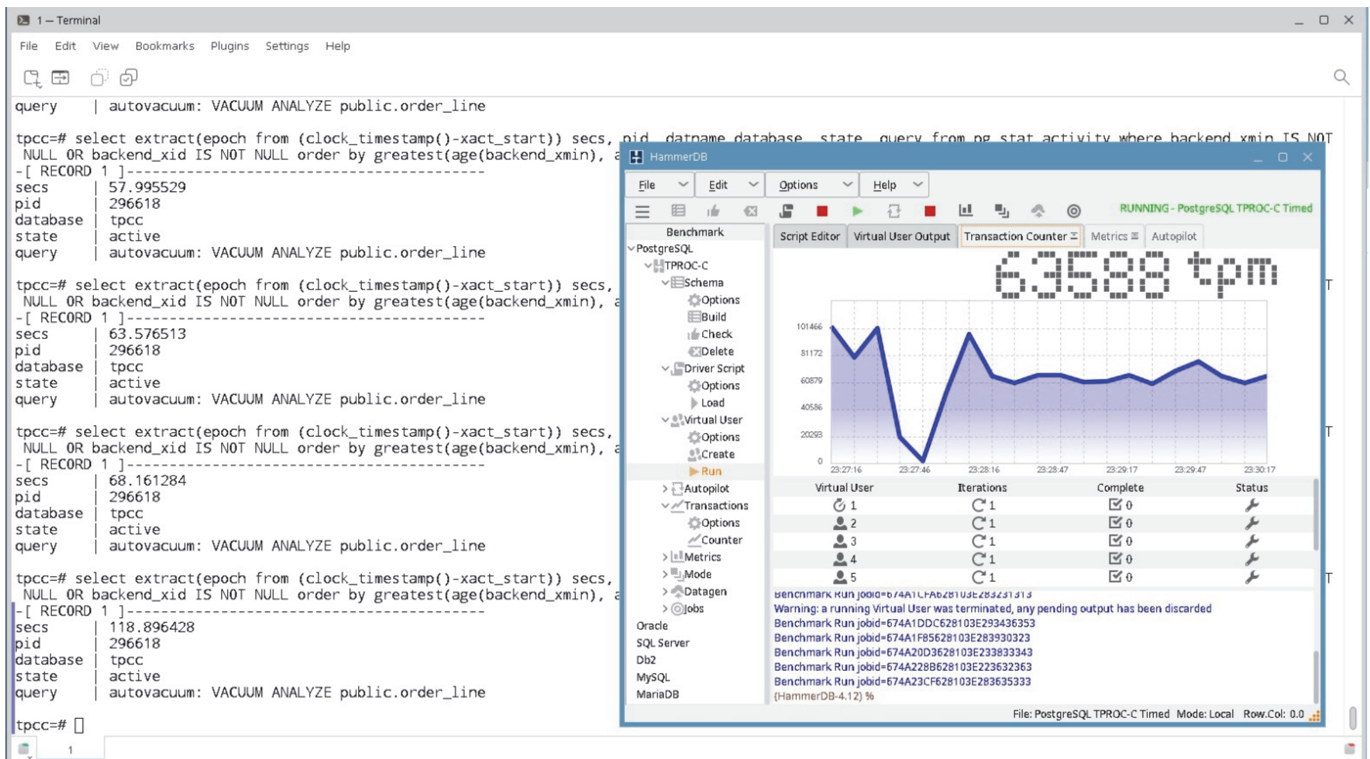
"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
You are now connected to database "tpcc" as user "postgres".
tpcc=# select extract(epoch from (clock_timestamp()-xact_start)) secs, pid,
datname database, state, query from pg_stat_activity where backend_xmin IS NOT
NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc limit 1 \gx
-[ RECORD 1 ]----------------------------------------
secs | 118 .896428
pid | 296618
database | tpcc
state | active
query | autovacuum: VACUUM ANALYZE public.order_line
```

In the absence of load, table analysis takes (on a DBMS with default configuration parameters) 27 seconds :

```
tpcc=# VACUUM (ANALYZE, verbose) public.order_line;
INFO: vacuuming "tpcc.public.order_line"
INFO: finished vacuuming "tpcc.public.order_line": index scans: 1
pages: 0 removed, 186118 remain, 166506 scanned (89.46% of total)
tuples: 446332 removed, 15377120 remain, 0 are dead but not yet removable, oldest xmin: 10827114
removable cutoff: 10827114, which was 0 XIDs old when operation ended
frozen: 79603 pages from table (42.77% of total) had 1825159 tuples frozen
index scan needed: 51544 pages from table (27.69% of total) had 597993 dead item identifiers removed
index "order_line_i1": pages: 81640 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 83.907 MB/s, avg write rate: 48.399 MB/s
buffer usage: 170193 hits, 296131 misses, 170812 dirtied
WAL usage: 384641 records, 164101 full page images, 1262224255 bytes
system usage: CPU: user: 6.17 s, system: 4.05 s, elapsed: 27.57 s
INFO: analyzing "public.order_line"
INFO: "order_line": scanned 30000 of 186118 pages, containing 2492677 live rows and 0 dead rows; 30000 rows
in sample, 15464402 estimated total rows
VACUUM
```

tpm dropped from 101000 to 63588 due to autoanalysis holding the database horizon.

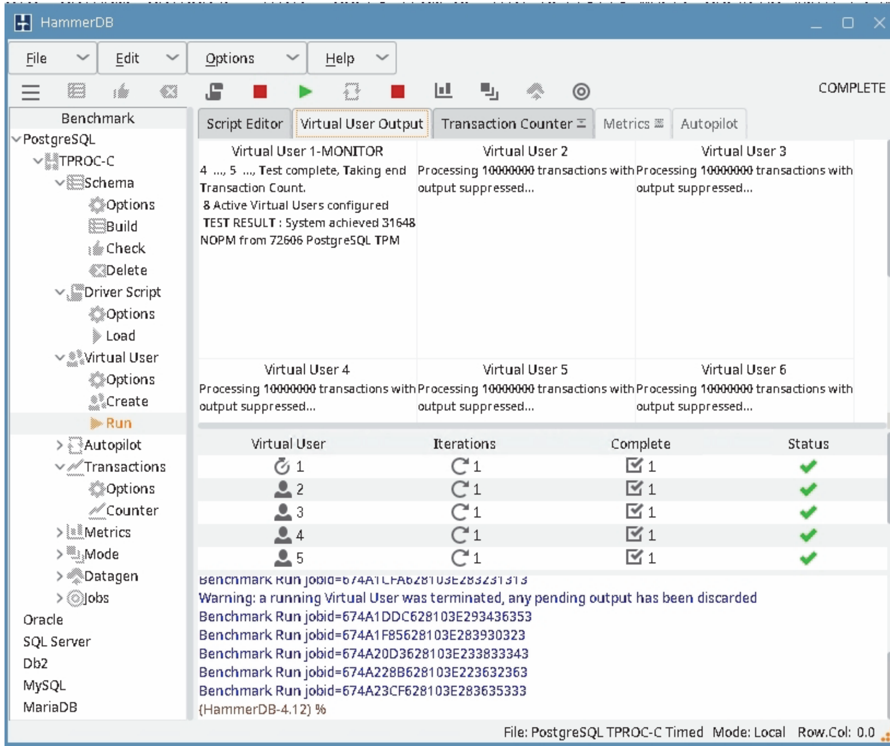


8) You don’t need to wait for the test to stop by time. Stop the test by clicking on the left red square (Destroy Virtual users) on the toolbar. On the right red square (Stop Transaction Counter) you don't need to press, otherwise the transaction counter will stop. In the PostgreSQL->TPROC-C-> Virtual User -> Options window , set User Delay(ms)=500, Repeat Delay(ms)=500. These parameters do not affect tpm much , they affect the fluctuations of the tpm graph . Tpm is affected by the number of clients (sessions with the DBMS), called Virtual Users.

Changing parameters in Virtual User -> Options , click on Virtual User -> Run and going to the Transaction Counter tab you can see how tpm changes. The maximum tpm is achieved when the number of clients is equal to the number of processor cores.

9) After testing is complete, close the HammerDB window and delete the tpcc database , which will grow several times after a short period of testing. The test puts rows in the tables and they grow.

After the test is completed, in the Virtual window User 1 will be shown the test result:



IN example result tpms (PostgreSQL TPM)=72606 and NOPM (New Orders per minute)=31648.

Over time, tpcm will decrease with repeated tests. Because of this, the test is not suitable for use in checking how cluster configuration changes will affect performance, since tpm is not only unstable, but also decreases over time. For the stability of the result, you will have to delete the tpcc database and create it again. The test can be used for long-term one-time testing for comparison with other DBMS of the same or different type. When testing, all parameters must be the same, especially the size of the database (number of warehouses), test duration, number of clients.

The TPC-C test is convenient because when it is run, you can analyze the operation of the instance using available tools (extensions, accessing views with statistics) and identify the most effective tools and metrics. This is a consequence of more complex commands than simple pgbench load tests TPC - B .

The HammerDB graphical application is convenient for such purposes.

```
postgres=# \c tpcc
You are now connected to database "tpcc" as user "postgres".
tpcc=# select pg_size_pretty(pg_database_size('tpcc'));
pg_size_pretty
----------------
 2682 MB
(1 row)
```

After creating the tables, the database took up 1GB.

```
tpcc=# vacuum full;
VACUUM
tpcc=# select pg_size_pretty(pg_database_size('tpcc'));
pg_size_pretty
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
----------------
2331 MB
(1 row)

postgres=# drop database tpcc;
DROP DATABASE
postgres=#
```

Part 5. Using the Go App - TPC

1) Install go - tpc :

```
postgres @ tantor :~$ mkdir gotpc
mkdir: cannot create directory 'gotpc': File exists
postgres@tantor:~$ cd gotpc
postgres@tantor:~/gotpc$ wget https://raw.githubusercontent.com/pingcap/go-tpc/master/install.sh
- 'install.sh' saved [2020/2020]
postgres@tantor:~/gotpc$ chmod +x install.sh
postgres@tantor:~/gotpc$ ./install.sh
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
0 0 0 0 0 0 0 0 --:--:-- --:--:-- --:--:-- 0
100 4984k 100 4984k 0 0 4370k 0 0:00:01 0:00:01 --:--:-- 23.2M
Detected shell: bash
Shell profile: /var/lib/postgresql/.bash_profile
/var/lib/postgresql/.bash_profile has been modified to to add go-tpc to PATH
open a new terminal or source /var/lib/postgresql/.bash_profile to use it
Installed path: /var/lib/postgresql/.go-tpc/bin/go-tpc
==============================================================
Have a try: go-tpc tpcc
==============================================================
postgres @ tantor :~/ gotpc $
```

2) Go to the utility directory and run the utility with the `tpcc option` **`prepare , which will create the`** gotpcc database and will create objects for the **`TPC-C`** type test in this database :

```
postgres@tantor:~/gotpc$ cd $HOME/.go-tpc/bin
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpcc prepare -d postgres -U postgres -p 'postgres' -D gotpcc -H
127.0.0.1 -P 5432 --conn-params sslmode=disable
maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
creating table warehouse
creating a district table
creating customer table
creating index idx_customer
creating table history
creating index idx_h_w_id
creating index idx_h_c_w_id
creating table new_order
creating table orders
creating index idx_order
creating table order_line
creating stock table
creating table item
load to item
load to warehouse in warehouse 1
Load stock in warehouse 1
load to district in warehouse 1
load to warehouse in warehouse 2
Loading stock in warehouse 2
load to district in warehouse 2
load to warehouse in warehouse 3
Load stock in warehouse 3
load to district in warehouse 3
load to warehouse in warehouse 4
Loading stock in warehouse 4
load to district in warehouse 4
load to warehouse in warehouse 5
Load stock in warehouse 5
load to district in warehouse 5
load to warehouse in warehouse 6
Load stock in warehouse 6
load to district in warehouse 6
load to warehouse in warehouse 7
Loading stock in warehouse 7
load to district in warehouse 7
load to warehouse in warehouse 8
Load stock in warehouse 8
```

```
load to district in warehouse 8
load to warehouse in warehouse 9
Load stock in warehouse 9
load to district in warehouse 9
load to warehouse in warehouse 10
Load stock in warehouse 10
load to district in warehouse 10
load to customer in warehouse 1 district 1
load to history in warehouse 1 district 1
..
```

For each test type of this utility, it is better to create tables in a separate database, since some tables have overlapping names. There are no local connections in the utility driver, the connection must be via a network interface, so all connection parameters are specified .

By default, the utility creates 10 warehouses. They are created within 4 minutes. The number of warehouses can be changed by specifying the parameter `--warehouses 4` .

3) After about 3-4 minutes you will see the lines " **begin to check** `warehouse 1 at condition 3.3.2.4` " in the utility log. The utility has started checking the data after loading. The check takes a long time and is pointless. It is needed for those DBMS that lose data, PostgreSQL does not lose data. Click on keyboard < ctrl+c > to interrupt check :

```
load to new_order in warehouse 10 district 10
load to order_line in warehouse 10 district 10
begin to check warehouse 1 at condition 3.3.2.4
begin to check warehouse 1 at condition 3.3.2.5
begin to check warehouse 1 at condition 3.3.2.6
begin to check warehouse 1 at condition 3.3.2.7
begin to check warehouse 1 at condition 3.3.2.8
begin to check warehouse 1 at condition 3.3.2.9
begin to check warehouse 1 at condition 3.3.2.10
begin to check warehouse 1 at condition 3.3.2.10
^C
Got signal [interrupt] to exit.
check prepare failed, err check warehouse 1 at condition 3.3.2.10 failed exec SELECT count(*)
FROM ( SELECT c.c_id, c.c_d_id, c.c_w_id, c.c_balance c1,
(SELECT sum(ol_amount) FROM orders, order_line
WHERE OL_W_ID=O_W_ID
AND OL_D_ID = O_D_ID
AND OL_O_ID = O_ID
AND OL_DELIVERY_D IS NOT NULL
AND O_W_ID=?
AND O_D_ID=c.C_D_ID
AND O_C_ID=c.C_ID) sm, (SELECT sum(h_amount) from history
WHERE H_C_W_ID=?
AND H_C_D_ID=c.C_D_ID
AND H_C_ID=c.C_ID) smh
FROM customer c
WHERE c.c_w_id = ? ) t
WHERE c1<>sm-smh failed pq: canceling statement due to user request
Finished
```

The size of the database created by the utility is 1GB :

```
postgres=# select pg_size_pretty(pg_database_size('gotpcc'));
pg_size_pretty
----------------
 1028 MB
(1 row)
```

4) Launch tpcc test :

```
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpcc run -d postgres -U postgres -p
'postgres' -D gotpcc -H 127.0.0.1 -P 5432 --conn-params sslmode=disable
maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
[Current] DELIVERY - Takes(s): 9.9, Count: 61, TPM: 369.4, Sum(ms): 879.3, Avg(ms): 14.4, 50th(ms): 14.2,
90th(ms): 16.8, 95th(ms): 17.8, 99th(ms): 21.0,
99.9th(ms): 27.3, Max(ms): 27.3 [Current] NEW_ORDER - Takes(s): 9.9, Count: 714, TPM: 4310.5, Sum(ms): 5931.1,
Avg(ms): 8.3, 50th(ms): 8.4, 90th(ms): 11.0, 95th(ms): 12.6, 99th(ms): 17.8, 99.9th(ms): 26.2, Max(ms): 29.4
[Current] ORDER_STATUS - Takes(s): 9.9, Count: 61, TPM: 369.7, Sum(ms): 104.9, Avg(ms): 1.7, 50th(ms): 2.1,
90th(ms): 2.1, 95th(ms): 2.6, 99th(ms): 3.1, 99.9th(ms): 5.2, Max(ms): 5.2 [Current] PAYMENT - Takes(s): 10.0,
Count: 697, TPM: 4197.6, Sum(ms): 2448.8, Avg(ms): 3.5, 50th(ms): 3.7, 90th(ms): 4.2, 95th(ms): 4.7, 99th(ms):
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
7.9, 99.9th(ms): 8.4, Max(ms): 13.1 [Current] STOCK_LEVEL - Takes(s): 9.9, Count: 65, TPM: 395.9, Sum(ms): 277.1, Avg(ms): 4.3, 50th(ms): 3.7, 90th(ms): 4.7, 95th(ms): 5.8, 99th(ms): 24.1, 99.9th(ms): 41.9, Max(ms): 41.9
```

Current TPM statistics will be displayed on the console.

5) After about a minute, press < ctrl + c > on the keyboard to interrupt the test. The utility will display statistics with the `tpmC result` :

```
^ C
Got signal [ interrupt ] to exit .
Finished
[Summary] DELIVERY - Takes(s): 182.8, Count: 1329, TPM: 436.3, Sum(ms): 17163.3, Avg(ms): 12.9, 50th(ms): 12.6, 90th(ms): 14.7, 95th(ms): 16.8, 99th(ms): 22.0, 99.9th(ms): 29.4, Max(ms): 30.4
[Summary] NEW_ORDER - Takes(s): 182.8, Count: 14194, TPM: 4659.0, Sum(ms): 106451.6, Avg(ms): 7.5, 50th(ms): 7.3, 90th(ms): 9.4, 95th(ms): 10.5, 99th(ms): 15.7, 99.9th(ms): 22.0, Max(ms): 41.9
[Summary] ORDER_STATUS - Takes(s): 182.8, Count: 1288, TPM: 422.9, Sum(ms): 2306.0, Avg(ms): 1.8, 50th(ms): 2.1, 90th(ms): 2.6, 95th(ms): 2.6, 99th(ms): 4.2, 99.9th(ms): 8.9, Max(ms): 10.0
[Summary] PAYMENT - Takes(s): 182.8, Count: 13431, TPM: 4407.9, Sum(ms): 46553.2, Avg(ms): 3.5, 50th(ms): 3.7, 90th(ms): 4.2, 95th(ms): 4.7, 99th(ms): 7.9, 99.9th(ms): 14.2, Max(ms): 19.9
[Summary] STOCK_LEVEL - Takes(s): 182.7, Count: 1262, TPM: 414.4, Sum(ms): 3707.7, Avg(ms): 2.9, 50th(ms): 3.1, 90th(ms): 3.7, 95th(ms): 4.2, 99th(ms): 8.9, 99.9th(ms): 24.1, Max(ms): 41.9
tpmC: 4659.0 , tpmTotal: 10340.5, efficiency: 3622.8%
```

6) Delete the database created by the utility:

```
postgres@tantor:~/.go-tpc/bin$ psql - with "drop database gotpcc;"
DROP DATABASE
```

7) Unlike HammerDB, the go-tpc utility does not work at full speed, but follows the TPC-C test rules, working with delays and gradually increasing the load. The utility has two more tests: TPC-H and the tpc utility's own test - ch (CH-benCHmark mixed load from TPC-C and TPC-H tests). The test is suitable for DBMSs that are used for general purposes with mixed load. PostgreSQL is rarely used with mixed load, since physical replicas are used for long (analytical) queries.

The TPC-H benchmark is for data warehouses and consists of read commands only. The TPC-H benchmark result is more stable, allowing for comparison. The downside is that the size of the official benchmark databases is quite large.

Database creation takes approximately 10 minutes. Database size is 1660MB.

The following steps of this practice are optional. The commands and their results are provided for reference only. You can use these commands as a reference or if you want to consolidate your skills in using the utility.

```
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpch prepare -d postgres -U postgres -p 'postgres' -D gotpch -H 127.0.0.1 -P 5432 --conn-params sslmode=disable
maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
creating a nation
creating region
creating part
creating supplier
creating parts supp
creating a customer
creating orders
creating lineitem
generating nation table
generate nation table done
generating region table
generate region table done
generating customers table
generate customers table done
generating suppliers table
generate suppliers table done
generating part/partsupplier tables
generate part/partsupplier tables done
generating orders/lineitem tables
generate orders/lineitem tables done
Finished
```

Launch TPC-H test :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres@tantor:~/.go-tpc/bin$ ./go-tpc tpch run -d postgres -U postgres -p 'postgres' -D gotpch -H 127.0.0.1 -P 5432 --conn-params sslmode=disable
maxprocs: Leaving GOMAXPROCS=4: CPU quota undefined
[Current] Q1: 6.21s
[Current] Q2: 0.91s
[Current] Q3: 1.31s
[Current] Q4: 0.44s
[Current] Q5: 0.44s
[Current] Q10: 1.24s
[Current] Q11: 0.30s
[Current] Q12: 1.17s
[Current] Q13: 1.04s
[Current] Q14: 1.17s
[Current] Q6: 0.77s
[Current] Q7: 0.64s
[Current] Q8: 0.97s
[Current] Q9: 3.05s
[Current] Q15: 0.84s
[Current] Q16: 0.57s
[Current] Q17: 1199.91s
[Current] Q18: 5.27s
[Current] Q19: 1.04s
```

The Q17 query in the test is long, it takes 3 hours (11999.91 s). The utility shows the maximum value of `1199.91s` for long queries, there is a software error in the utility.

Example monitoring query Q17:

```
postgres=# select extract(epoch from (clock_timestamp()-xact_start)) secs , pid, datname database, state, query from pg_stat_activity where backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin), age(backend_xid)) desc limit 1 \gx
-[ RECORD 1 ]----------------------------------------------
secs | 6434 .701066
pid | 316066
database | gotpch
state | active
query | +
| /*PLACEHOLDER*/ select +
| sum(l_extendedprice) / 7.0 as avg_yearly+
| from +
| lineitem, +
| part +
| where +
| p_partkey = l_partkey +
| and p_brand = 'Brand#43' +
| and p_container = 'LG PACK' +
| and l_quantity < ( +
| select +
| 0.2 * avg(l_quantity) +
| from +
| lineitem +
| where +
| l_partkey = p_partkey +
        | ); +
|
```

The query contains a correlated subquery. Correlated subqueries are heavy for all DBMS. For example , on specialized under OLAP DBMS clickhouse queries TPC-H Q17 and Q20 tests are not work "As of October 2024, **the query does not work out-of-the box due to correlated subqueries** . Corresponding issue: https://github.com/ClickHouse/ClickHouse/issues/6697" (https://clickhouse.com/docs/en/getting-started/example-datasets/tpch).

Query Q20 runs on PostgreSQL with default parameters for 7.6 hours, query Q17 runs for about 4 hours . Long queries are interesting for query execution optimization: finding out the reasons for long execution . It is worth moving on to query optimization after setting up the PostgreSQL instance. Plan execution this request :

```
postgres=# \c gotpch
You are now connected to database "gotpch" as user "postgres".
gotpch=# explain select sum(l_extendedprice) / 7.0 as avg_yearly from lineitem, part where p_partkey = l_partkey and p_brand = 'Brand#43' and p_container = 'LG PACK' and l_quantity < ( select 0.2 * avg(l_quantity) from lineitem where l_partkey = p_partkey);
QUERY PLAN
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
------------------------------------------------------------------------
Aggregate (cost=2204613.84..2204613.86 rows=1 width=32)
-> Hash Join (cost=6625.16..2204608.92 rows=1970 width=8)
Hash Cond: (lineitem.l_partkey = part.p_partkey)
Join Filter: (lineitem.l_quantity < ( SubPlan 1 ))
-> Seq Scan on lineitem (cost=0.00..184745.33 rows=6001233 width=21)
-> Hash (cost=6622.70..6622.70 rows=197 width=8)
-> Gather (cost=1000.00..6622.70 rows=197 width=8)
Workers Planned: 2
-> Parallel Seq Scan on part (cost=0.00..5603.00 rows=82 width=8)
Filter:((p_brand = 'Brand#43'::bpchar) AND (p_container = 'LG PACK'::bpchar))
      SubPlan 1
-> Aggregate (cost=199748.49..199748.51 rows=1 width=32)
-> Seq Scan on lineitem lineitem_1 (cost=0.00..199748.41 rows=31 width=5)
Filter: (l_partkey = part.p_partkey )
JIT:
Functions: 24
Options: Inlining true, Optimization true, Expressions true, Deforming true
(17 rows )
```

There are 6 million and 200 thousand rows in the tables, which is not so much:

```
gotpch=# select count(*) from lineitem;
count
---------
6001215
(1 row)

gotpch=# select count(*) from part;
count
--------
200000
(1 row )
```

A long query needs to be interrupted by typing < ctrl + c > on the keyboard and then deleting the **gotpch** database :

```
postgres@tantor:~/.go-tpc/bin$ psql - with "drop database gotpch;"
```

The go-tpc application contains three tests. To create a database for the third test (mixed), the " ch " parameter is used instead of " tpcc " and " tpch ":

```
go-tpc ch prepare -d postgres -U postgres -p 'postgres' -D gotpcch -H 127.0.0.1
-P 5432 --conn-params sslmode=disable
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Practice for Chapter 2

Part 1. Launching a Huge Pages Instance

1) Find the list of PostgreSQL instances:

```
postgres @ tantor :~$ ps - ef | grep / postgres
postgres 1423 1 0 Nov27 ? 00:00:06 /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data
postgres 1432 1 0 Nov27 ? 00:00:03 /usr/lib/postgresql/15/bin/postgres -D /var/lib/postgresql/15/main -c
config_file=/etc/postgresql/15/main/postgresql.conf
```

In the example there are two instances: Tantor DBMS and PostgreSQL Astralinux .

2) See how much memory the PostgreSQL instance is using and the size of huge pages:

```
postgres@tantor:~$ cat /proc/1432/status | grep VmPeak
VmPeak: 222268 kB
postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages: 2048 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
Hugetlb: 0 kB
```

Maximum memory consumption per instance: `222268 kB` / `2048 kB` = 109 pages **of 2 MB (2048**). Huge pages can be used by the shared pool (shared_buffers parameter **) and parallel** processes (**min_dynamic_shared_memory parameter) . Forks can also be used for other memory** structures. The peak memory consumption on a loaded instance shows the memory usage of all structures.

- zero value in `the AnonHugePages : line` indicates that some process has explicitly requested the use of THP via `the madvise () system call` . PostgreSQL does not use such a system call.

Run the free command:

```
postgres @ tantor :~$ free
              total used free shared buff/cache available
Mem : 4020796 1137792 764984 158692 2556828 2883004
Swap : 0 0 0
```

3) Stop the instance and check how many hugepages it can allocate maximum according to its configuration parameters:

```
postgres@tantor:~$ sudo systemctl stop postgresql
postgres@tantor:~$ /usr/lib/postgresql/15/bin/postgres -c
config_file=/etc/postgresql/15/main/postgresql.conf -D
/var/lib/postgresql/15/main -C shared_memory_size_in_huge_pages
70
```

4) Allocate memory for 100 pages and check that they are allocated:

```
postgres@tantor:~$ sudo sysctl -w vm.nr_hugepages=100
vm.nr_hugepages = 100
postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages: 0 kB
```

Oleg Ivanov

```
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 100
HugePages_Free: 27
HugePages_Rsvd: 1
HugePages_Surp: 0
Hugepagesize: 2048 kB
Hugetlb: 204800 kB
```

The value **HugePages_Free : = 27** indicates that there are not enough free pages for a shared pool of 128MB. If you have more than 70 pages **,** then this amount is enough. If there are less than 80, **then** allocate memory for 300 pages and check that there are more than 80 free pages :

```
postgres@tantor:~$ sudo sysctl -w vm.nr_hugepages=300
vm.nr_hugepages = 300

postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 218
HugePages_Free: 145
HugePages_Rsvd: 1
HugePages_Surp: 0
Hugepagesize: 2048 kB
Hugetlb: 446464 kB
```

5) Run the free command:

```
postgres @ tantor :~$ free
               total used free shared buff/cache available
Mem : 4020796 1331220 571396 158696 2556992 2689576
Swap : 0 0 0
```

The output of the free command shows the following: used increased by 193,428 bytes, free decreased by 193,588 bytes, available decreased by 193,428 bytes.

6) Restart postgresql service :

```
postgres@tantor:~$ sudo systemctl restart postgresql
postgres@tantor:~$ cat /proc/meminfo | grep Huge
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 218
HugePages_Free: 126
HugePages_Rsvd: 73
HugePages_Surp: 0
Hugepagesize: 2048 kB
Hugetlb: 446464 kB
```

7) Get a list of processes that are using 2MB huge pages:

```
postgres@tantor:~$ sudo grep "KernelPageSize: 2048 kB" /proc/[[:digit:]]*/smaps
| awk {'print $1'} | cut -d "/" -f3 | sort | uniq
112647
112649
112650
112652
112653
112654
```

8) Compare with the process numbers (PID) of the Astralinux instance PostgreSQL :

```
postgres@tantor:~$ ps -ef | grep 15/
postgres 112647 1 0 17:55 ? 00:00:00 /usr/lib/postgresql/15/bin/ postgres -D
/var/lib/postgresql/15/main -c config_file=/etc/postgresql/15/main/postgresql.conf
postgres 112648 112647 0 17:55 ? 00:00:00 postgres: 15/main: logger
postgres 112649 112647 0 17:55 ? 00:00:00 postgres: 15/main: checkpointer
postgres 112650 112647 0 17:55 ? 00:00:00 postgres: 15/main: background writer
postgres 112652 112647 0 17:55 ? 00:00:00 postgres: 15/main: walwriter
postgres 112653 112647 0 17:55 ? 00:00:00 postgres: 15/main: autovacuum launcher
postgres 112654 112647 0 17:55 ? 00:00:00 postgres: 15/main: logical replication launcher
```

Numbers processes match . Instance uses huge pages .

The fact that the instance has started using huge pages will also be indicated by an increase in **the HugePages _ Rsvd value** or a decrease **in HugePages _ Free** . Due to the inconvenience of checking by these operating system metrics whether the pages are reserved by the instance or are already in use by the instance or other operating system processes, in version 17 of PostgreSQL the `huge_pages_status parameter was added` , which shows whether huge pages are used by the instance or not.

`the huge_pages = on` parameter . With this parameter value, the instance `will` not start unless it allocates huge pages.

Huge pages are reserved for use by parallel processes only at instance startup and only by the `min_dynamic_shared_memory parameter` . If huge pages are not reserved or there are not enough of them, parallel processes will allocate and use regular pages.

9) See the size and type of huge pages that the postgres process is using :

```
postgres@tantor:~$ cat /proc/ 112647 /smaps_rollup | grep tlb
Shared_Hugetlb: 12288 kB
Private_Hugetlb: 0 kB
```

postgres process uses shared huge pages.

Part 2. Changing the `oom_score value`

1) Run the command:

```
postgres@tantor:~$ for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END
{ getline cmd < "/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom <
"/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID %s) \n", PSS, oom, cmd,
'$PID'}' /proc/$PID/smaps; done|sort -n -r
28579 - 0 -> /opt/tantor/db/16/bin/postgres -D/var/lib/postgresql/tantor-se-16/data (PID 112360)
5629 -668->postgres: walwriter (PID 112364)
5166 - 70 -> /usr/lib/postgresql/15/bin/postgres -D/var/lib/postgresql/15/main-
cconfig_file=/etc/postgresql/15/main/postgresql.conf (PID 112647)
3207 -668->postgres: autovacuum launcher (PID 112365)
2945 -668->postgres: 15/main: autovacuum launcher (PID 112653)
2933 -668-> postgres: autoprewarm leader (PID 112366)
2685 -668-> postgres: 15/main: logical replication launcher (PID 112654)
2644 -668->postgres: checkpointer (PID 112361)
2618 -668-> postgres: logical replication launcher (PID 112368)
2366 -668-> postgres: pg_wait_sampling collector (PID 112367)
2157 -668->postgres: background writer (PID 112362)
1497 -667->postgres: 15/main: logger (PID 112648)
1479 -667->postgres: 15/main: checkpointer (PID 112649)
1468 -667->postgres: 15/main: walwriter (PID 112652)
1438 -667->postgres: 15/main: background writer (PID 112650)
```

command shows the value **of oom_score PostgreSQ** L instance processes .

The main process of the Tantor DBMS **instance has oom_score =** 0 . Astralinux PostgreSQL instance **oom_score=** 70 .

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The oom_score reduction was done by editing the service file. See content file :

```
postgres@tantor:~$ cat /usr/lib/systemd/system/tantor-se-server-16.service
[Unit]
Description=Tantor Special Edition database server 16
Documentation=https://www.postgresql.org/docs/16/static/
After=syslog.target
After=network.target

[Service]
Type=forking

User=postgres
Group=postgres

LimitNOFILE=infinity
LimitNOFILESoft=infinity

# Where to send early-startup messages from the server (before the logging options of postgresql.conf take effect)
# This is normally controlled by the global default set by systemd
# StandardOutput=syslog

# Disable OOM kill on the postmaster
OOMScoreAdjust=-1000
# ... but allow it still to be effective for child processes
# (note that these settings are ignored by Postgres releases before 9.5)
Environment=PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
Environment=PG_OOM_ADJUST_VALUE=0

# Maximum number of seconds pg_ctl will wait for postgres to start. Note that PGSTARTTIMEOUT should be less than
# TimeoutSec value.
Environment=PGSTARTTIMEOUT=270
Environment=PGDATA=/var/lib/postgresql/tantor-se-16/data

ExecStartPre=/opt/tantor/db/16/bin/postgresql-check-db-dir ${PGDATA}
ExecStart=/opt/tantor/db/16/bin/pg_ctl start -D ${PGDATA} -s -w -t ${PGSTARTTIMEOUT}
ExecStop=/opt/tantor/db/16/bin/pg_ctl stop -D ${PGDATA} -s -m fast
ExecReload=/opt/tantor/db/16/bin/pg_ctl reload -D ${PGDATA} -s

# Give a reasonable amount of time for the server to start up/shut down. Ideally, the timeout for starting
# PostgreSQL server should be handled more nicely by pg_ctl in ExecStart, so keep its timeout smaller than this
# value.
TimeoutSec=300

[Install]
WantedBy=multi-user.target
```

The file says that OOM kill is disabled for the postmaster process, but works for other instance processes. If you need to reduce the oom_score value , you can change the **PG_OOM_ADJUST_VALUE parameter value in this** file .

3) This point is optional, you can skip it. Reduce oom_score by 200. To do this, edit the service file and set the value **PG _ OOM _ ADJUST _ VALUE = -300** :

```
postgres@tantor:~$ su -
Password: root
root@tantor:~# cd /usr/lib/systemd/system/
root@tantor:~# mcedit tantor-se-server-16.service
edit and then save the file with F 2 ENTER close the editor with F 10
root@tantor:/usr/lib/systemd/system# cat tantor-se-server-16.service | grep OOM
# Disable OOM kill on the postmaster
OOMScoreAdjust=-1000
Environment=PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
Environment= PG_OOM_ADJUST_VALUE=-300
root@tantor:/usr/lib/systemd/system# systemctl daemon-reload
root@tantor:/usr/lib/systemd/system# restart
root@tantor:/usr/lib/systemd/system# for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END { getline cmd < "/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom < "/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID %s) \n", PSS, oom, cmd, '$PID'}' /proc/$PID/smaps; done|sort -n -r
33392 -0-> /opt/tantor/db/16/bin/postgres -D/var/lib/postgresql/tantor-se-16/data (PID 119419)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
4993 -70-> /usr/lib/postgresql/15/bin/postgres -D/var/lib/postgresql/15/main-
cconfig_file=/etc/postgresql/15/main/postgresql.conf (PID 112647)
3232 - 468 -> postgres: autovacuum launcher (PID 119424)
2865 -668->postgres: 15/main: autovacuum launcher (PID 112653)
2770 - 468 -> postgres: logical replication launcher (PID 119427)
2648 -668-> postgres: 15/main: logical replication launcher (PID 112654)
2533 - 468 -> postgres: background writer (PID 119421)
2523 - 468 -> postgres: pg_wait_sampling collector (PID 119426)
2019 -667->postgres: 15/main: checkpointer (PID 112649)
1777 - 468 -> postgres: autoprewarm leader (PID 119425)
1562 - 468 -> postgres: checkpointer (PID 119420)
1517 -667->postgres: 15/main: walwriter (PID 112652)
1480 -667->postgres: 15/main: background writer (PID 112650)
1475 -667->postgres: 15/main: logger (PID 112648)
1463 - 468 -> postgres: walwriter (PID 119423)
```

All processes of the tantor instance oom_score decreased by 200. After editing the service file, the configuration was reread using **the systemctl command daemon - reload** and restarted the instance.

4) Check that ksm is disabled:

```
postgres @ tantor :~$ cat / proc / vmstat | grep ksm
ksm_swpin_copy 0
cow_ksm 0
postgres@tantor:~$ cat /sys/kernel/mm/ksm/run
0
postgres@tantor:~$ cat /sys/kernel/mm/ksm/pages_scanned
0
```

This check is done just in case. ksm is useful and can be enabled if the operating system runs virtual machines from a single virtual machine image file.

5) Check the values of the parameters that affect the allocation of memory larger than the existing one:

```
postgres@tantor:~$ sudo sysctl -a | grep vm.over
vm.overcommit_kbytes = 0
vm.overcommit_memory = 0
vm.overcommit_ratio = 50
postgres@tantor:~$ sudo sysctl -a | grep vm.swap
vm.swappiness = 60
```

Since the swap partition is disabled in the virtual machine, `vm . overcommit _ memory` must be zero. Since swapping is disabled, the value of `the vm . swappiness parameter` doesn't matter.

Part 3. Unloading long lines with the `pg_dump utility`

1) Create a database named ela and a table with the following commands:

```
postgres@tantor:~$ psql
postgres=# create database ela;
CREATE DATABASE
postgres=# \c ela
You are now connected to database "ela" as user "postgres".NOTICE: table "t2"
does not exist, skipping
postgres=# drop table if exists t2;
create table t2(c1 text, c2 text);
insert into t2 (c1) VALUES (repeat('a', 1024*1024*512));
update t2 set c2 = c1;
\q
DROP TABLE
```

```
CREATE TABLE
INSERT 0 1
UPDATE 1
```

2) Stop the postgresql instance

```
postgres@tantor:~$ sudo systemctl stop postgresql
```

3) Set the number of huge pages to 0:

```
postgres@tantor:~$ sudo sysctl -w vm.nr_hugepages=0
vm.nr_hugepages = 0
```

4) Try it unload base data ela utility `pg_dump` :

```
postgres@tantor:~$ pg_dump -c -C -d ela -f ela.sql
pg_dump: error: Dumping the contents of table "t2" failed: PQgetResult() failed.
pg_dump: detail: Error message from server: ERROR: out of memory
DETAIL: Cannot enlarge string buffer containing 536870913 bytes by 536870912 more bytes.
pg_dump: detail: Command was: COPY public.t2 (c1, c2) TO stdout;
```

An error occurred that the contents of the table could not be unloaded due to the memory size for `the string being exceeded. buffer` . The maximum buffer size is 1GB.

5) Starting with version 16.2, the Tantor DBMS has a parameter `enable_large_allocations` , which increases the size of the string buffer to 2GB. The parameter can also be set in pg_dump utility sessions using the utility parameter. Dump the ela database using this parameter:

```
postgres @ tantor :~$ time pg_dump - c - d ela - f ela . sql -- enable - large
- allocations

real 0m12.556s
user 0m1.614s
sys 0m0.699s
```

ela database download took 12.556 seconds.

6) A file of 1GB size was created:

```
postgres@tantor:~$ ls -al ela.sql
-rw-r--r-- 1 postgres postgres 1073742755 ela.sql
```

Part 4. Out of Memory

1) In the terminal window, check how much memory is free:

```
postgres@tantor:~$ free
total used free shared buff/cache available
Mem: 4020792 993520 2888632 185788 546988 3027272
Swap: 0 0 0
```

Free 2.8GB , available 3GB
2) Run the psql utility:

```
postgres @ tantor :~$ psql - d ela
Type "help" for help.

ela=# \c ela
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

3) In another terminal, look at the list of PostgreSQL instance processes:

```
root@tantor:~$ for PID in $(pgrep "postgres"); do awk '/Pss/ {PSS+=$2} END { getline cmd <
"/proc/'$PID'/cmdline"; sub("\0", " ", cmd); getline oom < "/proc/'$PID'/oom_score"; printf "%.0f -%s-> %s (PID
%s) \n", PSS, oom, cmd, '$PID'}' /proc/$PID/smaps; done|sort -n -r
3039132 -922->postgres: postgres ela [local] COPY (PID 12518)
28619 -0-> /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data (PID 12499)
5502 -668->postgres: walwriter (PID 12503)
3031 -668->postgres: autovacuum launcher (PID 12504)
2753 -668->postgres: checkpointer (PID 12500)
2507 -668-> postgres: logical replication launcher (PID 12507)
2499 -668-> postgres: pg_wait_sampling collector (PID 12506)
2069 -668->postgres: background writer (PID 12501)
1697 -668-> postgres: autoprewarm leader (PID 12505)
```

The first line contains the number of the server process connected to the ela database .

3) In the psql window , execute the script obtained by the pg_dump utility :

```
ela=# \i ela.sql
SET
CREATE TABLE
ALTER TABLE
psql:ela.sql:44: server closed the connection unexpectedly
This probably means the server terminated abnormally
before or while processing the request.
psql:ela.sql:44: error: connection to server was lost
```

The connection was lost.

4) Look at the latest messages from the Linux kernel:

```
postgres@tantor:~$ sudo dmesg | tail -n 120
```

```
[16387.760734] Mem-Info:
[16387.760737] active_anon:342614 inactive_anon:609810 isolated_anon:0
active_file:22 inactive_file:71 isolated_file:0
unevictable:0 dirty:8 writeback:0
slab_reclaimable:5664 slab_unreclaimable:13043
mapped:15337 shmem:46435 pagetables:5001
sec_pagetables:0 bounce:0
kernel_misc_reclaimable:0
free:20934 free_pcp:1 free_cma:0
[16387.760743] Node 0 active_anon:1370456kB inactive_anon:2439240kB active_file:88kB inactive_file:284kB
unevictable:0kB isolated(anon):0kB isolated(file):0kB mapped:61348kB dirty:32kB writeback:0kB shmem:185740kB
shmem_thp:0kB shmem_pmdmapped:0kB anon_thp:0kB writeback_tmp:0kB kernel_stack:5696kB pagetables:20004kB
sec_pagetables:0kB all_unreclaimable? no
[16387.760749] Node 0 DMA free:14848kB boost:0kB min:260kB low:324kB high:388kB reserved_highatomic:0KB
active_anon:0kB inactive_anon:0kB active_file:0kB inactive_file:0kB unevictable:0kB writepending:0kB
present:15992kB managed:15360kB mlocked:0kB bounce:0kB free_pcp:0kB local_pcp:0kB free_cma:0kB
[16387.760756] lowmem_reserve[]: 0 3385 3825 3825 3825
[16387.760770] Node 0 DMA32 free:61272kB boost:0kB min:59580kB low:74472kB high:89364kB
reserved_highatomic:0KB active_anon:1077032kB inactive_anon:2365224kB active_file:0kB inactive_file:0kB
unevictable:0kB writepending:32kB present:3653568kB managed:3555048kB mlocked:0kB bounce:0kB free_pcp:4kB
local_pcp:0kB free_cma:0kB
[16387.760778] lowmem_reserve[]: 0 0 439 439 439
[16387.760791] Node 0 Normal free:7616kB boost:0kB min:7736kB low:9668kB high:11600kB reserved_highatomic:0KB
active_anon:105944kB inactive_anon:261496kB active_file:416kB inactive_file:0kB unevictable:0kB writepending:0kB
present:524288kB managed:450384kB mlocked:0kB bounce:0kB free_pcp:0kB local_pcp:0kB free_cma:0kB
[16387.760798] lowmem_reserve[]: 0 0 0 0 0
[16387.760811] Node 0 DMA: 0*4kB 0*8kB 0*16kB 0*32kB 0*64kB 0*128kB 0*256kB 1*512kB (U) 0*1024kB 1*2048kB (M)
3*4096kB (M) = 14848kB
[16387.760846] Node 0 DMA32: 849*4kB (UME) 615*8kB (UME) 229*16kB (UME) 278*32kB (UME) 177*64kB (UE) 67*128kB
(UME) 34*256kB (UME) 15*512kB (UME) 1*1024kB (U) 0*2048kB 1*4096kB (M) = 62284kB
[16387.760893] Node 0 Normal: 230*4kB (UME) 153*8kB (UME) 107*16kB (UME) 71*32kB (UME) 25*64kB (UME) 5*128kB
(M) 0*256kB 0*512kB 0*1024kB 0*2048kB 0*4096kB = 8368kB
[16387.760934] Node 0 hugepages_total=0 hugepages_free=0 hugepages_surp=0 hugepages_size=2048kB
[16387.760937] 46530 total pagecache pages
[16387.760939] 0 pages in swap cache
[16387.760941] Free swap = 0kB
[16387.760943] Total swap = 0kB
[16387.760946] 1048462 pages RAM
[16387.760948] 0 pages HighMem/MovableOnly
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
[16387.760950] 43264 pages reserved
[16387.760952] 0 pages hwpoisoned
[16387.760954] Tasks state (memory values in pages ):
[16387.760956] [ pid ] uid tgid total_vm      rss pgtables_bytes swapents oom_score_adj name
[16387.760961] [270] 0 270 13053 288 118784 0 -250 systemd-journal
[16387.760966] [299] 0 299 8116 608 86016 0 -1000 systemd-udevd
[16387.760971] [ 426] 0 426 3359 66 49152 0 -1000 auditd
[16387.760975] [451] 0 451 59920 269 102400 0 0 accounts-daemon
[16387.760978] [452] 0 452 658 32 45056 0 0 acpid
[16387.760982] [455] 109 455 2246 96 61440 0 0 avahi-daemon
[16387.760986] [456] 0 456 1814 64 53248 0 0 cron
[16387.760989] [457] 100 457 2605 448 61440 0 -900 dbus-daemon
[16387.760993] [461] 109 461 2202 98 57344 0 0 avahi-daemon
[16387.760996] [467] 995 467 60688 895 110592 0 0 polkitd
[16387.761000] [473] 0 473 6953 256 81920 0 0 systemd-logind
[16387.761003] [562] 0 562 67078 832 163840 0 0 NetworkManager
[16387.761007] [578] 0 578 4923 256 77824 0 0 wpa_supplicant
[16387.761010] [585] 0 585 80121 589 122880 0 0 ModemManager
[16387.761013] [703] 114 703 423624 1552 208896 0 0 docker-registry
[16387.761017] [ 721] 0 721 340017 3498 299008 0 -999 containerd
[16387.761020] [ 786] 0 786 4954 448 81920 0 -1000 sshd
[16387.761024] [865] 0 865 620 96 45056 0 0 fly-getexe
[16387.761027] [891] 0 891 1630 64 49152 0 0 agetty
[16387.761031] [ 1744] 0 1744 9731 4013 118784 0 0 astra-event-dia
[16387.761034] [ 1768] 0 1768 384560 7030 425984 0 -500 dockerd
[16387.761038] [1774] 0 1774 345665 5100 466944 0 0 syslog-ng
[16387.761041] [5571] 0 5571 3670 192 73728 0 0 fly-dm
[16387.761044] [5578] 102 5578 100486 24100 581632 0 0 Xorg
[16387.761047] [5587] 0 5587 106865 136 106496 0 0 VBoxService
[16387.761051] [5642] 102 5642 5567 512 81920 0 100 systemd
[16387.761054] [5645] 102 5645 28019 1385 102400 0 100 (sd-pam)
[16387.761058] [5670] 0 5670 6821 400 102400 0 0 fly-dm
[16387.761061] [5682] 1000 5682 5610 576 81920 0 100 systemd
[16387.761064] [5683] 1000 5683 28019 1385 102400 0 100 (sd-pam)
[16387.761068] [5698] 1000 5698 13047 4141 147456 0 0 fly-wm
[16387.761071] [5758] 1000 5758 2317 256 57344 0 200 dbus-daemon
[16387.761074] [5791] 1000 5791 60058 832 192512 0 200 kglobalaccel5
[16387.761078] [5798] 1000 5798 4628 98 57344 0 0 VBoxClient
[16387.761081] [5799] 1000 5799 37695 130 69632 0 0 VBoxClient
[16387.761085] [5811] 1000 5811 4628 130 61440 0 0 VBoxClient
[16387.761088] [5812] 1000 5812 37687 98 73728 0 0 VBoxClient
[16387.761091] [5817] 1000 5817 4628 130 57344 0 0 VBoxClient
[16387.761095] [5819] 1000 5819 37816 66 69632 0 0 VBoxClient
[16387.761098] [5822] 1000 5822 4628 98 57344 0 0 VBoxClient
[16387.761101] [5823] 1000 5823 37748 194 73728 0 0 VBoxClient
[16387.761104] [5837] 0 5837 58952 256 98304 0 0 upowerd
[16387.761108] [5844] 1000 5844 2867 202 49152 0 0 ssh-agent
[16387.761111] [5870] 1000 5870 103544 1216 258048 0 0 polkit-kde-auth
[16387.761115] [5871] 1000 5871 125625 1600 290816 0 0 org_kde_powerde
[16387.761118] [5873] 1000 5873 67186318 736 196608 0 0 baloo_file
[16387.761121] [5876] 1000 5876 188404 3439 364544 0 0 fly-notificatio
[16387.761124] [5877] 1000 5877 39425 576 163840 0 0 astra-event-wat
[16387.761128] [5880] 1000 5880 36132 384 139264 0 0 fly-cups-watch
[16387.761131] [5881] 1000 5881 63043 1056 208896 0 0 kscreend
[16387.761135] [5882] 1000 5882 67439028 7809 712704 0 0 fly-start-menu
[16387.761138] [5883] 1000 5883 67294 1184 241664 0 0 fly-shutdown-sc
[16387.761142] [5886] 1000 5886 69346 1391 262144 0 0 fly-touchpadd
[16387.761145] [5892] 1000 5892 77765 288 102400 0 0 at-spi-bus-laun
[16387.761148] [5896] 1000 5896 58767 160 94208 0 0 agent
[16387.761152] [5900] 1000 5900 90195 1472 282624 0 0 fly-notify-osd-
[16387.761155] [5911] 1000 5911 393645 8394 720896 0 0 fly-sound-apple
[16387.761158] [5916] 1000 5916 87538 1600 266240 0 0 fly-reflex-serv
[16387.761171] [5921] 1000 5921 2208 160 61440 0 0 dbus-daemon
[16387.761176] [5922] 1000 5922 14788 3917 163840 0 0 applet.py
[16387.761179] [5929] 1000 5929 4633 256 73728 0 0 compton
[16387.761182] [5944] 1000 5944 123154 1376 278528 0 200 kactivitymanage
[16387.761186] [5974] 1000 5974 106825 418 106496 0 0 pulseaudio
[16387.761190] [5976] 0 5976 99686 565 139264 0 0 udisksd
[16387.761193] [5985] 1000 5985 60389 384 106496 0 0 gsettings-helpe
[16387.761196] [6002] 1000 6002 41071 224 86016 0 0 at-spi2-registr
[16387.761200] [6003] 1000 6003 161984 4706 495616 0 0 nm-applet
[16387.761203] [6017] 1000 6017 59383 832 188416 0 200 kscreen_backend
[16387.761207] [ 6067] 1000 6067 59604 224 102400 0 200 gvfsd
[16387.761211] [6450] 1000 6450 154277 5493 385024 0 0 fly-term
[16387.761214] [6459] 1000 6459 2225 416 57344 0 0 bash
[16387.761218] [6835] 1000 6835 5043 320 81920 0 0 su
[16387.761221] [6836] 113 6836 2322 448 53248 0 0 bash
[16387.761225] [8280] 0 8280 8723 1216 102400 0 0 cupsd
[16387.761229] [10858] 1000 10858 154156 5320 376832 0 0 fly-term
[16387.761232] [10865] 1000 10865 2225 384 57344 0 0 bash
[16387.761236] [12288] 113 12288 66375 8928 229376 0 -900 postgres
[16387.761239] [12289] 113 12289 19967 637 122880 0 0 postgres
[16387.761243] [12290] 113 12290 66442 1313 155648 0 0 postgres
[16387.761246] [12291] 113 12291 66409 1185 151552 0 0 postgres
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
[16387.761249] [12295] 113 12295 66375 1729 143360 0 0 postgres
[16387.761252] [12296] 113 12296 66878 993 172032 0 0 postgres
[16387.761256] [12297] 113 12297 66854 865 159744 0 0 postgres
[16387.761260] [12499] 113 12499 57491 4352 184320 0 -1000 postgres
[16387.761263] [12500] 113 12500 57528 1104 151552 0 0 postgres
[16387.761267] [12501] 113 12501 57524 1008 147456 0 0 postgres
[16387.761271] [12503] 113 12503 57524 1744 143360 0 0 postgres
[16387.761274] [12504] 113 12504 57923 1008 163840 0 0 postgres
[16387.761277] [12505] 113 12505 57524 742 147456 0 0 postgres
[16387.761280] [12506] 113 12506 57635 880 143360 0 0 postgres
[16387.761283] [12507] 113 12507 57891 912 151552 0 0 postgres
[16387.761287] [12517] 113 12517 6495 448 94208 0 0 psql
[16387.761290] [ 12518 ] 113 12518 1630962  789553 6533120 0 0 postgres
[16387.761293] [12540] 1000 12540 5043 256 86016 0 0 su
[16387.761297] [12541] 113 12541 2221 384 57344 0 0 bash
```

```
[16387.761300] oom-kill:constraint=CONSTRAINT_NONE,nodemask=(null),
cpuset=tantor-se-server-16.service,mems_allowed=0,global_oom,
task_memcg=/system.slice/tantor-se-server-16.service,task=postgres,pid=12518,
uid=113
[16387.761329] Out of memory: Killed process 12518 ( postgres ) total-vm:
6523848kB , anon-rss: 3151300kB , file-rss:0kB, shmem-rss:6912kB, UID:113
pgtables:6380kB oom_score_adj:0
```

The OOM kill process stopped the server process that was running the script.

When OOM-kill was triggered, the `oom_score_adj value` did not play any role, since there were no memory consumers in the operating system except for PostgreSQL processes .

The OOM kill process shows the **virtual** memory size `total - vm : 6523848 kB` . **`6523848 kB`** `/4 kB` (page size) = **`1630962`** (in pages) is the memory size of the server process at the time it was stopped.

Memory allocation in diagnostic messages is shown in 4K pages, except for **`the pgtables_bytes column`** :

```
[16387.760954] Tasks state (memory values in pages ):
[16387.760956] [ pid ] uid tgid total_vm     rss pgtables_bytes swapents oom_score_adj name
[16387.761290] [ 12518 ] 113 12518 1630962  789553 6533120 0 0 postgres
```

===== ==== note ==========

The server process read a file into its local memory, which was 3GB in size .

The virtual machine has 4GB and no swap. According to the `free utility` , the available memory (`available`) was **`3027272`** `k ibi` . **At the same time, the operating system allocated `6523848 kB` of virtual memory** . If the value `vm.overcommit_memory =2` with `vm . overcommit_ratio =50, then the process would be able to allocate only 1.5GB of memory and would be denied an attempt to allocate more, despite the fact that the available memory was 3GB. In this case ,` the OOM-kill process would not manifest itself. Example :

```
postgres=# \i ela.sql
SET
CREATE TABLE
ALTER TABLE
psql:ela.sql:44: ERROR: out of memory
DETAIL: Failed on request of size 2147483646 in memory context "COPY".
CONTEXT: COPY t2, line 1
```

`vm.overcommit_memory=` 2 on a virtual machine without swap enabled , or increase:

```
root@tantor:~# sysctl -w vm.overcommit_ratio=100
```

otherwise you will not be able to give any command :

```
root@tantor:~# free
total used free shared buff/cache available
Mem: 4020796 948788 1958532 188700 1533332 3072008
Swap: 0 0 0
root@tantor:~# sysctl -w vm.overcommit_memory=2
vm.overcommit_memory = 2
root@tantor:~# free
-bash: fork: Cannot allocate memory
root@tantor:~# reboot
```

```
-bash: fork: Cannot allocate memory
```

After enabling swap in the next part of the practice, you can check that with `vm . overcommit _ memory =2` OOM kill does not work, instead of working, the process allocating memory is given an error.

=====================

5) View the list of PostgreSQL processes:

```
postgres @ tantor :~$ for PID in $( pgrep " postgres "); do awk '/ Pss /{ PSS +=$2} END { getline cmd < "/ proc /'$ PID '/ cmdline "; sub ("\0", "", cmd ); getline oom < "/ proc / '$ PID '/ oom_score " ; printf "%.0 f -% s -> % s ( PID % s ) \ n ", PSS , oom , cmd , '$ PID '}' / proc / $ PID / smaps ; done | sort - n - r
```

The list is empty, or only Astralinux instance processes are running PostgreSQL . After an OOM kill stopped the server process , the instance was restarted according to the value of the `restart_after_crash` configuration parameter :

```
postgres=# select name, setting, context, max_val, min_val from pg_settings where name ~ 'restart';
name | setting | context | max_val | min_val
---------------------+---------+---------+---------+---------
restart_after_crash | on | sighup | |
(1 row )
```

By default, this option is enabled, and after a server process crashes, the postgres process will crash all child processes (equivalent to an immediate stop without a checkpoint) and restart the processes (equivalent to a " crash recovery " - starting the instance after a crash). If you disable this option, all processes in the instance will be forcibly stopped, including the postgres process. The state of the cluster after stopping will be the same as after a forced stop with the `pg_ctl stop -m immediate` command :

```
postgres@tantor:~$ pg_controldata | grep state
Database cluster state: in production
```

6) Launch instance tantor:

```
postgres@tantor:~$ sudo systemctl start tantor-se-server-16
```

If the instance does not start, you can wait until it stops or issue the command **`pg_ctl stop - m immediate`** and repeat the **`sudo command systemctl start tantor - se - server -16`**

7) This point can be omitted. A simple query can cause a lack of memory and stop psql :

```
postgres@tantor:~$ psql

postgres=# select repeat('a', 100000000) from generate_series(1, 100);
Killed
postgres@tantor:~$ sudo dmesg | tail -5
[ 1002.311864] [ 6421] 113 6421 747298 718592 5861376 0 0 psql
[ 1002.311867] [ 6422] 113 6422 139618 75066 770048 0 0 postgres
[ 1002.311870] oom-kill :constraint=CONSTRAINT_NONE,nodemask=(null),
cpuset=containerd.service,mems_allowed=0,global_oom,task_memcg=/user.slice/user-1000.slice/session-3.scope, task=psql,pid= 6421 ,uid=113
[ 1002.311894] Out of memory: Killed process 6421 ( psql ) total-vm:2989192kB, anon-rss:2874368kB, file-rss:0kB, shmem-rss:0kB, UID:113 pgtables:5724kB oom_score_adj:0
```

If `vm.overcommit_memory=2` then the result will be:

```
postgres=# select repeat('a', 100000000) from generate_series(1, 100);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
out of memory for query result
```

set `vm.overcommit_memory=2` only after setting `vm . overcommit_ratio = 100 or enabling swap, otherwise the virtual machine will immediately run out of memory and you will not be able to issue` any commands.

OOM kill usually kills the psql process, but it can also kill instance processes, causing the instance to crash:

```
postgres=# select repeat('a', 1000000000) from generate_series(1, 1000);
server closed the connection unexpectedly
This probably means the server terminated abnormally
before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.
!?>
```

```
[1211447.321264] [ pid ] uid tgid total_vm rss pgtables_bytes swapents oom_score _adj name
[1211447.321658] [524388] 113 524388 756951 205344 4251648 313792 0 psql
[1211447.321661] [524389] 113 524389   808432 716332 6062080 17383 0 postgres
[1211447.321664] oom-kill :constraint=CONSTRAINT_NONE,nodemask=(null),cpuset=user.slice,mems_allowed=0,
global_oom,task_memcg=/system.slice/tantor-se-server-16.service,task=postgres,pid=524389,uid=113
[1211447.321735] Out of memory: Killed process 524389 (postgres) total-vm:3233728kB, anon-rss:2863536kB,
file-rss:1664kB, shmem-rss:128kB, UID:113 pgtables:5920kB oom_score_adj:0
```

In this case, the instance may not start and it will need to be started with the command:
`sudo systemctl start tantor-se-server-16`

Part 5. Enabling swapping

1) Add a 2GB swap file and enable swap:

```
root@tantor:~# dd if=/dev/zero of=/swap_file bs=1M count=2048
2048+0 records in
2048+0 records out
2147483648 bytes (2.1 GB, 2.0 GiB) copied, 22.3827 s, 95.9 MB/s
root@tantor:~# chmod 600 /swap_file
root@tantor:~# mkswap /swap_file
Setting up swapspace version 1, size = 2 GiB (2147479552 bytes)
no label, UUID=4b56ea3b-ac66-46ef-801b-2aa473f27ef6
root@tantor:~# swapon /swap_file
root@tantor:~# free -m
total used free shared buff/cache available
Mem : 3926 881 672 19 2680 3044
Swap : 2047 0 2047
```

Swapping will not be enabled automatically when you reboot the virtual machine.

2) Do it script dump :

```
postgres=# \timing
Timing is on.
ela=# \i ela.sql
...
SET
CREATE TABLE
ALTER TABLE
COPY 1
Time: 18,979.619 ms (00:18.980)
```

The COPY command took **18** seconds to load the string.

3) In the same psql window in which the loading script was executed, look at the `value of the enable_large_allocations parameter :`

```
ela=# show enable_large_allocations ;
enable_large_allocations
--------------------------
 on
(1 row)
```

The parameter was enabled when executing the ela.sql file with the command:
`SET enable_large_allocations TO on;` from this file .
By default, the option is disabled.

4) In another terminal window, look at the beginning of the `ela.sql file` :

```
postgres@tantor:~$ head -n 20 ela.sql
--
-- PostgreSQL database dump
SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET transaction_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;
SET enable_large_allocations TO on;
```

Meaning parameter `enable_large_allocations` was unloaded utility **`pg_dump`** .

5) look at the `characteristics of the parameter enable_large_allocations` :

```
ela=# select * from pg_settings where name like '%large%' \gx
-[ RECORD 1 ]---+------------------------------------------
name | enable_large_allocations
setting | on
unit |
category | Resource Usage/Memory
short_desc | Sets whether to use large memory buffers - greater than 1 Gb, up to 2 Gb
extra_desc |
context | superuser
vartype | bool
source | session
min_val |
max_val |
enumvals |
boot_val | off
reset_val | off
sourcefile |
sourceline |
pending_restart | f
```

The parameter can be set at different levels, including the session level.
The parameter increases the size of the string buffer from 1GB to 2GB. Strings up to 2GB in size can be unloaded. Strings larger than 2GB will not be unloaded.

Part 6. Page cache

1) Open or switch to a terminal with root privileges :

```
postgres@tantor:~$ su -
Password: root
```

2) Run the commands:

```
root@tantor:~# cat /proc/meminfo | grep Cached
Cached: 1367440 kB
SwapCached: 49328 kB
root@tantor:~# free
total used free shared buff/cache available
Mem : 4020792 605576 2250740 36796 1452148 3415216
Swap : 2097148 508696 1588452
```

By default, the free command displays values in kibibytes (bytes divided by 1024 1024). To display in kilobytes, use the --kylo or --si option (bytes divided by 1000). The difference noticeable , in second discharge left (3 6 820 3 7 703):

```
root@tantor:~# free
total used free shared buff/cache available
Mem: 4020792 575192 3493796 3 6 820 215152 3445600
Swap: 2097148 507160 1589988
root@tantor:~# free --si
total used free shared buff/cache available
Mem: 4117291 589750 3576889 3 7 703 220323 3527540
Swap: 2147479 519331 1628147
root@tantor:~# free --bytes
total used free shared buff/cache available
Mem: 4117291008 588890112 3577745408 3 7 703 680 220340224 3528400896
Swap: 2147479552 519331840 1628147712
```

`Cached` is included in `buff / cache` , the difference (taking into account different units of measurement) is the value of `buff` .

3) Make clean page cache blocks free:

```
root@tantor:~# echo 1 > /proc/sys/vm/drop_caches
root@tantor:~# cat /proc/meminfo | grep Cached
Cached: 156684 kB
SwapCached: 49332 kB
root@tantor:~# free
total used free shared buff/cache available
Mem : 4020792 536520 3530520 36800 184272 3484272
Swap : 2097148 508696 1588452
```

Values `Cached` and `buff / cache` decreased by 1.2GB. Value `free` increased.

4) Look at the number of pages of different sizes:

```
root@tantor:~# cat /proc/buddyinfo
Node 0, zone DMA 0 0 0 0 0 0 0 1 0 1 3
Node 0, zone DMA32 1670 1466 868 707 734 503 294 166 111 111 648
Node 0, zone Normal 444 283 262 98 31 10 3 39 32 2 1
```

Node 0 - physical processor number. Zones:
`zone DMA` - virtual memory with an offset from zero to 16 MB
`zone DMA 32` - from 16MB to 4GB
`zone Normal` - from 4GB to `2^48` (2 to the power of 48)
Each zone is divided into parts of the memory address space of size (`4096 bytes *2^ n`):
`4Kb 8Kb 16Kb 32Kb 64Kb 128Kb 256Kb 512Kb 1Mb 2Mb 4Mb`

5) Look at the page set fragmentation index:

```
root@tantor:~# cat /sys/kernel/debug/extfrag/extfrag_index
Node 0, zone DMA -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone DMA32 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone Normal -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 0.964 0.982 0.991 0.996
```

Defragmentation will start automatically at decrease free memory below vm.min_free_kbytes=67584 or meanings vm.watermark_scale_factor . The default value is vm.watermark_scale_factor=10 , which means 0.1% of the free physical memory. These values are too small and defragmentation is launched as a last resort. Recommendations for setting values are given in the theoretical part of the chapter.

6) Run defragmentation manually:

```
root@tantor:~# echo 1 > /proc/sys/vm/compact_memory
```

7) Check the defragmentation result (how the fragmentation index has changed):

```
root@tantor:~# cat /sys/kernel/debug/extfrag/extfrag_index
Node 0, zone DMA -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone DMA32 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000
Node 0, zone Normal -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 -1.000 0.991
```

8) Look how many dirty 4 KB pages are in the linux page cache:

```
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 126
nr_dirty_threshold 176446
nr_dirty_background_threshold 88115
```

9) Run the sync command, which flushes dirty pages to disk:

```
root@tantor:~# sync
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 0
nr_dirty_threshold 174962
nr_dirty_background_threshold 87374
root@tantor:~# cat /proc/vmstat | grep dirty
nr_dirty 2
nr_dirty_threshold 175071
nr_dirty_background_threshold 87428
```

The sync command flushes all dirty pages and nr_dirty becomes 0 at some point, but when processes are running, the pages constantly become dirty and the value increases almost immediately .

10) Look at the values of the parameters related to working with dirty pages:

```
root@tantor:~# sysctl -a | grep dirty
vm.dirty_background_bytes = 0
vm.dirty_background_ratio = 10
vm.dirty_bytes = 0
vm.dirty_expire_centisecs = 3000
vm.dirty_ratio = 20
vm.dirty_writeback_centisecs = 500
vm.dirtytime_expire_seconds = 43200
```

The duration of page retention from the moment of change is determined by the parameter: vm . dirty_expire_centisecs - how much the buffer can be dirty before it is marked for writing, default is 3000 (30 seconds), can be changed to 500 (5 seconds) .
The retention is also affected by the parameter: vm . dirty _ writeback _ centisecs - the waiting period between writes to disk, by default 500 (5 seconds), can be reduced to 250 (2.5 seconds). Recommended values for other parameters:

```
vm . dirty_ratio = 10
```

```
vm . dirty _ background _ ratio = 3
```

These parameters can be left unchanged, since the linux cache pages are sent to write by the instance processes by calling fdatasync on WAL , fsync on tablespace files at the end of the checkpoint and do this for fault tolerance (protection against power failure). These parameters can be used to smooth out the peak load that occurs at the end of the checkpoint. But even for load smoothing there are PostgreSQL parameters checkpoint_flush_after and bgwriter_flush_after , which are also set to optimal values for normal load and normal equipment (HDD and SSD).

11) Look at the PostgreSQL configuration parameters that affect the retention of dirty pages in the page cache:

```
postgres=# \dconfig *flush*
List of configuration parameters
Parameter | Value
------------------------+-------
backend_flush_after | 0
bgwriter_flush_after | 512kB
checkpoint_flush_after | 256kB
wal_writer_flush_after | 1MB
(4 rows)
```

These settings limit the amount of dirty pages in the linux page cache and reduce the chance of stalling when fsyncing at the end of a checkpoint or when linux flushes dirty pages to disk according to the settings in the previous item. If the settings in the previous item allow a large number of dirty pages to accumulate, relatively long delays may occur, making the instance run less smoothly.

12) Execute the following commands :

```
postgres=# alter system set shared_preload_libraries = pg_stat_statements,
pg_wait_sampling, pg_stat_kcache, pg_qualstats, pg_store_plans, pg_prewarm;
```

13) Restart the instance, as the parameter change will only take effect after the instance is restarted:

```
root@tantor:~# systemctl restart tantor-se-server-16
```

load shared libraries into process memory when the instance starts . The loaded pg_wait_sampling library will be needed in the next practice.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Practice for Chapter 3

Part 1. Standard pgbench test

1) In the discussion https :// www . postgresql . org / message - id / flat /53 FD 5 D 6 C .40105%40 catalyst . net . nz an example of a pgbench test with HyperThread enabled and disabled on a 4-core processor was given . Let's see on what number of clients the maximum tps is achieved on your virtual machine. Create test tables with a scale factor of 300 as in the test at the link:

```
postgres@tantor:~$ pgbench -i -s 300
dropping old tables...
creating tables...
generating data (client-side)...
30000000 of 30000000 tuples (100%) done (elapsed 50.12 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 78.79 s (drop tables 0.02 s, create tables 0.01 s, client-side generate
50.79 s, vacuum 2.46 s, primary keys 25.52 s).
```

The tables took 78 seconds to create.

2) Perform 15 second tests with different numbers of connections:

```
postgres@tantor:~$ pgbench -c 64 -T 15 2> /dev/null | grep tps
tps = 1773.570100 (without initial connection time)
postgres@tantor:~$ pgbench -c 100 -T 15 2> /dev/null | grep tps
tps = 1646.101354 (without initial connection time)
postgres@tantor:~$ pgbench -c 32 -T 15 2> /dev/null | grep tps
tps = 1685.528291 (without initial connection time)
postgres@tantor:~$ pgbench -c 16 -T 15 2> /dev/null | grep tps
tps = 1550.934797 (without initial connection time)
```

Maximum tps is achieved with 64 connections.

3) Repeat the test:

```
postgres @ tantor :~$ pgbench - c 64 - T 15 2> / dev / null | grep tps
tps = 1676.523228 (without initial connection time)
postgres@tantor:~$ pgbench -c 100 -T 15 2> /dev/null | grep tps
tps = 1572.848543 (without initial connection time)
postgres@tantor:~$ pgbench -c 32 -T 15 2> /dev/null | grep tps
tps = 1694.453483 (without initial connection time)
postgres@tantor:~$ pgbench -c 16 -T 15 2> /dev/null | grep tps
tps = 1477.982431 (without initial connection time)
```

tps values have dropped slightly in all tests. The default test makes changes to table rows and index records. Repeated runs of the tests give consistently worse results. The test with table changes cannot be used without recreating the tables.
In the linked discussion, depending on the processor, the maximum tps was achieved at 256, 96, 48 connections on processors with different numbers of cores. Accuracy to the nearest power of two: the value given is 256, and the real maximum can be in the range from 140 to 350.

4) You can use a select-only test that does not change data. This test depends on the buffer cache filling and the first test executions have a lower tps than the subsequent ones. The maximum tps was achieved at 16 connections.

```
postgres@tantor:~$ pgbench -b select-only -c 64 -T 15 2> /dev/null | grep tps
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
tps = 8687.455945 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 32 -T 15 2> /dev/null | grep tps
tps = 9112.963307 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 16 -T 15 2> /dev/null | grep tps
tps = 9611.093353 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 8 -T 15 2> /dev/null | grep tps
tps = 9548.481330 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 4 -T 15 2> /dev/null | grep tps
tps = 7841.653129 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 100 -T 15 2> /dev/null | grep tps
tps = 8780.049265 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 64 -T 15 2> /dev/null | grep tps
tps = 9200.191422 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 32 -T 15 2> /dev/null | grep tps
tps = 9499.272270 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 16 -T 15 2> /dev/null | grep tps
tps = 9820.530795 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 8 -T 15 2> /dev/null | grep tps
tps = 9637.473378 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -c 4 -T 15 2> /dev/null | grep tps
tps = 7825.917563 (without initial connection time)
```

Part 2. Binding processes to a processor core

If the host has a single processor, then binding PostgreSQL instance processes is unlikely to provide any benefit. The operating system binds its threads to cores. Threads that work with the same memory structures are assigned to the same core. When binding instance processes, it is worth considering which cores the operating system threads are assigned to, so that these cores do not become a bottleneck.

1) Look at the number of processor cores:

```
postgres@tantor:~$ nproc
4
```

2) Run the command to see the linux kernel thread affinity . The kernel thread daemon kthreadd has a process ID of pid= 2 :

```
for tid in $( pgrep - w 2 | tr '\ n ' ' '); do
 ps --no-headers -q "${tid}" -o pid -o user -o psr -o cmd;
taskset -cp "${tid}";
done
26 root 2 [cpuhp/2]
pid 26's current affinity list: 2
27 root 2 [idle_inject/2]
pid 27's current affinity list: 2
28 root 2 [migration/2]
pid 28's current affinity list: 2
29 root 2 [ksoftirqd/2]
pid 29's current affinity list: 2
73 root 3 [scsi_eh_2]
pid 73's current affinity list: 0-3
137 root 2 [kworker/2:1H-kblockd]
...
```

Part streams linux kernels are tied To to everyone from cpu cores . kernel threads are used by linux for execution background operations .

3) Run the command to see the affinity of processes named postgres :

```
for tid in $(pgrep postgres | tr '\n' ' '); do
ps --no-headers -q "${tid}" -o pid -o user -o psr -o cmd;
taskset -cp "${tid}";
done

126673 postgres 1 /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
pid 126673's current affinity list: 0-3
 126674 postgres 1 postgres: checkpointer
pid 126674 's current affinity list: 0-3
126675 postgres 3 postgres: background writer
pid 126675's current affinity list: 0-3
126677 postgres 0 postgres: walwriter
pid 126677's current affinity list: 0-3
126678 postgres 1 postgres: autovacuum launcher
pid 126678's current affinity list: 0-3
126679 postgres 3 postgres: autoprewarm leader
pid 126679's current affinity list: 0-3
126680 postgres 0 postgres: pg_wait_sampling collector
pid 126680's current affinity list: 0-3
126681 postgres 3 postgres: logical replication launcher
pid 126681's current affinity list: 0-3
...
```

postgres processes There is no binding to the CPU . In the example, processes can be assigned to any available core: 0,1,2,3.

4) Bind the checkpointer process to core 1 :

```
postgres@tantor:~$ taskset -p 12290
pid 12290's current affinity list: 0-3
pid 12290's new affinity list: 1
```

5) Remove the checkpointer process binding to core 1:

```
postgres@tantor:~$ taskset -p -c 0-999 12290
pid 12290's current affinity list: 1
pid 12290's new affinity list: 0-3
```

There is no option to remove the binding, you can only specify a new binding in which to list all cores. The simplest is to specify a range from 0 to the maximum number of cores or higher .

Part 3. Execution context switches (" VCX / ICX ")

postgres processes :

```
postgres@tantor:~$ pidstat -w -l -C postgres
Linux 6.6.28-1-generic (tantor) _x86_64_ (4 CPU)

02:25:37 PM UID PID cswch/s nvcswch/s Command
02:25:37 PM 113 1405 0.10 0.02 /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data
02:25:37 PM 113 1485 0.07 0.01 /usr/lib/postgresql/15/bin/postgres -D /var/lib/postgresql/15/main -c
config_file=/etc/postgresql/15/main/postgresql.conf
02:25:37 PM 113 1586 0.01 0.00 postgres: checkpointer
02:25:37 PM 113 1587 0.39 0.00 postgres: background writer
02:25:37 PM 113 1620 0.02 0.00 postgres: 15/main: logger
02:25:37 PM 113 1693 0.26 0.00 postgres: walwriter
02:25:37 PM 113 1694 0.10 0.01 postgres: autovacuum launcher
02:25:37 PM 113 1695 0.02 0.04 postgres: autoprewarm leader
02:25:37 PM 113 1696    94.35 0.11 postgres: pg_wait_sampling collector
02:25:37 PM 113 1698 0.01 0.00 postgres: logical replication launcher
02:25:37 PM 113 1751 0.01 0.00 postgres: 15/main: checkpointer
02:25:37 PM 113 1752 0.39 0.00 postgres: 15/main: background writer
02:25:37 PM 113 1817 0.26 0.00 postgres: 15/main: walwriter
02:25:37 PM 113 1818 0.05 0.01 postgres: 15/main: autovacuum launcher
02:25:37 PM 113 1819 0.01 0.00 postgres: 15/main: logical replication launcher
02:25:37 PM 0 7653 0.00 0.00 pidstat -w -l -C postgres
```

than an hour or the operating system has been rebooted, **pidstat** will show that the **pg_wait_sampling process collector** Context switches at a rate of ~100 (94.46 in the example) per second. This process collects statistics by polling instance processes. Most polling cycles complete quickly, the process releases the kernel and the cswch statistic is incremented. Jiffy is a unit of time that represents the number of timer interrupts from booting Linux. Each timer interrupt increases the number of Jiffies by one. The relationship between Jiffy and seconds is determined by the HZ constant in the Linux kernel. This value determines the number of Jiffies per second. In Linux before version 2.4, the tick rate between Jifies was 10 ms. Since Linux 2.4, the tick became 1 ms. Since

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

kernel version 3.10, the tick rate varies and can be slowed down by setting CONFIG_NO_HZ_FULL=y when compiling the kernel. In this case, timer interrupts are disabled not only for idle processors, but also during code execution if there are no tasks in the execution queue on the core.

2) View the current number of execution context switches for the process `pg_wait_sampling collector` . At intervals of ~seconds, execute the command:

```
postgres @ tantor :~$ grep ctxt / proc / 1696 / status
voluntary_ctxt_switches: 304 222
nonvoluntary_ctxt_switches: 334
postgres@tantor:~$ grep ctxt /proc/ 1696 /status
voluntary_ctxt_switches: 304 343
nonvoluntary_ctxt_switches: 335
```

of random context switches according to this cumulative statistic is about 100 per second. Involuntary switches are also present. The data matches the result of pidstat . Process `pg_wait_sampling collector` polls the state of all processes in the instance at a frequency specified by the parameter `pg_wait_sampling.history_period` or `pg_wait_sampling.profile_period` . The default values for these parameters are 10 milliseconds, which corresponds to a frequency of 100 times per second.

3) If you restart the instance, then after restarting the `cswch / s values` will gradually increase. Perform V root terminal :

```
root@tantor:~# systemctl restart tantor-se-server-16
root@tantor:~# pidstat -w -l -C postgres | grep sampling
02:36:31 PM 113 7728 4.25 0.00 postgres: pg_wait_sampling collector
```

The averaging period is **from the moment the operating system is started** , which makes the use of the pidstat utility meaningless.
To get a meaningful result, it is worth using cumulative statistics. The value will reach 10 per second with a real frequency of 100 per second in about 7 minutes. At the same time, cumulative statistics shows the frequency correctly 100 times per second. Run the command twice with an interval of about a second:

```
root @ tantor :~# grep ctxt / proc /7728/ status
voluntary_ctxt_switches: 13 433
nonvoluntary_ctxt_switches: 10
root@tantor:~# grep ctxt /proc/7728/status
voluntary_ctxt_switches: 13 551
nonvoluntary_ctxt_switches: 11
```

4) To monitor context switches across the entire operating system, you can use the `perf utility` :

```
root@tantor:~# perf stat -a
^C
Performance counter stats for 'system wide':

68,010.80 msec cpu-clock # 2,000 CPUs utilized
9,885 context-switches # 145.345 /sec
250 cpu-migrations # 3.676 /sec
2,774 page-faults # 40.788 /sec
<not supported> cycles
<not supported> instructions
<not supported> branches
<not supported> branch-misses

      34.005359735 seconds time elapsed
```

The utility with such parameters works until it is stopped by the key combination <ctrl+c>.
The utility gives the correct values: 145 context switches per second. Disadvantage: does not give the number of involuntary context switches.
To collect the same statistics for a single process, use the command:

```
perf stat - p process_number
```

Part 4. Monitoring CPU load

1) Install, if not installed, the atop and htop utilities in the terminal under the root user :

```
root @tantor:~# apt update
Hit:1 cdrom://OS Astra Linux 1.8.1.6 DVD 1.8_x86-64 InRelease
Hit:2 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-extended 1.8_x86-64 InRelease
Ign:3 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-devel 1.8_x86-64 InRelease
Hit:4 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-main 1.8_x86-64 InRelease
Err:5 https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-devel 1.8_x86-64 Release
404 Not Found [IP: 130.193.50.59 443]
Reading package lists... Done
E: The repository 'https://download.astralinux.ru/astra/stable/1.8_x86-64/repository-devel 1.8_x86-64 Release'
does not have a Release file.
N: Updating from such a repository can't be done securely, and is therefore disabled by default.
N: See apt-secure(8) manpage for repository creation and user configuration details.
root@tantor:~# apt install htop -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
htop is already the newest version (3.2.2-2+b1).
0 upgraded, 0 newly installed, 0 to remove and 603 not upgraded.
root@tantor:~# apt install atop -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
atop is already the newest version (2.8.1-1+b1).
0 upgraded, 0 newly installed, 0 to remove and 603 not upgraded.
```

2) Delete the directory if it exists and run the `pg_basebackup` utility in the postgres user terminal , slowing down the backup to 50 kilobytes per second:

```
postgres @tantor:~$ rm -rf /var/lib/postgresql/backup/1
postgres @tantor:~$ time pg_basebackup -c fast -D $HOME/backup/1 -P -r 50k
```

The utility will start the backup. You do not need to close the terminal window with the running utility.

3) In another terminal, as root , run the top command:

```
root @tantor:~# ps -e -o pcpu,vsz,rss,pss,cls,psr,rops,wops,s,cmd --sort -pcpu | head -2
%CPU VSZ RSS PSS CLS PSR ROPS WOPS S CMD
99.9 24608 8576 1486 TS 1 43 8385 R pg_basebackup -c fast -D /var/lib/postgresql/backup/1
-P -r 1M
```

What problem is visible in the command output?
CPU load 99.9%
Next, look at the output of the top, atop, and htop utilities in order to understand which utility is more convenient for identifying the problem of processor load.

4) Run the top utility:

```
root @ tantor :~# top
```

```
1 — Terminal
File  Edit  View  Bookmarks  Plugins  Settings  Help

top - 16:38:03 up  1:00,  3 users,  load average: 1.07, 0.99, 0.85
Tasks: 209 total,   2 running, 207 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25.5 us,  0.5 sy,  0.0 ni, 72.9 id,  1.2 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3926.6 total,    665.2 free,   1033.2 used,   2637.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2893.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   7905 postgres  20   0   24608   8576   7296 R 100.0   0.2   6:01.16 pg_basebackup
   5659 fly-dm    -2   0  404028 154236  65664 S   1.0   3.8   1:03.84 Xorg
     82 root      20   0       0      0      0 I   0.3   0.0   0:00.99 kworker/2:1-events
   1374 docker-+  20   0 1694240  23208  16640 S   0.3   0.6   0:00.73 docker-registry
   5872 astra     20   0  151264   3080   2816 S   0.3   0.1   0:27.25 VBoxClient
   5967 astra     20   0   18672   7936   6912 S   0.3   0.2   0:11.45 compton
```

What CPU load does the utility show?
In the %CPU column the utility shows 100.0.
In the %Cpu(s) field: 25.5 us, 0.5 sy .

5) Press the 1 key on the keyboard. The utility output will change, it will show the load separately by processor cores:

```
1 — Terminal
File  Edit  View  Bookmarks  Plugins  Settings  Help

top - 16:39:45 up  1:01,  3 users,  load average: 1.01, 1.00, 0.87
Tasks: 209 total,   3 running, 206 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.3 us,  0.7 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  1.0 us,  0.0 sy,  0.0 ni, 99.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  1.0 us,  0.3 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3926.6 total,    542.8 free,   1050.5 used,   2743.9 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2876.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   7905 postgres  20   0   24608   8576   7296 R  99.7   0.2   7:43.95 pg_basebackup
   5659 fly-dm    -2   0  404028 154236  65664 S   1.7   3.8   1:04.72 Xorg
   6322 astra     20   0  618900  85020  67724 S   1.0   2.1   0:05.33 fly-term
```

Load on 3 core: %Cpu3: 100.0 us, 0.0 sy and this is the real load.
This means that the third core is fully loaded, and the USER / SYS ratio is 100:0 .

Look at the `load metric on the top right average : 1.01, 1.00, 0.87`
The metric is also present in the `/ proc / loadavg file` and the output of the `uptime command` :

```
root@tantor:~# uptime
09:39:07 up 18:01, 3 users, load average: 0.19, 0.20, 0.18
```

This metric shows the number of active (working or willing to work) processes, averaged over intervals of 1, 5, 15 minutes. The metric takes into account processes that consume processor time and are waiting for a result from the disk subsystem. Processes that are waiting for data from network interfaces are not taken into account in the metric.
If the number of active processes is less than the number of cores, then if all processes are active, they will not experience a shortage of computing resources. But the number of processes is usually greater than the number of cores. If a large number of active processes appears, then access to the cores is divided between the active processes. When the DBMS is running, the processor is usually not a bottleneck (a scarce resource) and the metric does not indicate any problems. The metric can be useful in that by comparing three numbers you can assess whether the current load is peak or constant. If you create a large number of active processes, for example, by running the pgbench utility in a separate terminal with a number of sessions of 90:

```
postgres@tantor:~$ pgbench -c 90 -j 90 -T 10000 -P 5
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 892.4 tps, lat 105.630 ms stddev 87.005, 0 failed
progress: 10.0 s, 413.0 tps, lat 237.783 ms stddev 257.410, 0 failed
...
```

By default, the configuration parameter `max_connections=100` , so the number of sessions is selected to be 90.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Next, you can look at the output of the top utility for all processors and individually, and you will see that if the load exceeds the number of cores, then the total load on the processor becomes useful. If the total load %Cpu(s) is greater than 90%, then the processor is a bottleneck. With a long-term load, the 90% limit (for the purpose of determining resource scarcity) can vary from 85% to 95% depending on the share of disk I/O waiting.

Example of total load - there is a **`%Cpu(s) metric`** :

```
1 — Terminal
File  Edit  View  Bookmarks  Plugins  Settings  Help

top - 10:49:54 up 19:12,  3 users,  load average: 8.49, 10.10, 10.35
Tasks: 309 total,   3 running, 306 sleeping,   0 stopped,   0 zombie
%Cpu(s): 25.6 us, 21.7 sy,  0.0 ni, 28.6 id, 23.8 wa,  0.0 hi,  0.3 si,  0.0 st
MiB Mem :   3926.6 total,    213.6 free,   1298.4 used,   2871.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2628.2 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    269 root      20   0  299232 167996 166332 R  77.2   4.2  28:55.99 systemd-journal
   1896 root      20   0 1634952  75720  54944 S  26.5   1.9   4:44.64 syslog-ng
  32001 postgres  20   0 2864188  12928   6912 S   9.6   0.3   4:15.56 pgbench
   1680 postgres  20   0  231516 142188 139520 D   1.3   3.5   0:41.01 postgres
  32139 postgres  20   0  233772 153708 150144 S   1.3   3.8   0:18.14 postgres
  32159 postgres  20   0  233756 153324 149632 S   1.3   3.8   0:18.02 postgres
   1813 postgres  20   0  232344   9708   6784 S   1.0   0.2  41:57.27 postgres
   5659 fly-dm    -2   0  404560 153468  64384 S   1.0   3.8   1:58.95 Xorg
   5951 astra     20   0  257.3g 138276 107024 S   1.0   3.4  32:52.97 fly-start-menu
  32109 postgres  20   0  233752 153196 149632 S   1.0   3.8   0:17.85 postgres
  32116 postgres  20   0  233748 153324 149632 S   1.0   3.8   0:18.09 postgres
```

Average load across all cores: 100-28.6=71.4% the processor is not a bottleneck.
However, if we reduce the number of server processes from 90 to ~50 for four cores in this example, then the maximum tps will be achieved. With a large load of processors, context switching, and under a very large wait for receiving timeslots on the processor core, a delay is added. The overhead of context switching (replacing the contents of kernel caches) cannot be measured directly.

Example of processor load - metrics **`%Cpu0`** ... **`%Cpu 3`** :

```
1 — Terminal
File  Edit  View  Bookmarks  Plugins  Settings  Help

top - 10:51:13 up 19:13,  3 users,  load average: 9.37, 9.62, 10.14
Tasks: 310 total,   1 running, 309 sleeping,   0 stopped,   0 zombie
%Cpu0  : 18.2 us, 13.3 sy,  0.0 ni, 31.5 id, 36.0 wa,  0.0 hi,  1.0 si,  0.0 st
%Cpu1  : 34.5 us, 29.4 sy,  0.0 ni, 30.7 id,  5.1 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu2  : 13.4 us, 14.4 sy,  0.0 ni, 48.3 id, 24.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  : 17.6 us, 12.8 sy,  0.0 ni, 29.8 id, 39.8 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3926.6 total,    117.2 free,   1308.5 used,   2957.4 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2618.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    269 root      20   0   85456  39540  37876 S  60.6   1.0  29:38.35 systemd-journal
  32001 postgres  20   0 2864188  12928   6912 S  10.3   0.3   4:21.92 pgbench
   1680 postgres  20   0  231516 142188 139520 D   1.3   3.5   0:41.97 postgres
   1813 postgres  20   0  232344   9708   6784 S   1.3   0.2  41:58.64 postgres
  32105 postgres  20   0  233744 153708 150016 S   1.3   3.8   0:18.31 postgres
  32124 postgres  20   0  233724 153836 150144 S   1.3   3.8   0:18.33 postgres
  32142 postgres  20   0  233752 153708 150016 S   1.3   3.8   0:18.46 postgres
  32144 postgres  20   0  233752 153580 150016 S   1.3   3.8   0:18.38 postgres
  32145 postgres  20   0  233748 153964 150272 S   1.3   3.8   0:18.34 postgres
```

The processor load is useful when the load is less than the number of cores and to determine whether there are processes that load the core by 100% and for this process (a thread that cannot be paired and serviced by several cores) the processor is a bottleneck.

Close the top utility by pressing the `q key on your keyboard .`

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

If pgbench is running , it can be stopped with the key combination < ctrl + c >.

6) Run the mpstat utility :

```
root @ tantor :~# mpstat - n
Linux 6.6.28-1-generic (tantor) _x86_64_ (4 CPU)

08:40:00 PM NODE %usr %nice %sys %iowait %irq %soft %steal %guest %gnice %idle
08:40:00 PM  all    5.71 0.05 0.39 0.42 0.00 0.46 0.00 0.00 0.00 92.97
```

See if this utility is convenient for monitoring the load of one core at 100% and the USER / SYS ratio ?

7) Run the a top utility:

```
root @ tantor :~# atop
```

```
1 — Terminal
File  Edit  View  Bookmarks  Plugins  Settings  Help

ATOP - tantor                         2024/12/04  20:24:36                  -----------------                      4h46m43s elapsed
PRC | sys    8m19s | user  11m35s |              | #proc    209 | #trun      5 | #tslpi   292 | #tslpu    67 | #zombie    0 | clones 13707 |              | #exit      0 |
CPU | sys       2% | user     20% | irq       2% |              | idle    374% | wait      2% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       1% | user     10% | irq       2% |              | idle     86% | cpu000 w  1% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       0% | user      7% | irq       0% |              | idle     93% | cpu001 w  0% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       0% | user      4% | irq       0% |              | idle     95% | cpu003 w  1% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       0% | user      1% | irq       0% |              | idle     98% | cpu002 w  0% |              | steal     0% | guest     0% |              | curf 3.60GHz |
CPL | numcpu     4 |              |              | avg1    0.64 | avg5    0.36 | avg15   0.45 |              |              | csw  9694625 |              | intr 8506941 |
MEM | tot     3.8G | free  963.4M | cache   2.1G | dirty  22.2M | buff   94.8M |              | slab  124.1M | slrec  66.4M |              | pgtab  15.4M |              |
MEM | numnode    1 |              | shmem 171.6M | shrss   0.0M | shswp   0.0M |              |              | tcpsk   0.0M | udpsk   0.0M |              |              |
SWP | tot     0.0M | free    0.0M |              | swcac   0.0M |              |              |              |              | vmcom   2.5G |              | vmlim   1.9G |
PAG | scan  102472 | steal 101140 | stall      0 | compact    0 | numamig    0 | migrate    0 | pgin  515637 | pgout 1345e3 | swin       0 | swout      0 | oomkill    0 |
PSI | cpusome   2% | memsome   0% | memfull   0% | iosome    0% | iofull    0% | cs     0/1/2 | ms     0/0/0 | mf     0/0/0 | is     0/0/0 | if     0/0/0 |              |
DSK |          sda | busy      1% | read   52362 | write  56657 | discrd     0 | KiB/r     39 | KiB/w     94 | KiB/d      0 | MBr/s    0.1 | MBw/s    0.3 | avio 1.78 ms |
NET | transport    | tcpi     103 | tcpo      89 | udpi      34 | udpo      34 | tcpao     14 | tcppo      0 | tcprs      0 | tcpie      0 | tcpor     12 | udpie      0 |
NET | network      | ipi      142 | ipo      152 | ipfrw      0 | deliv    139 |              |              |              |              | icmpi      2 | icmpo     23 |
NET | enp0s3    0% | pcki     298 | pcko     119 | sp 1000 Mbps | si    0 Kbps | so    0 Kbps | mlti       0 | erri       0 | erro       0 | drpi       0 | drpo       0 |
NET | lo      ---- | pcki      34 | pcko      34 | sp    0 Mbps | si    0 Kbps | so    0 Kbps | mlti       0 | erri       0 | erro       0 | drpi       0 | drpo       0 |
*** System and Process Activity since Boot ***                                                                               Unrestricted view (privileged)
  PID   SYSCPU   USRCPU   RDELAY   BDELAY    VGROW    RGROW    RDDSK    WRDSK  RUID      EUID      ST  EXC   THR  S  CPUNR   CPU  CMD          1/10
 1813    3m16s    3m10s    3m48s    0.00s   226.5M     9.4M       0B       0B  postgres  postgres  N-    -     1  S      3    2%  postgres
 5951    2m18s    3m17s    2m52s    0.00s   257.3G   136.5M    37.4M       0B  astra     astra     N-    -    14  S      3    2%  fly-start-menu
 5872   83.75s   95.08s    3m10s    0.00s   147.7M     3.0M       0B     4.0K  astra     astra     N-    -     4  S      0    1%  VBoxClient
 5659   11.11s   65.38s    0.28s    0.00s   394.6M   150.6M    13.4M    36.0K  fly-dm    fly-dm    N-    -     2  S      2    0%  Xorg
13689    0.30s   44.78s    0.17s    0.00s    24.0M     8.4M       0B    45.5M  postgres  postgres  N-    -     1  R      0    0%  pg_basebackup
 1896    5.98s   25.05s    5.46s    0.00s     1.6G    69.1M    15.0M     5.9M  root      root      N-    -    13  S      1    0%  syslog-ng
 1382    6.51s   12.22s    9.71s    0.00s     1.4G    49.1M    44.3M     4.0K  root      root      N-    -    10  S      3    0%  containerd
```

Knowing the real load, find it in the `atop utility window` CPU load given by `the pg_basebackup utility .`
After 10 seconds the window will change and display different numbers:

```
1 — Terminal
File  Edit  View  Bookmarks  Plugins  Settings  Help

ATOP - tantor                         2024/12/04  16:41:01                  -----------------                           10s elapsed
PRC | sys    0.26s | user  11.05s |              | #proc    209 | #trun      4 | #tslpi   293 | #tslpu    67 | #zombie    0 | clones     1 |              | #exit      1 |
CPU | sys       2% | user    110% | irq       0% |              | idle    283% | wait      6% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       0% | user    100% | irq       0% |              | idle      0% | cpu003 w  0% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       1% | user      7% | irq       0% |              | idle     93% | cpu002 w  0% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       1% | user      2% | irq       0% |              | idle     93% | cpu000 w  4% |              | steal     0% | guest     0% |              | curf 3.60GHz |
cpu | sys       0% | user      1% | irq       0% |              | idle     98% | cpu001 w  1% |              | steal     0% | guest     0% |              | curf 3.60GHz |
CPL | numcpu     4 |              |              | avg1    1.00 | avg5    1.00 | avg15   0.89 |              |              | csw    11940 |              | intr   10278 |
MEM | tot     3.8G | free  463.7M | cache   2.6G | dirty  21.8M | buff   83.5M |              | slab  136.7M | slrec  78.5M |              | pgtab  15.4M |              |
MEM | numnode    1 |              | shmem 171.0M | shrss   0.0M | shswp   0.0M |              |              | tcpsk   0.0M | udpsk   0.0M |              |              |
SWP | tot     0.0M | free    0.0M |              | swcac   0.0M |              |              |              |              | vmcom   2.5G |              | vmlim   1.9G |
PAG | scan       0 | steal      0 | stall      0 | compact    0 | numamig    0 | migrate    0 | pgin      29 | pgout   2146 | swin       0 | swout      0 | oomkill    0 |
PSI | cpusome   0% | memsome   0% | memfull   0% | iosome    1% | iofull    1% | cs     0/0/0 | ms     0/0/0 | mf     0/0/0 | is     1/0/0 | if     0/0/0 |              |
DSK |          sda | busy      7% | read       2 | write   1249 | discrd     0 | KiB/r     58 | KiB/w      6 | KiB/d      0 | MBr/s    0.0 | MBw/s    0.8 | avio 0.56 ms |

  PID   SYSCPU   USRCPU   RDELAY   BDELAY    VGROW    RGROW    RDDSK    WRDSK  RUID      EUID      ST  EXC   THR  S  CPUNR   CPU  CMD           1/3
 7905    0.03s    9.98s    0.00s    0.00s       0B       0B       0B    10.0M  postgres  postgres  --    -     1  R      3  100%  pg_basebackup
 1896    0.04s    0.25s    0.95s    0.00s       0B   512.0K       0B       0B  root      root      --    -    13  S      0    3%  syslog-ng
 5659    0.04s    0.22s    0.00s    0.00s       0B       0B       0B       0B  fly-dm    fly-dm    --    -     2  S      2    3%  Xorg
  269    0.02s    0.19s    0.00s    0.00s       0B     1.4M   116.0K     4.5M  root      root      --    -     1  R      0    2%  systemd-journa
 6322    0.00s    0.17s    0.03s    0.00s       0B       0B       0B       0B  astra     astra     --    -     3  S      2    2%  fly-term
 7987    0.01s    0.09s    0.00s    0.00s       0B       0B       0B       0B  root      root      --    -     1  R      2    1%  atop
 5967    0.03s    0.02s    0.23s    0.00s       0B       0B       0B       0B  astra     astra     --    -     1  S      2    1%  compton
  427    0.00s    0.04s    0.01s    0.00s       0B       0B       0B   336.0K  root      root      --    -     2  S      1    0%  auditd
 5951    0.01s    0.02s    0.00s    0.00s       0B       0B       0B       0B  astra     astra     --    -    14  S      0    0%  fly-start-menu
 1813    0.01s    0.02s    0.00s    0.00s       0B       0B       0B       0B  postgres  postgres  --    -     1  S      0    0%  postgres
   62    0.03s    0.00s    0.00s    0.00s       0B       0B       0B       0B  root      root      --    -     1  I      0    0%  kworker/0:1H-k
```

The displayed data is updated by default every 10 seconds.
Note for yourself: is it convenient to view the load and is the US E R/SYS ratio visible.
Close the utility by pressing the q key on your keyboard.

8) Run the htop utility :

```
root @ tantor :~# htop
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

:8.8 — Подключение к удаленному рабочему столу

Astralinux 1.8.1 [Работает] - Oracle VM VirtualBox

Вид Ввод Устройства Справка

1 – Terminal

File Edit View Bookmarks Plugins Settings Help

```
    0[|                                                          0.6%] Tasks: 108, 154 thr, 102 kthr; 2 running
    1[|||||                                                      4.5%] Load average: 1.08 1.02 0.90
    2[|||                                                        2.5%] Uptime: 01:03:57
    3[||||||||||||||||||||||||||||||||||||||||||||||||||||||||100.0%]
  Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||652M/3.83G]
  Swp[                                                          0K/0K]

 Main  I/O
  PID USER       PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
 7905 postgres    20   0 24608  8576  7296 R 100.0  0.2  9:48.95 pg_basebackup -c fast -D /var/lib/postgresql/backup/1 -P -r 1M
 8000 root        20   0  9200  4992  3584 R   3.2  0.1  0:00.28 htop
 5659 fly-dm      -2   0  394M  150M 65664 S   1.3  3.8  1:07.35 /usr/lib/xorg/Xorg -br -novtswitch -quiet -keeptty :0 vt7 -logfile /var/log/fly-dm/Xorg.%s.log -seat seat0 -aut
 1813 postgres    20   0  226M  9580  6784 S   0.6  0.2  1:01.58 postgres: pg_wait_sampling collector
 5722 fly-dm      20   0  394M  150M 65664 S   0.6  3.8  0:00.60 /usr/lib/xorg/Xorg -br -novtswitch -quiet -keeptty :0 vt7 -logfile /var/log/fly-dm/Xorg.%s.log -seat seat0 -aut
 6322 astra       20   0  604M 84468 67724 S   0.6  2.1  0:06.79 fly-term
    1 root        20   0  168M 15276 10796 S   0.0  0.4  0:02.40 /sbin/init splash
  269 root        20   0  231M  116M  115M S   0.0  3.0  0:05.65 /lib/systemd/systemd-journald
  300 root        20   0 32464  8932  6372 S   0.0  0.2  0:00.39 /lib/systemd/systemd-udevd
  426 systemd-re  20   0 25544 14720 12288 S   0.0  0.4  0:00.11 /lib/systemd/systemd-resolved
  427 root        16  -4 13436  2176  1920 S   0.0  0.1  0:00.95 /sbin/auditd
  428 root        16  -4 13436  2176  1920 S   0.0  0.1  0:00.04 /sbin/auditd
```

Note for yourself: is it convenient to view the load and is the US E R/SYS ratio visible in the h top utility.
The load on the processor cores is shown as pseudo-graphics symbols.
Close the utility by pressing the q key on your keyboard.
The goal of this part of the practice is to choose a utility that will make it easier to view the load on processor cores.

Part 5. Collecting statistics into a file and viewing it with the `atop utility`

1) The atop utility allows you to collect and write statistics to a binary file. The file can be "played" on another host, which is convenient for analysis. The minimum frequency of statistics collection is once per second, by default once every 10 seconds. The volume of collected data is large, but you can use parameters to configure which statistics to collect to reduce the file size.

2) Start collecting statistics at 1 second intervals to the file /atop.record :

```
root@tantor:~# atop -w /atop.record 1
```

The terminal will not return a prompt, this is normal. If the file existed, the utility will not erase the collected data, but will start writing data to the end of the file.

3) In a separate root user terminal, run the atop utility in binary file viewing mode:

```
astra@tantor:~$ atop -r /atop.record
```

Use the keys on the keyboard ' t ' and 'T' (<Shift+t>) to go to the next and return to the previous interval. The first line issued by the utility shows the time for which statistics were collected. To exit the utility, press the key 'q'.

4) In the terminal where the utility is running, press < ctrl+c >:

```
root@tantor:~# atop -w /atop.record 1
^C
root@tantor:~#
```

The utility will stop. The file can be viewed again with the atop utility . You can copy the file to another host and view the file on the other host.

5) Delete the file `/atop.record` :

```
root @ tantor :~# rm - f / atop . record
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

6) If the `pg_basebackup utility` has not finished the backup, interrupt the utility by pressing the < ctrl + c > key combination on the keyboard:

```
postgres@tantor:~$ time pg_basebackup -c fast -D $HOME/backup/1 -P -r 50k
^C 3781/902114 kB (68%), 0/1 tablespace

real 19m57.486s
user 19m53.055s
sys 0m3.413s
```

Note that the time utility returned the correct US E R/SY S values .

7) Delete the directory with the backup:

```
postgres @ tantor :~$ rm - rf / var / lib / postgresql / backup /1
```

Part 6. Linux Time Source

1) Look at the list of sources that Linux considered possible to use:

```
postgres@tantor:~$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc acpi_pm
```

2) Look at what time source is used:

```
postgres@tantor:~$ cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

3) Create a text file named `clock_timing.c with the following` contents:

```
#include <time.h>
int main()
{
int rc;
long i;
struct timespec ts;
for(i=0; i< 10000000 ; i++) rc = clock_gettime(CLOCK_MONOTONIC, &ts);
return 0;
}
```

4) Compile file :

```
 postgres @tantor:~$ gcc clock_timing.c -o clock_timing -lrt
```

5) The program reads the time indicator 10 million times. Measure the execution time of the program:

```
postgres @ tantor :~$ time ./ clock _ timing
real 0m13,967s
user 0m13,938s
sys  0m0,008s
```

time utility is useful for getting the execution time of programs. It returns the actual US E R/SYS and the total execution time of the program.

6) Run `the pg_test_timing command :`

```
postgres@tantor:~$ pg_test_timing
```

Oleg Ivanov

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 3931.22 ns
Histogram of timing durations:
< us % of total count
     1 0.02475 497
2 60.52305 1215427
4 39.20322 787281
8 0.02480 498
16 0.08869 1781
32 0.06384 1282
64 0.05348 1074
128 0.01215 244
256 0.00388 78
512 0.00065 13
1024 0.00060 12
2048 0.00030 6
4096 0.00035 7
8192 0.00020 4
16384 0.00005 1
```

The `pg_test_timing` program comes standard with PostgreSQL and is used to measure the speed of a time source, and returns the distribution of values returned by the `time source` .
Distribution maximum at ~2.5 milliseconds.
The symbols " `us` " denote milliseconds (`µs`). The fluctuations in the time output in the utility output are not less than 1 millisecond. This means that numbers less than 1 millisecond are random.
`pg_test_timing` program provides more detailed information than `clock_timing.c` . The clock_timing.c program `is provided to illustrate the ease of creating your own tests in C`. The PostgreSQL code is written in `C`.

7) Create a table for the test by running the following commands in psql:

```
postgres=# drop table if exists t;
create table t(pk bigserial, c1 text default
'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');
insert into t select *, 'a' from generate_series(1, 100000);
DROP TABLE
CREATE TABLE
INSERT 0 100000
```

8) Run the command:

```
postgres=# explain analyze select count(pk) from t;
QUERY PLAN
-------------------------------------------------------------------------------
Aggregate (cost=1352.50..1352.51 rows=1 width=8) (actual time=307.270..307.276 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1190.20 rows=64920 width=8) (actual time=0.017..152.937 rows=100000 loops=1)
Planning Time: 0.038 ms
Execution Time: 307 . 383 ms
(4 rows)
```

Command execution time is 307 milliseconds.

9) Replace the time source tsc on acpi_pm . In the root console , run the command:

```
root@tantor:~# echo acpi_pm >/sys/devices/system/clocksource/clocksource0/current_clocksource
```

The command can be executed without switching to the root console:

```
postgres@tantor:~$
sudo sh -c 'echo acpi_pm>/sys/devices/system/clocksource/clocksource0/current_clocksource'
```

10) Run the command:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres=# explain analyze select count(pk) from t;
QUERY PLAN
------------------------------------------------------------------------------------
Aggregate (cost=1352.50..1352.51 rows=1 width=8) (actual time=805.557..805.573 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1190.20 rows=64920 width=8) (actual time=0.015..399.513 rows=100000 loops=1)
Planning Time: 0.148 ms
Execution Time: 805 . 792 ms
(4 rows)
```

Command execution time has been increased from 0.3 to 0.8 seconds.
For the duration of a simple `explain command analyze` was significantly affected by the change in time source.

11) Measure the execution time of the program:

```
postgres @ tantor :~$ time ./ clock _ timing

real 0m38.669s
user 0m16.329s
sys 0 m 22.331 s
```

The execution time has increased significantly - almost 3 times. The USER/SYS ratio has changed from 1:1 to 8:11. It can be seen that the USER/SYS ratio can be used to identify inefficient code. A skewed ratio towards SYS (deviation from the 60:40 ratio) indicates that the kernel code that is issuing the time is inefficient.

12) Run `the pg_test_timing command :`

```
postgres@tantor:~$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 3889.59 ns
Histogram of timing durations:
< us % of total count
     1 0.00389 30
2 0.00052 4
4 25.33700 195422
8 74.23528 572570
16 0.21470 1656
32 0.08376 646
64 0.10411 803
128 0.01776 137
256 0.00117 9
512 0.00013 1
1024 0.00052 4
2048 0.00052 4
4096 0.00039 3
8192 0.00000 0
16384 0.00026 2
```

The maximum distribution of the timestamp output speed with the `acpi_pm time source` was ~ 7 milliseconds, and with the tsc time source it was ~ 2.5 milliseconds.

13) Verify source time tsc :

```
root@tantor:~# echo tsc >/sys/devices/system/clocksource/clocksource0/current_clocksource
```

or

```
postgres@tantor:~$
sudo sh -c 'echo tsc >/sys/devices/system/clocksource/clocksource0/current_clocksource'
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

By default, the time source is selected by Linux every time it boots. There is a possibility that a slow time source will be selected arbitrarily. This can also happen after updating the Linux kernel. The probability is increased by: failure to disable power saving in BIOS, enabling Hyper Threading in BIOS, any hardware activity during the time source testing, for example, Intel ME activity. **The probability of an arbitrary time source replacement is high on virtual machines, since the hardware load is uneven when loading a virtual machine.** It is recommended to check which time source is used or to fix the desired time source in the Linux kernel boot parameters. An example of changing `/ boot / grub / grub . cfg` is given in the theoretical part of the course. The fastest time source is tsc .

Part 7. Network connections

1) Open a root user terminal or switch to the root console:

```
postgres @ tantor :~$ su -
Password : root
```

2) Check if the slow start algorithm after idle time is enabled:

```
root@tantor:~# sysctl -a | grep net.ipv4.tcp_slow_start_after_idle
net.ipv4.tcp_slow_start_after_idle = 1
```

The slow start algorithm after idle time is enabled. If the session was idle and there was no network traffic, then in networks with noticeable network latency (network latency) the data transfer rate will increase gradually, rather than immediately using the network bandwidth. The first three TCP packets are transmitted without delay. This is usually enough to transmit commands and small results. Most OLTP systems do not experience slowdowns, since the request and result fit into three network packets.

3) Look at what algorithm is used to control the load on the network data transmission channel:

```
root@tantor:~# sysctl -a | grep net.ipv4.tcp_congestion_control
net.ipv4.tcp_congestion_control = cubic
```

cubic algorithm is used . This is an algorithm that sets the volume of data transferred by measuring the loss of TCP packets, which is undesirable and leads to retransmission of packets. The probability of transferring large amounts of data, using a network with a large network delay is small, so the choice of algorithm is rarely paid attention to and the network is not a bottleneck for DBMS and their clients. However, if large amounts of data are transferred over a network channel, and this is streaming replication and backup, then their efficiency may be less than it could be.

4) Look at the parameters that determine keeping the socket open:

```
root@tantor:~# sysctl -a | grep keepalive
net.ipv4.tcp_keepalive_intvl = 75
net.ipv4.tcp_keepalive_probes = 9
net.ipv4.tcp_keepalive_time = 7200
```

These are default values and can be reduced.
In addition to these parameters, the values of the `net.ipv4.tcp_retries2 parameter` and the previously mentioned `net.ipv4.tcp_slow_start_after_idle parameter are changed .`

Part 8. Replacing the scheduling policy and checking the scheduler operation

1) Create a file to test the CPU load:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
astra@tantor:~$ mcedit i.sh

#!/bin/sh
while true; do echo 1 > /dev/null ; done;
```

2) Check that the file contents have been saved successfully:

```
astra@tantor:~$ cat i.sh
#!/bin/sh
while true; do echo 1 > /dev/null ; done;
```

3) Launch processes equal to the number of virtual machine processor cores. An example is given for four cores:

```
astra@tantor:~$ nproc
4
astra @ tantor :~$ taskset ./ i . sh &
[2] 212239
astra@tantor:~$ taskset./i.sh &
[3] 212242
astra@tantor:~$ taskset./i.sh &
[4] 212243
astra@tantor:~$ taskset./i.sh &
[5] 212244
```

4) Two processes load both cores by 100%. This can be checked with the top utility.
Make sure that the pidstat utility is misleading, showing a small load:

```
root@tantor:~# pidstat -w | grep i.sh
06:58:52 AM 1000 212239 0.00 0.02 i.sh
06:58:52 AM 1000 212242 0.00 0.02 i.sh
06:58:52 AM 1000 212243 0.00 0.02 i.sh
06:58:52 AM 1000 212244 0.00 0.02 i . sh
```

5) Look at what the vmstat utility shows:

```
root@tantor:~# vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
rb swpd free buff cache si so bi bo in cs us sy id wa st
 7 0 0 179276 271824 720548 0 0 2 3 75 78 2 0 98 0 0
```

vmstat utility shows no load and a small number of context switches.
Given that the number of switches is quite large.

Run the command at intervals of about one second:

```
astra@tantor:~$ grep ctxt /proc/22242/status
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 3841
astra@tantor:~$ grep ctxt /proc/22242/status
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 3945
```

Contexts are switched at a rate of 100 times per second.

6) Set the SCED_RR (Round Robin Scheduler) scheduler policy for the processes - the timeslice allocated to the processes is the same and equal to the value of `the kernel.sched_rr_timeslice_ms parameter :`

```
astra@tantor:~# sudo chrt -r -p 10 212242
astra@tantor:~# sudo chrt -r -p 10 212239
astra@tantor:~# sudo chrt -r -p 10 212243
astra@tantor:~# sudo chrt -r -p 10 212244
```

Processes with the SCHED_DEADLINE policy can preempt processes with SCHED_FIFO and SCHED_RR, that is, they have the highest priority.

7) Check the result of the command:

```
astra @ tantor :~$ chrt - p 212242
pid 212242's current scheduling policy: SCHED_RR
pid 212242's current scheduling priority: 10
```

8) Execute the command at intervals of about one second:

```
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 27518
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 27523
```

Contexts began to switch at a frequency of 10 times per second

9) Look at the RR scheduler timeslice :

```
astra @ tantor :~$ sudo sysctl - a | grep kernel . sched _ rr _ timeslice _ ms
kernel.sched_rr_timeslice_ms = 100
```

timeslice 100 milliseconds = 1/10 of a second, which corresponds to switching 10 times per second.

10) Set timeslice to one second:

```
astra@tantor:~# sudo sysctl kernel.sched_rr_timeslice_ms=1000
kernel.sched_rr_timeslice_ms = 1000
```

The frequency of context switches will not be more than 100 times per second regardless of the settings. This is correct, as it allows not to produce large overhead costs for the scheduler's work on context switching.

11) Check that the process contexts are c RR policy began to switch once per second. Run commands with an interval of once per second and see how the values of the forced context switch counter change:

```
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 62348
root@tantor:~# grep ctxt /proc/212233/status
voluntary_ctxt_switches: 0
nonvoluntary_ctxt_switches: 62349
```

The values will increase by one at a rate of once per second.

12) Run the command and after about 20 seconds interrupt its execution by typing < ctrl + c >:

```
root@tantor:~# perf stat -p 212233
^C
Performance counter stats for process id '22381':

9,340.10 msec task-clock:u # 0.477 CPUs utilized
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
0 context-switches:u # 0.000 /sec
0 cpu-migrations:u # 0.000 /sec
0 page-faults:u # 0.000 /sec
<not supported> cycles:u
<not supported> instructions:u
<not supported> branches:u
<not supported> branch-misses:u

      19.595645993 seconds time elapsed
```

`perf` utility gives the correct data of zero arbitrary context switches per second . The number of forced switches, which are more valuable for diagnostics, is not shown, which is misleading.
The top utility shows the correct data with sufficient accuracy for a context switching frequency of 1 second: the load of processor cores reaches 100%. The total load of all cores %CPU is ~50% for each i.sh process.

```
top - 23:04:28 up  3:03,  3 users,  load average: 4.08, 3.65, 2.91
Tasks: 173 total,   6 running, 167 sleeping,   0 stopped,   0 zombie
%Cpu0  :  1.0 us,  0.7 sy,  0.0 ni, 98.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  : 94.0 us,  1.0 sy,  0.0 ni,  5.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3927.2 total,   1439.9 free,   1005.2 used,   1898.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2922.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  22381 astra    -11   0    7580   3200   2944 R  58.8   0.1  10:25.68 i.sh
  22380 astra    -11   0    7580   3200   2944 R  35.9   0.1  10:30.74 i.sh
     16 root      20   0       0      0      0 R   0.7   0.0   4:52.74 rcu_preempt
   6420 astra     20   0  257.0g 139696 108444 S   0.7   3.5   1:58.75 fly-start-menu
```

13) Stop the processes:

```
astra @ tantor :~$ killall i . sh
[1]- Terminated taskset./i.sh
[2]+ Terminated taskset./i.sh
```

In this part of the practice, you learned how to test the result of changing the scheduler parameters. Using the example given, you can test the work of the schedulers `SCHED _ OTHER , SCHED _ FIFO , SCHED _ DEADLINE .`
`SCHED_OTHER` policy (and its derivatives `SCHED_BATCH, SCHED_IDLE`) are not real-time policies. **Real-time policies** (for interactive tasks where responsiveness is important - getting timeslices with some frequency) include: `SCHED_FIFO` - preempted only by processes with higher priority or the `SCHED_DEADLINE policy` . Setting the `SCHED_FIFO policy to a process` results in the process occupying the core for a long time.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Practice for Chapter 4

Part 1. Disk subsystem parameters

1) Block devices are located in the /dev directory mounted on the devtmpfs virtual file system. See which devices from /dev are mounted:

```
root@tantor:~# mount | grep /dev
udev on /dev type devtmpfs
(rw,nosuid,relatime,size=1966296k,nr_inodes=491574,mode=755,inode64)
devpts on /dev/pts type devpts
(rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
/dev/sda1 on / type ext4 (rw,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,inode64)
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,pagesize=2M)
mqueue on /dev/mqueue type mqueue (rw,nosuid,nodev,noexec,relatime)
```

/dev contains files only for those devices that are currently available (connected). If a device is disconnected, the file is removed from /dev.
/ dev / sda is mounted in the virtual machine .

2) Look at the list of block devices:

```
root @ tantor :~# ls - l / dev | grep br
brw-rw---- 1 root disk 7, 0 loop0
brw-rw---- 1 root disk 7, 1 loop1
brw-rw---- 1 root disk 7, 2 loop2
brw-rw---- 1 root disk 7, 3 loop3
brw-rw---- 1 root disk 7, 4 loop4
brw-rw---- 1 root disk 7, 5 loop5
brw-rw---- 1 root disk 7, 6 loop6
brw-rw---- 1 root disk 7, 7 loop7
brw-rw---- 1 root disk 259, 0 nvme0n1
brw-rw---- 1 root disk 8, 0 sda
brw-rw---- 1 root disk 8, 1 sda1
brw-rw----+ 1 root cdrom 11, 0 sr0
```

The virtual machine has a virtual disk /dev/sda connected to it with one partition /dev/sda1.
The first letter b stands for block device. Instead of the file size, two numbers are given: the type and serial number (or operating mode) of the device.

3) Look at the contents of the `/ sys / dev / block directory` :

```
root@tantor:~# ls -al /sys/dev/block
total 0
drwxr-xr-x 2 root root 0 .
drwxr-xr-x 4 root root 0 ..
lrwxrwxrwx 1 root root 0 11:0 ->
../../devices/pci0000:00/0000:00:01.1/ata2/host1/target1:0:0/1:0:0:0/block/sr0
lrwxrwxrwx 1 root root 0 259:0 ->
../../devices/pci0000:00/0000:00:0e.0/nvme/nvme0/nvme0n1
lrwxrwxrwx 1 root root 0 7:0 -> ../../devices/virtual/block/loop0
lrwxrwxrwx 1 root root 0 7:1 -> ../../devices/virtual/block/loop1
lrwxrwxrwx 1 root root 0 7:2 -> ../../devices/virtual/block/loop2
lrwxrwxrwx 1 root root 0 7:3 -> ../../devices/virtual/block/loop3
lrwxrwxrwx 1 root root 0 7:4 -> ../../devices/virtual/block/loop4
lrwxrwxrwx 1 root root 0 7:5 -> ../../devices/virtual/block/loop5
lrwxrwxrwx 1 root root 0 7:6 -> ../../devices/virtual/block/loop6
lrwxrwxrwx 1 root root 0 7:7 -> ../../devices/virtual/block/loop7
lrwxrwxrwx 1 root root 0 8:0 ->
../../devices/pci0000:00/0000:00:0d.0/ata1/host0/target0:0:0/0:0:0:0/block/sda
```

```
lrwxrwxrwx 1 root root 0 8:1 ->
../../devices/pci0000:00/0000:00:0d.0/ata1/host0/target0:0:0/0:0:0:0/block/sda/sda1
```

The directory contains symbolic links to block devices. The sda device is connected to the PCI bus via the ATA interface.

4) See what I/O scheduler is used with the sda device :

```
root@tantor:~# cat /sys/block/sda/queue/scheduler
[none] mq-deadline
root@tantor:~# cat /sys/dev/block/8:0/queue/scheduler
[none] mq-deadline
```

sda device uses a scheduler named `none` .

5) Do it next commands :

```
root@tantor:~# echo kyber > /sys/block/sda/queue/scheduler
root@tantor:~# cat /sys/dev/block/8:0/queue/scheduler
none mq-deadline [kyber]
root@tantor:~# echo bfq > /sys/block/sda/queue/scheduler
root@tantor:~# cat /sys/block/sda/queue/scheduler
none mq-deadline kyber [bfq]
root@tantor:~# echo abc > /sys/block/sda/queue/scheduler
-bash: echo: write error: Invalid argument
root@tantor:~# echo none > /sys/block/sda/queue/scheduler
root@tantor:~# cat /sys/block/sda/queue/scheduler
[none] mq-deadline kyber bfq
```

Teams change scheduler . Non-existent scheduler abc cannot be specified. Initially, not all available schedulers were shown, now they are.

6) Look at the contents of the file `/etc/udev/rules.d/70-schedulerset.rules` , which sets the parameters for selecting the I/O scheduler:

```
 root@tantor:~# cat /etc/udev/rules.d/70-schedulerset.rules
ACTION=="add|change", KERNEL=="sd[az]", ATTR{queue/rotational}=="0",
ATTR{queue/scheduler}=" none "
```

For all devices named `sda .. sdz` the scheduler is set **to none** .

7) View the physical sector size of the device:

```
root@tantor:~# lsblk -td
NAME ALIGNMENT MIN-IO OPT-IO PHY-SEC LOG-SEC ROTA SCHED RQ-SIZE RA WSAME
sda 0 512 0 512 512 0 none 32 128 0B
sr0 0 512 0 512 512 1 mq-deadline 2 128 0B
```

Linux thinks that the sector size of device `sda` is 512 bytes.

8) Look at the parameters with which the file system was mounted on the `partition / dev / sda1` :

```
root@tantor:~# tune2fs -l /dev/sda1 | grep opt
Default mount options: user_xattr acl
root@tantor:~# mount | grep ext4
/dev/sda1 on / type ext4 (rw, relatime )
root@tantor:~# cat /etc/fstab | grep ext4
UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 / ext4 defaults 1 1
```

The file system is mounted with default parameters . Parameters that determine the functionality of the file system can be stored in the partition itself (in the "superblock").
9) See the full list of parameters:

```
root @ tantor :~# cat / proc / fs / ext 4/ sda 1/ options
rw
bsddf
nogrpid
block_validity
dioread_nolock
nodiscard
delalloc
nowarn_on_error
journal_checksum
barrier
auto_da_alloc
user_xattr
acl
noquota
resuid=0
resgid=0
errors=continue
commit=5
min_batch_time=0
max_batch_time=15000
stripe=0
data=ordered
inode_readahead_blks=32
init_itable=10
max_dir_size_kb=0
```

10) Set the discard property in the file system superblock and verify that it is set:

```
root@tantor:~# tune2fs -o +discard /dev/sda1
tune2fs 1.47.0 (5-Feb-2023)
root@tantor:~# tune2fs -l /dev/sda1 | grep opt
Default mount options: user_xattr acl discard
```

discard property has been added to the options that were added .
discard option can be set:
in the filesystem superblock as the default mount option
in the file system mount options file - / etc / fstab
cryptsetup configuration - / etc / crypttab
in LVM configuration - /etc/lvm/lvm.conf
in bootloader configuration - / boot / grub / grub . cfg

11) Check if the discard commands were sent by the operating system to the sda controller :

```
root@tantor:~# fstrim -v /
fstrim: /: the discard operation is not supported
root@tantor:~# lsblk --discard
NAME DISC-ALN DISC-GRAN DISC-MAX DISC-ZERO
sda 0 0B 0B 0
└─sda1 0 0B 0B 0
sr 0 0 0 B 0 B 0
```

Zeros in DISC-GRAN and DISC-MAX mean that discard was not used.
discard commands were sent.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

In the virtual machine where the practices are run, partitions are files. Files take up space in the host file system, so discard is not used.

12) Look at the name of the package in which the fstrim utility was installed:

```
root@tantor:~# dpkg -S fstrim
util-linux:/sbin/fstrim
util-linux: /usr/share/man/man8/fstrim.8.gz
util-linux: /lib/systemd/system/fstrim.service
util-linux: /lib/systemd/system/fstrim.timer
util-linux: /usr/share/bash-completion/completions/fstrim
```

The package is called `util - linux`

Part 2. Installing packages in Astralinux

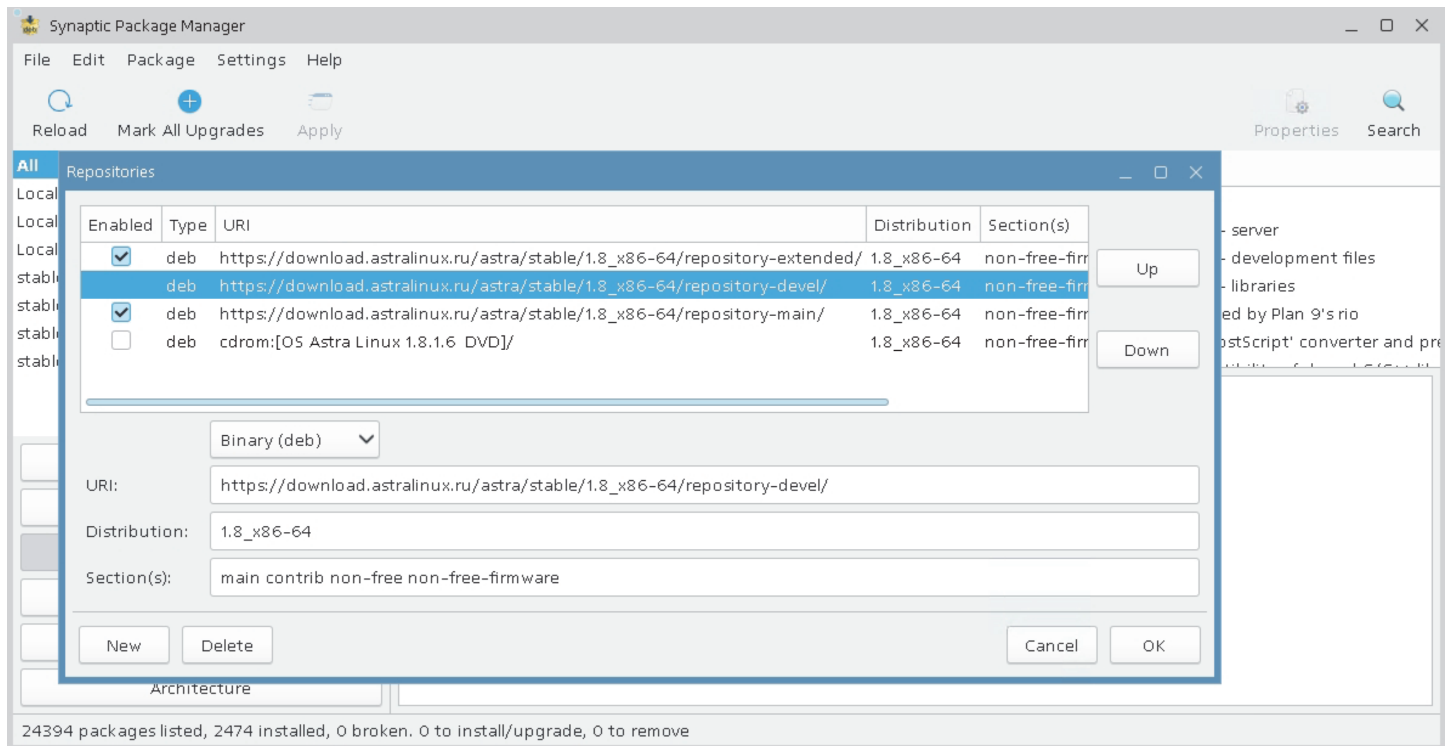
1) Launch a new terminal. In the astra user terminal, run the synaptic-pkexec utility:

```
astra @ tantor :~$ synaptic - pkexec
```

In the password entry window, enter **astra** and click OK

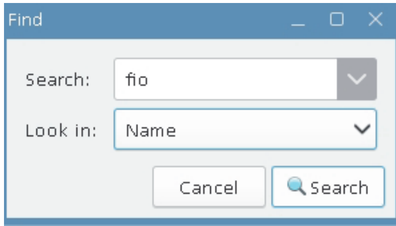
Synaptic is a graphical package manager for Astralinux .

2) In the Settings -> Repositories menu, make sure that the two checkboxes for the main and extended repositories are checked. Click OK.



3) In the menu Edit -> Search type fio . In the Look field in select Name . Click the Search button :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru



4) In the list of packages that appears, select the fio package . If the utility is installed (there is a green circle before the utility name), then skip this step. You can also install the uuid- runtime package if the uuid- runtime package is not installed.
If the fio utility is not installed, then select " Mark" from the drop-down menu for installation ". If the window does not close, click Mark. On the toolbar (three buttons under the menu), click Apply. In the window that appears, click Apply. If a window appears asking about Kernel update, click Next.

5) Close the Synaptic window by clicking the cross at the top right of the Synaptic application window.

6) In the astra user terminal window , type the command:

```
astra@tantor:~$ sudo apt install fio -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
fio is already the newest version (3.33-3+b1).
0 upgraded, 0 newly installed, 0 to remove and 636 not upgraded.
```

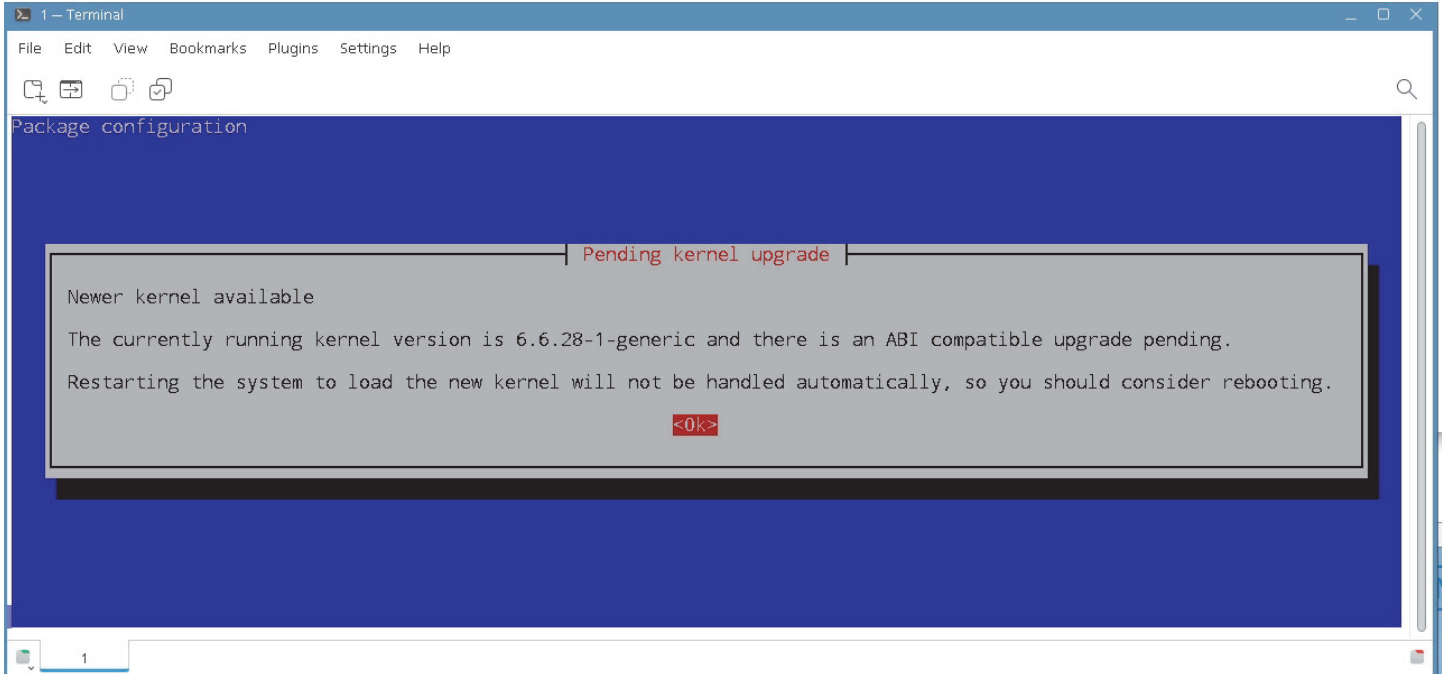
In the example, the command shows that the fio package is installed.
Close the astra user terminal by typing the key combination < ctrl + d >

7) Install nvme-cli and uuid-runtime packages:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
root@tantor:~# apt install nvme-cli uuid-runtime -y
```

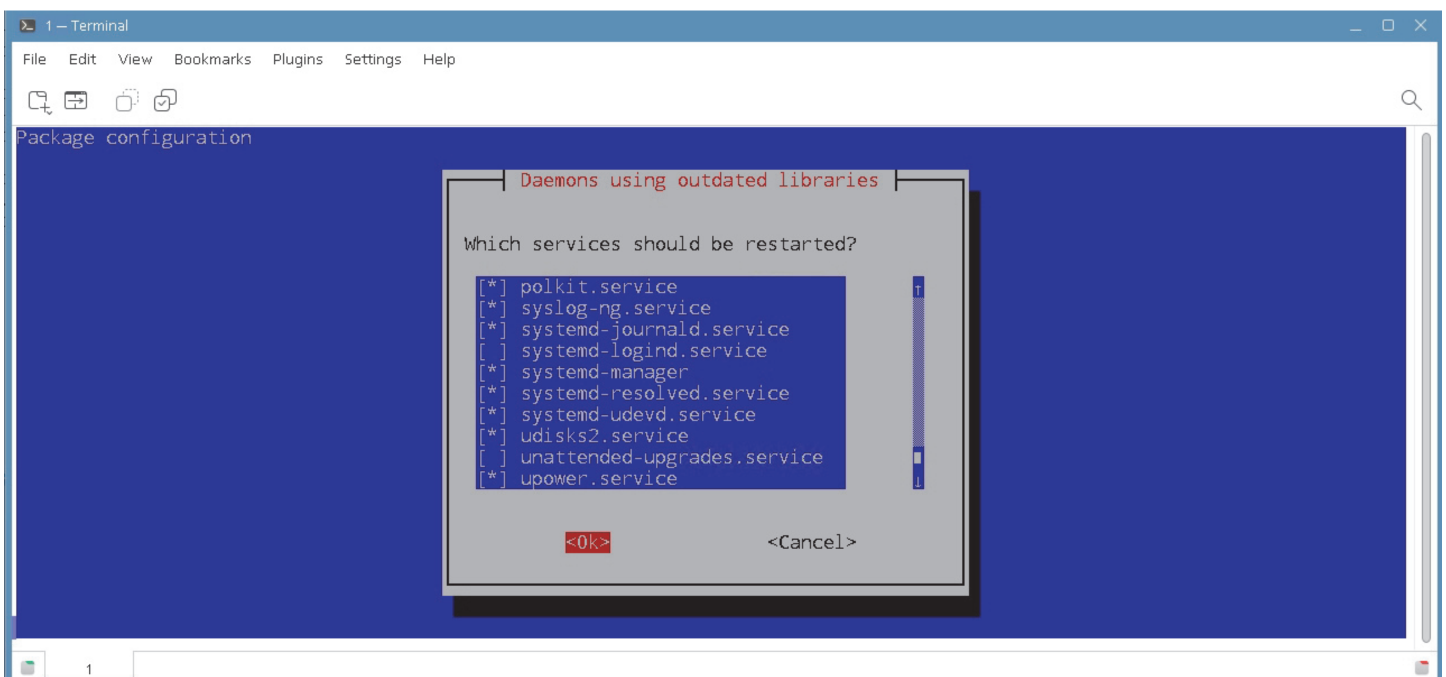
If a window with blue pseudo-graphics opens, click OK in the first window:



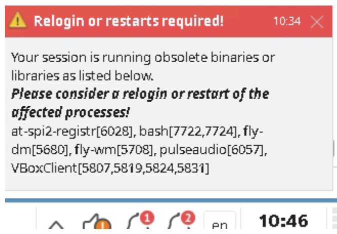
The window notifies you that you need to update all packages for which updates have been released. You can remove notifications by uninstalling the `apt - y package remove needrestart`

The active element of the pseudographics windows is highlighted in red. To switch between the visual elements of the window, use the <TAB> key, the arrow keys on the keyboard, or the mouse.

The following window will open. Press the <TAB> key, the < OK > visual element will be highlighted in the window.
The uuidd service will be started without rebooting Linux and will prompt you to reboot the services that depend on what was done during package installation. You do not need to change the set of services. If the < OK > window element is highlighted in red, press the < ENTER > key on your keyboard or click the < OK > window element with your mouse .



The message about the need to re-login can be ignored and closed:



The nvme command line utility and the uuidd service were installed.
The nvme utility, installed in the nvme - cli package , is used to read the characteristics and metrics of NVMe devices. The utility is used to determine the state of memory chips, track service life, update firmware, erase contents, reinitialize, read error logs.
The uuidd package ensures that generated uuids are unique on multi-CPU hosts. It is installed because it is required to install nvme-cli.
nvme utility is used to obtain information about NVMe devices .

You have learned how to install packages in Astralinux.

Part 3. Working with SSD and testing disk performance with the fio utility

1) Run a few commands using the `nvme utility` :

```
root@tantor:~# nvme list
Node Generic SN           Model Namespace Usage Format FW Rev
------------ ---------- ------------ -------------------- --------- ----- ----------------------- --------
/dev/nvme0n1 /dev/ng0n1 VB1234-56789 ORCL-VBOX-NVME-VER12 1 8.59GB/8.59GB 512B + 0B 1.0
root@tantor:~# nvme list-subsys
nvme-subsys0 - NQN=nqn.2014.08.org.nvmexpress:80ee80eeVB1234-56789 ORCL-VBOX-NVME-VER12
\
+- nvme0 pcie 0000:00:0e.0 live
root@tantor:~# nvme id-ctrl /dev/nvme0n1 | head -6
NVME Identify Controller:
vid: 0x80ee
ssvid: 0x80ee
sn : VB1234-56789
mn : ORCL-VBOX-NVME-VER12
fr : 1.0
```

2) Look at the list of nvme devices :

```
root@tantor:~# ls /dev/nvm*
/dev/nvme0 /dev/nvme0n1 /dev/nvme-fabrics
```

There is one NVMe device.

3) Create a file system on the NVMe device:

```
root@tantor:~# mkfs -E discard /dev/nvme0n1
mke2fs 1.47.0 (5-Feb-2023)
Creating filesystem with 2097152 4k blocks and 524288 inodes
Filesystem UUID: 0c0f68b7-d1d9-47ff-908c-9831503837f3
Superblock backups stored on blocks: 32768, 98304, 163840, 229376, 294912,
819200, 884736, 1605632
Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

The option is used by default. It is given to emphasize the importance of cleaning the contents of the SSD. In the description of the command (man mkfs . ext 4) specifies that the discard option sends a command to the device controller to clear the contents:

Attempt to discard blocks at mkfs time (discarding blocks initially is useful on solid state devices and sparse / thin-provisioned storage). When the device advertises that discard also zeroes data (any subsequent read after the discard and before write returns zero), then mark all not-yet-zeroed inode tables as zeroed. This significantly speeds up file system initialization. This is set as default.

Please note that the log was not created, the line is missing:
`Creating journal (16384 blocks): done`

4) Change the file system superblock to include the discard option:

```
root@tantor:~# tune2fs -o +discard /dev/nvme0n1
tune 2 fs 1.47.0 (5- Feb -2023)
```

5) Create a directory `/u01` , mount the created file system in it, see what size the mounted partition is:

```
root@tantor:~# mkdir /u01
root@tantor:~# mount /dev/nvme0n1 /u01
root@tantor:/# df -Th | grep "^/dev"
/dev/sda1 ext4 47G 16G 29G 36% /
/dev/nvme0n1 ext2 7.9G 24K 7.5G 1% /u01
```

The partition is mounted in `/u01` , its size is 8GB, file system **is ext2** .

6) In the root user terminal , run the command:

```
root@tantor:~# fio --ioengine=psync --filename=/tmp/test --size=1G --time_based --name=fio --group_reporting --runtime=10 --direct=1 --sync=1 --iodepth=1 --rw=read --bs=8k --numjobs=1
fio: (g=0): rw=read, bs=(R) 8192B-8192B, (W) 8192B-8192B, (T) 8192B-8192B, ioengine=psync, iodepth=1
fio-3.33
Starting 1 process
fio: Laying out IO file (1 file / 1024MiB)
Jobs: 1 (f=1): [R(1)][100.0%][r=36.4MiB/s][r=4664 IOPS][eta 00m:00s]
fio: (groupid=0, jobs=1): err= 0: pid=7432: Mon Dec 9 00:07:59 2024
read: IOPS= 4682 , BW=36.6MiB/s ( 38.4MB/s )(366MiB/10001msec)
clat (usec): min=92, max=8607, avg=207.12, stdev=306.94
lat (usec): min=93, max=8608, avg=208.56, stdev=306.95
clat percentiles (usec):
| 1.00th=[ 143], 5.00th=[ 149], 10.00th=[ 151], 20.00th=[ 155],
| 30.00th=[ 157], 40.00th=[ 159], 50.00th=[ 159], 60.00th=[ 161],
| 70.00th=[ 165], 80.00th=[ 172], 90.00th=[ 237], 95.00th=[ 253],
| 99.00th=[ 2245], 99.50th=[ 2900], 99.90th=[ 3490], 99.95th=[ 3654],
| 99.99th=[ 3982]
bw ( KiB/s): min=33213, max=40688, per=100.00%, avg=37505.37, stdev=1932.35, samples=19
iops : min= 4151, max= 5086, avg=4687.95, stdev=241.58, samples=19
lat (usec) : 100=0.01%, 250=94.32%, 500=3.66%, 750=0.61%, 1000=0.03%
lat (msec) : 2=0.15%, 4=1.22%, 10=0.01%
cpu : usr=12.33%, sys=11.35% , ctx=46830, majf=0, minf=14
IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=46827,0,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
READ: bw=36.6MiB/s ( 38.4MB/s ), 36.6MiB/s-36.6MiB/s (38.4MB/s-38.4MB/s), io=366MiB (384MB), run=10001-10001msec

Disk stats (read/write):
sda: ios=45651/4, merge=0/1, ticks=8268/7, in_queue=8282, util=97.89%
```

The test results are highlighted in red. The rest of the verbose output does not require attention and is used in rare cases.

Test parameters:
PostgreSQL's work with the file system corresponds to `-- ioengine = psync`
PostgreSQL block size `-- bs =8 k`
The queue size (`iodepth`) is set to 1 because direct=1 is selected.

tantor

The test performed only reading from a file. Test result: maximum number of input/output operations IOPS= 4682 , average speed 38.4MB/s .

7) Test it NVMe partition :

```
root@tantor:~# fio --ioengine=psync --filename=/u01/test --size=1G --time_based --name=fio --group_reporting --runtime=10 --direct=1 --sync=1 --iodepth=1 --rw=read --bs=8k --numjobs=1
```

```
fio: (g=0): rw=read, bs=(R) 8192B-8192B, (W) 8192B-8192B, (T) 8192B-8192B, ioengine=psync, iodepth=1
fio-3.33
Starting 1 process
fio: Laying out IO file (1 file / 1024MiB)
Jobs: 1 (f=1): [R(1)][100.0%][r=43.1MiB/s][r=5520 IOPS][eta 00m:00s]
fio: (groupid=0, jobs=1): err= 0: pid=7441: Mon Dec 9 00:08:59 2024
read: IOPS= 5312 , BW=41.5MiB/s ( 43.5MB/s )(415MiB/10001msec)
clat (usec): min=77, max=11930, avg=181.67, stdev=300.00
lat (usec): min=78, max=11932, avg=183.12, stdev=300.04
clat percentiles (usec):
| 1.00th=[ 128], 5.00th=[ 133], 10.00th=[ 137], 20.00th=[ 139],
| 30.00th=[ 141], 40.00th=[ 143], 50.00th=[ 145], 60.00th=[ 147],
| 70.00th=[ 149], 80.00th=[ 151], 90.00th=[ 167], 95.00th=[ 229],
| 99.00th=[ 2147], 99.50th=[ 2835], 99.90th=[ 3523], 99.95th=[ 3752],
| 99.99th=[ 5932]
bw ( KiB/s): min=38848, max=43936, per=99.80%, avg=42413.47, stdev=1514.01, samples=19
iops : min= 4856, max= 5492, avg=5301.68, stdev=189.25, samples=19
lat (usec) : 100=0.11%, 250=97.55%, 500=0.99%, 750=0.12%, 1000=0.08%
lat (msec) : 2=0.08%, 4=1.04%, 10=0.02%, 20=0.01%
cpu : usr=12.13%, sys=12.54% , ctx=53130, majf=0, minf=15
IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
issued rwts: total=53128,0,0,0 short=0,0,0,0 dropped=0,0,0,0
latency : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
READ: bw=41.5MiB/s ( 43.5MB/s ), 41.5MiB/s-41.5MiB/s (43.5MB/s-43.5MB/s), io=415MiB (435MB), run=10001-10001msec

Disk stats (read/write):
nvme0n1: ios=52790/2, merge=0/1, ticks=8283/5, in_queue=8289, util=98.93%
```

Physically, the virtual machine stores / dev / sda and / dev / nvme0 in adjacent VMDK files on the host file system. Different types of I/O buses are used: SATA and NVMe. The noticeable difference in performance is ~10% . The practical result is that using NVMe in VirtualBox has an advantage over SATA AHCI.

8) Temporary files /tmp/test and / u01 / test of 1GB each were created. Delete these files:

```
root @ tantor :~# rm - f / tmp / test
root @ tantor :~# rm - f / u 01/ test
```

9) Unmount / u01 :

```
root @ tantor :~# umount / u 01
```

10) This point is optional.
Example of testing a write within a partition. The partition must be unmounted and recreated after the test, since the contents of the partition will be erased:

```
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=4k --rw=randwrite --iodepth=32 --runtime=10 --fsync=1 | grep BW=
write: IOPS=583, BW=2335KiB/s ( 2391kB/s )(22.8MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize= 8k --rw=randwrite --iodepth=32 --runtime=10 --fsync=1 | grep BW=
write: IOPS=561, BW=4491KiB/s ( 4599kB/s )(43.9MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=16k --rw=randwrite --iodepth=32 --runtime=10 --fsync=1 | grep BW=
write: IOPS=504, BW=8066KiB/s ( 8260kB/s )(78.8MiB/10001msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=32k --rw=randwrite --iodepth=32 --runtime=10 --fsync=1 | grep BW=
```

```
write: IOPS=411, BW=12.9MiB/s ( 13.5MB/s )(129MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=64k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
write: IOPS=321, BW=20.1MiB/s ( 21.1MB/s )(201MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=128k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=1 | grep BW=
write: IOPS=276, BW=34.5MiB/s ( 36.2MB/s )(346MiB/10004msec); 0 zone resets

root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=4k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
write: IOPS=26.6k, BW=104MiB/s ( 109MB/s )(1038MiB/10002msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=8k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
write: IOPS=16.6k, BW=130MiB/s ( 136MB/s )(1296MiB/10007msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=16k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
write: IOPS=9845, BW=154MiB/s ( 161MB/s )(1539MiB/10006msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=32k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
write: IOPS=8015, BW=250MiB/s ( 263MB/s )(2508MiB/10015msec); 0 zone resets
root@tantor:/# fio --filename=/dev/nvme0n1 --name=a --blocksize=64k --rw=randwrite --iodepth=32 --runtime=10 --
fsync=0 | grep BW=
write: IOPS=4582, BW=286MiB/s ( 300MB/s )(2864MiB/10001msec); 0 zone resets
```

The examples tested parallel random writes of blocks of different sizes with and without fsync. The f sync option sends a request to the device controller to write, clearing the internal caches of the storage device. The write guarantee is determined by the device controller. The fio utility can read and write both to files and directly work with disk partitions and devices. This allows you to measure the work of physical interfaces and devices without using the Linux page cache.

Part 4. Testing the ext4 fastcommit journal

1) Create a journaled **ext 4** filesystem on **the / dev / nvme0n1 partition :**

```
root@tantor:~# mkfs.ext4 /dev/nvme0n1
mke2fs 1.47.0 (5-Feb-2023)
/dev/nvme0n1 contains a ext2 file system
Proceed anyway? (y,N) y
Creating filesystem with 2097152 4k blocks and 524288 inodes
Filesystem UUID: 9c8c5140-2744-4f2f-acc6-baeab282efd6
Superblock backups stored on blocks:
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

Note: adding a log is possible without formatting using the command:
tune2fs -O + has_journal /dev/nvme0n1
Data is saved in the file system.

2) Look at the characteristics of the created file system:

```
root@tantor:/# dumpe2fs /dev/nvme0n1 | head -55
dumpe2fs 1.47.0 (5-Feb-2023)
Filesystem volume name: <none>
Last mounted on: <not available>
Filesystem UUID: d38ab2fb-3ba2-4593-9ce4-e5ce9ed6c06c
Filesystem magic number: 0xEF53
Filesystem revision #: 1 (dynamic)
Filesystem features: has_journal ext_attr resize_inode dir_index filetype extent 64bit flex_bg sparse_super
large_file huge_file dir_nlink extra_isize metadata_csum
Filesystem flags: signed_directory_hash
Default mount options: user_xattr acl
Filesystem state: clean
Errors behavior: Continue
Filesystem OS type: Linux
Inode count: 524288
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
Block count: 2097152
Reserved block count: 104857
Overhead clusters: 58505
Free blocks: 2038641
Free inodes: 524277
First block: 0
Block size: 4096
Fragment size: 4096
Group descriptor size: 64
Reserved GDT blocks: 1023
Blocks per group: 32768
Fragments per group: 32768
Inodes per group: 8192
Inode blocks per group: 512
Flex block group size: 16
Filesystem created: Sun Dec 8 01:14:49 2035
Last mount time: n/a
Last write time: Sun Dec 8 01:14:49 2035
Mount count: 0
Maximum mount count: -1
Last checked: Sun Dec 8 01:14:49 2035
Check interval: 0 (<none>)
Lifetime writes: 4129 kB
Reserved blocks uid: 0 (user root)
Reserved blocks gid: 0 (group root)
First inode: 11
Inode size: 256
Required extra size: 32
Desired extra size: 32
Journal inode: 8
Default directory hash: half_md4
Directory Hash Seed: 3e913b49-f24d-4c17-a94d-b66abf6f0e9e
Journal backup: inode blocks
Checksum type: crc32c
Checksum: 0x37fd9c6b
Journal features: (none)
Total journal size: 64M
Total journal blocks: 16384
Max transaction length: 16384
Fast commit length: 0
Journal sequence: 0x00000001
Journal start: 0
```

3) Mount the created file system to /u01:

```
root@tantor:/# mount /dev/nvme0n1 /u01
```

4) Go to the root of the mounted file system, create a directory, go to it and run a test of the speed of writing to WAL files:

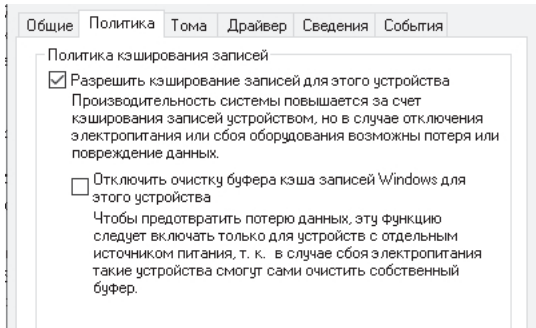
```
root@tantor:/# cd /u01
root@tantor:/u01# mkdir a1
root@tantor:/u01# cd a1
root@tantor:/u01/a1# pg_test_fsync
```

```
5 seconds per test
O_DIRECT supported on this platform for open_datasync and open_sync.

Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
open_datasync 566.595 ops/sec 1765 usecs/op
fdatasync 567 .752 ops/sec 1761 usecs/op
fsync 194.505 ops/sec 5141 usecs/op
fsync_writethrough n/a
open_sync 202.077 ops/sec 4949 usecs/op

Compare file sync methods using two 8kB writes:
(in wal_sync_method preference order, except fdatasync is Linux's default)
open_datasync 287.440 ops/sec 3479 usecs/op
fdatasync 506 .992 ops/sec 1972 usecs/op
fsync 189.935 ops/sec 5265 usecs/op
fsync_writethrough n/a
open_sync 97.948 ops/sec 10209 usecs/op

Compare open_sync with different write sizes:
(This is designed to compare the cost of writing 16kB in different write
open_sync sizes.)
1 * 16kB open_sync write 185.468 ops/sec 5392 usecs/op
2 * 8kB open_sync writes 100.674 ops/sec 9933 usecs/op
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
4 * 4kB open_sync writes 48.484 ops/sec 20625 usecs/op
8 * 2kB open_sync writes 25.928 ops/sec 38569 usecs/op
16 * 1kB open_sync writes 12.982 ops/sec 77033 usecs/op

Test if fsync on non-write file descriptor is honored:
(If the times are similar, fsync() can sync data written on a different
descriptor.)
write, fsync, close 192.591 ops/sec 5192 usecs/op
write, close, fsync 195.865 ops/sec 5106 usecs/op

Non-synced 8kB writes:
write 450551.547 ops/sec 2 usecs/op
```

`pg_test_fsync` utility tests the write speed in various ways to files in the directory from which it was launched. It was launched in the **`/u01/a1 directory`**
Creating a subdirectory does not affect the test result.
If you get values that are an order of magnitude higher: instead of 567 you get ~8000, then you have write caching enabled in the host operating system where the virtual machine is running. Example, if the host operating system (where the virtual machine is running) is Windows:



If you repeat the test several times, the repeatability of the results will most likely be low: the spread of values will be ~10%. This is normal for performing practices, but you cannot compare the values obtained by the tests. The nvme0 section is located in the host operating system file.

5) Look at the statistics of the regular (jbd2) log:

```
root@tantor:/u01/a1# cat /proc/fs/jbd2/nvme0n1-8/info
10438 transactions (10436 requested), each up to 4096 blocks
average:
0ms waiting for transaction
0ms request delay
  4 ms running transaction
0ms transaction was being locked
0ms flushing data (in ordered mode)
0ms logging transaction
  5040us average transaction commit time
  1 handle per transaction
  1 block per transaction
  3 logged blocks per transaction
```

3 block on transaction , `10438` transactions . The number of transactions depends on how many operations the **`pg_test_fsync utility managed to perform`** in the time allotted to her.

6) Look at the fast magazine statistics commit (" fc "):

```
root@tantor:/u01/a1# cat /proc/fs/ext4/nvme0n1/fc_info
fc stats:
0 commits
0 ineligible
0 numblks
0us avg_commit_time
Ineligible reasons:
"Extended attributes changed": 0
"Cross rename": 0
```

```
"Journal flag changed": 0
"Insufficient memory": 0
"Swap boot": 0
"Resize": 0
"Dir renamed": 0
"Falloc range op": 0
" Data journalling ": 0
" Encrypted filename ": 0
```

There is no quick commit log, so the statistics are empty.

7) Recreate the filesystem on / dev / nvme0n1 :

```
root@tantor:/u01# cd /
root@tantor:/# umount /u01
root@tantor:/# mkfs.ext4 /dev/nvme0n1
mke2fs 1.47.0 (5-Feb-2023)
/dev/nvme0n1 contains a ext4 file system
Proceed anyway? (y,N) y
Creating filesystem with 2097152 4k blocks and 524288 inodes
Filesystem UUID: 102365ba-9b22-4ac6-95f9-9f35c146e2ae
Superblock backups stored on blocks:
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information: done
```

8) Enable the fast commit journal and mount the file system:

```
root@tantor:/# tune2fs -O + fast_commit /dev/nvme0n1
tune2fs 1.47.0 (5-Feb-2023)
root@tantor:/# dumpe2fs /dev/nvme0n1 | grep commit
dumpe2fs 1.47.0 (5-Feb-2023)
Filesystem features: has_journal ext_attr resize_inode dir_index fast_commit
filetype extent 64bit flex_bg sparse_super large_file huge_file dir_nlink
extra_isize metadata_csum
Fast commit length:      256
```

9) C mount file system :

```
root@tantor:/# mount /dev/nvme0n1 /u01
```

10) Repeat the test:

```
root @ tantor :/# cd / u 01
root@tantor:/u01# mkdir a1
root@tantor:/u01# cd a1
root@tantor:/u01/a1# pg_test_fsync
5 seconds per test
O_DIRECT supported on this platform for open_datasync and open_sync.

Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
open_datasync 509.642 ops/sec 1962 usecs/op
fdatasync 554 .320 ops/sec 1804 usecs/op
fsync 290.867 ops/sec 3438 usecs/op
fsync_writethrough n/a
open_sync 294.644 ops/sec 3394 usecs/op

Compare file sync methods using two 8kB writes:
(in wal_sync_method preference order, except fdatasync is Linux's default)
open_datasync 285.667 ops/sec 3501 usecs/op
fdatasync 505 .847 ops/sec 1977 usecs/op
fsync 273.459 ops/sec 3657 usecs/op
fsync_writethrough n/a
```

```
open_sync 143.576 ops/sec 6965 usecs/op

Compare open_sync with different write sizes:
(This is designed to compare the cost of writing 16kB in different write
open_sync sizes.)
1 * 16kB open_sync write 273.442 ops/sec 3657 usecs/op
2 * 8kB open_sync writes 144.108 ops/sec 6939 usecs/op
4 * 4kB open_sync writes 74.927 ops/sec 13346 usecs/op
8 * 2kB open_sync writes 37.400 ops/sec 26738 usecs/op
16 * 1kB open_sync writes 19.074 ops/sec 52429 usecs/op

Test if fsync on non-write file descriptor is honored:
(If the times are similar, fsync() can sync data written on a different
descriptor.)
write, fsync, close 288.831 ops/sec 3462 usecs/op
write, close, fsync 292.142 ops/sec 3423 usecs/op

Non-synced 8kB writes:
write 440343.998 ops/sec 2 usecs/op
```

11) Look at the statistics of writing to the file system logs:

```
root@tantor:/u01# cat /proc/fs/jbd2/nvme0n1-8/info
69 transactions (67 requested), each up to 4032 blocks
average:
0ms waiting for transaction
0ms request delay
  1376ms running transaction
0ms transaction was being locked
0ms flushing data (in ordered mode)
0ms logging transaction
  5194us average transaction commit time
  278 handles per transaction
  1 block per transaction
  3 logged blocks per transaction
root@tantor:/u01# cat /proc/fs/ext4/nvme0n1/fc_info
fc stats:
16175 commits
62 ineligible
16175 numblks
1607us avg_commit_time
Ineligible reasons:
"Extended attributes changed": 0
"Cross rename": 0
"Journal flag changed": 0
"Insufficient memory": 0
"Swap boot": 0
"Resize": 0
"Dir renamed": 0
"Falloc range op": 0
"Data journalling": 0
"Encrypted filename": 0
```

3 blocks on transaction , `69*3=` **207** blocks V jbd2 journal . But **in addition** to this journal, 16175 blocks were written to the fast commit journal. The journal writes did not require separate fsync operations, the writes were performed block by block with the Force Unit Access (FUA) write-immediate flag. FUA is applied only to one block and does not cause the entire storage controller cache to be flushed.
In the previous test, `10438*3=` **31314** blocks were written to the jbd2 log in 10438 operations.

What is better {16382 and 31314 blocks} or {16244 and 10438 write operations}? S SDs write to NAND chips in 16K (or larger) blocks, which is equal to 4 blocks (4K) of journal. Therefore, the number of operations is more important for SSDs. Therefore, the results of the conducted test are approximately the same: **567** and **554** ops / sec . HDD has a block size of 512 bytes or 4K, but sequential writing of three blocks is faster than one because in the second case you have to wait for the disk to rotate. For a combination of HDD disks with non-volatile memory, it is even more difficult to predict the result. **the** fast commit log will give an advantage or not is testing. In this example, the pg_test_fsync utility was used for testing **, which is designed to test the** work with WAL files.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The fastcommit log reveals its advantages on file systems where metadata changes frequently, i.e. files are often created, deleted, and resized. This can happen, for example, in a PostgreSQL tablespace directory where temporary files are often created and deleted. In the directory where WAL files are located, files are created infrequently. For directories such as PGDATA or the tablespace directory for temporary files, other tests are needed , not pg_test_fsync .
Most likely, with such directories, the fast commit log will provide benefits with the same level of fault tolerance.
The JBD2 journal always writes at least three blocks, so FUA cannot be used. Initiating a write to the JBD2 journal causes the entire controller cache to be emptied. If the controller cache has accumulated more than 16 KB, the write may theoretically be slower. In practice, the write speed to NAND chips is comparable to the speed of memory, and the SSD controller can write asynchronously.

12) This and the following points are optional until the end of this part of the practice.
Previously we tested the ext 4 file system. You can test the speed without a journal (ext2 functionality). Disable the journal on / dev / nvme 0 n 1 :

```
root @ tantor :/ u 01# cd /
root @ tantor :/# umount / u 01
root@tantor:/# tune2fs -O ^ has_journal /dev/nvme0n1
root@tantor:/# dumpe2fs /dev/nvme0n1 | grep commit
dumpe2fs 1.47.0 (5-Feb-2023)
Filesystem features: ext_attr resize_inode dir_index filetype extent 64bit
flex_bg sparse_super large_file huge_file dir_nlink extra_isize metadata_csum
Fast commit length:      0
```

The command output does not contain the has_journal and fast_commit properties . This means that the file system is journalless .

13) Mount the file system:

```
root @ tantor :/# mount / dev / nvme 0 n 1 / u 01
```

14) Repeat the test:

```
root @ tantor :/# cd / u 01
root@tantor:/u01# mkdir a1
root@tantor:/u01# cd a1
root@tantor:/u01/a1# pg_test_fsync
5 seconds per test
O_DIRECT supported on this platform for open_datasync and open_sync.

Compare file sync methods using one 8kB write:
(in wal_sync_method preference order, except fdatasync is Linux's default)
open_datasync 561.778 ops/sec 1780 usecs/op
fdatasync 567 .933 ops/sec 1761 usecs/op
fsync 323.302 ops/sec 3093 usecs/op
fsync_writethrough n/a
open_sync 343.145 ops/sec 2914 usecs/op

Compare file sync methods using two 8kB writes:
(in wal_sync_method preference order, except fdatasync is Linux's default)
open_datasync 284.528 ops/sec 3515 usecs/op
fdatasync 505 .311 ops/sec 1979 usecs/op
fsync 311.811 ops/sec 3207 usecs/op
fsync_writethrough n/a
open_sync 163.685 ops/sec 6109 usecs/op

Compare open_sync with different write sizes:
(This is designed to compare the cost of writing 16kB in different write
open_sync sizes.)
1 * 16kB open_sync write 301.329 ops/sec 3319 usecs/op
2 * 8kB open_sync writes 166.580 ops/sec 6003 usecs/op
4 * 4kB open_sync writes 85.399 ops/sec 11710 usecs/op
8 * 2kB open_sync writes 44.381 ops/sec 22532 usecs/op
16 * 1kB open_sync writes 22.332 ops/sec 44779 usecs/op

Test if fsync on non-write file descriptor is honored:
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
(If the times are similar, fsync() can sync data written on a different
descriptor.)
write, fsync, close 321.995 ops/sec 3106 usecs/op
write, close, fsync 344.132 ops/sec 2906 usecs/op

Non-synced 8kB writes:
write 348190.711 ops/sec 3 usecs/op
```

For fdatasync the values are the same as with the journal.

Part 5. Removing the limit on the number of open files

1) By default, the maximum number of connections is 100. Set the maximum number of connections to 10000:

```
postgres@tantor:~$ psql -c "alter system set max_connections=1100;"
ALTER SYSTEM
```

2) Try running pgbench test with 1100 connections:

```
postgres@tantor:~$ pgbench -c 1100 -T 10 -P 5
pgbench: error: need at least 1103 open files, but system limit is 1024
pgbench: hint: Reduce number of clients, or use limit/ulimit to increase the
system limit.
```

An error was returned stating that the pgbench utility wants to open 1103 files, but the operating system limits the postgres user processes to 1024 files.

3) Check the real limits of already running processes named postgres :

```
postgres@tantor:~$ for PID in $(pgrep "postgres"); do cat /proc/$PID/limits |
grep files; done | uniq
Max open files 1024 524288 files
```

Soft limit is 1024 files, hard limit is 524288 files.

4) Look at the limit for new processes for users postgres , root , astra :

```
postgres@tantor:~$ sudo -u postgres bash -c 'ulimit -n'
1024
postgres@tantor:~$ sudo -u root bash -c 'ulimit -n'
1024
postgres@tantor:~$ sudo -u astra bash -c 'ulimit -n'
1024
```

All user processes have a limit of 1024. This limit is suitable for desktop use, but not for a server. Although this is a soft limit, applications and utilities may not handle warnings and refuse to work, such as the pgbench utility.

Does this limit affect instance processes? By default, the max_files_per_process configuration parameter = 1000 , so it does not affect. However, before increasing the max_files_per_process value, you must remove the operating system - level limits.

5) To change the limits for instances started manually by the pg_ctl utility, you need to add or change the following lines in the /etc/security/limits.conf file:

```
* hard nofile infinity
root hard nofile infinity
* soft nofile infinity
root soft nofile infinity
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
postgres@tantor:~$ sudo mcedit /etc/security/limits.conf
```

6) Check that the changes have taken effect:

```
postgres@tantor:~$ sudo -u postgres bash -c 'ulimit -n'
1048576
postgres@tantor:~$ sudo -u astra bash -c 'ulimit -n'
1048576
postgres@tantor:~$ sudo -u root bash -c 'ulimit -n'
1048576
postgres@tantor:~$ sudo -u postgres bash -c 'ulimit -Ht'
unlimited
postgres@tantor:~$ sudo -u astra bash -c 'ulimit -Ht'
unlimited
postgres@tantor:~$ sudo -u root bash -c 'ulimit -Ht'
unlimited
```

This will not affect instances launched via `systemd` .
Edit file `/usr/lib/systemd/system/tantor-se-server-16.service`, adding after `[Service]`

```
LimitNOFILE= infinity
LimitNOFILESoft= infinity
```

```
postgres@tantor:~$ sudo mcedit /usr/lib/systemd/system/tantor-se-server-16.service
```

8) Update the systemd configuration and restart the instance:

```
postgres@tantor:~$ sudo systemctl daemon-reload
postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
```

9) Check the real limits of already running processes named postgres :

```
postgres@tantor:~$ for PID in $(pgrep "postgres"); do cat /proc/$PID/limits |
grep files; done | uniq
Max open files 1024 524288 files
Max open files 1048576          1048576 files
```

There are processes with a limit of 1024 - this is an instance of Astralinux PostgreSQL , its service file was not edited. The tantor instance processes had their limits changed.

10) Run the pgbench test, specifying 1100 connections:

```
postgres@tantor:~$ pgbench -c 1100 -T 10 -P 5
starting vacuum...end.
progress: 7.9 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 10.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 15.0 s, 55.4 tps, lat 4859.114 ms stddev 1268.481, 0 failed
progress: 20.0 s, 67.8 tps, lat 9833.140 ms stddev 1435.181, 0 failed
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
Number of clients: 1100
Number of threads: 1
Maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 1100
number of failed transactions: 0 (0.000%)
latency average = 10740.735 ms
stddev latency = 4219.216 ms
initial connection time = 7798.795 ms
tps = 65.627380 (without initial connection time)
```

The test was completed.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Part 6. Example of creating plpgsql code for testing

1) Create an `init .sql file :`

```
postgres @ tantor :~$ mcedit init .sql

create schema test;
select format('create table test.film_summary%s (film_id int, title varchar,
release_year smallint) with (autovacuum_enabled=off);', g.id) from
generate_series(1, 10000) as g(id)
\ gexec
```

The command creates 10,000 empty tables, 10,000 TOAST tables, and 10,000 TOAST indexes. Autovacuum is disabled so that when changes are made to table rows, it does not suddenly trigger and thus does not create fluctuations in the metrics in the test results. The schema is used for convenience, so that the tables are not visible in the search path and so that they can be removed by removing the schema.

2) Run the script :

```
postgres@tantor:~$ time psql -f init.sql > /dev/null

real 2m29 .225s
user 0m1.687s
sys 0 m 0.999 s
```

created the test schema and 10,000 tables in the schema in 2.5 minutes .

3) Try deleting tables and schema:

```
postgres=# \timing
Timing is on.
postgres=# DO
$$
begin
for i in 1..10000 loop
execute concat('drop table if exists test.film_summary',i);
end loop;
execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

ERROR: out of shared memory
HINT: You might need to increase max_locks_per_transaction.
CONTEXT: SQL statement "drop table if exists test.film_summary 2571 "
PL/pgSQL function inline_code_block line 4 at EXECUTE
Time : 936.719 ms
```

Using plpgsql resulted in a lock count issue: **commands were executed by a single transaction** . That's why tables were created by a script, not a plpgsql block.

After 0.9 seconds, the lock limit was exceeded when deleting the 2571 table. There were 2571 locks set. The number of locks depends on the parameters `max_locks_per_transaction*max_connextion s =64*100=6400` . How do the numbers 2571 and 6400 relate? For each table, 3 locks are set: the table itself, its TOAST table and TOAST index. 2571*3=7713 which is greater than 6400. It is greater because the memory for the lock structure and any memory structures is allocated with rounding.

4) Add the commit command and execute the block:

```
postgres=# DO
$$
begin
for i in 1..10000 loop
execute concat('drop table if exists test.film_summary',i);
commit;
end loop;
execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 46568.714 ms (00:46.569)
```

The block will execute successfully and delete 30,000 objects in 46 seconds . That's a long time.

5) Recreate the tables:

```
postgres@tantor:~$ time psql -f init.sql > /dev/null

real 3m12 .168s
user 0m0.404s
sys 0m0.223s
```

6) Create more efficient plpgsql code with intermediate transaction commits:

```
postgres=# \timing on \\
DO
$$
begin
for i in 1..10000 by 100 loop
  for j in 0..99 loop
execute concat('drop table if exists test.film_summary',i+j);
  end loop;
commit;
end loop;
execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 6597.938 ms (00:06.598)
```

then the execution time will be 6.5 seconds . Intermediate transaction commits hold the horizon a little longer, but puts less stress on lightweight WAL structure locks and executes faster.

This was an example of how, when writing test scripts, it is important to pay attention to what transactions are explicitly or implicitly formed to execute commands.

If you increase the `max_connections number` to 1000, restart the instance, create the tables, and execute the block without `commit` , the tables will be deleted in 6 seconds :

```
Time : 6030.153 ms (00:06.030)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

There is no need to check this, so as not to waste time (3 minutes to create tables).

If you increase `the max_connections number` to 1000, restart the instance, create the tables and delete them with the command:

```
postgres=# \timing
Timing is on.
postgres=# drop schema test cascade;
NOTICE: drop cascades to 10000 other objects
DETAIL: drop cascades to table test.film_summary1
drop cascades to table test.film_summary2
drop cascades to table test.film_summary3
...
drop cascades to table test.film_summary99
drop cascades to table test.film_summary100
and 9900 other objects (see server log for list)
DROP SCHEMA
Time : 7526.961 ms (00:07.527)
```

then the removal time will be 7.5 seconds .
7) The `init.sql file` of this part of the practice is useful and can be used to create a large number of objects for testing purposes.
If you need to fill tables with rows, you can use the command:

```
select format(' insert into test.film_summary%1$s select i, %2$s || i, i from
generate_series(0, 153) as i;', g.id, E'\'text number \'') from
generate_series(1, 10000) as g(id)
\ gexec
```

The command creates 154 rows in each table in one block, almost completely filling the block. Most blocks in real tables are almost completely filled, and the tables will match this. The maximum number of rows in a block is 156, space for a couple of rows is needed so that when a row in a block is updated, HOT cleanup will work.
The script with this command will be executed in 46 seconds .

If you need to read blocks to update the insert commit flag, you can run the command:

```
select format('select * from test.film_summary%s where film_id = 0;', g.id) from
generate_series(1, 10000) as g(id)
\ gexec
```

The script with this command will be executed in 3.5 seconds .

You can also use a command that collects statistics and freezes the rows so that they don't suddenly get dirty in the future:

```
select format('vacuum (freeze, analyze) test.film_summary%s;', g.id) from
generate_series(1, 10000) as g(id)
\ gexec
```

However, the script with this command will run for 1 minute 45 seconds , which is long.

8) Tables can be used for load testing.

For this create a function:

```
create or replace function inittest1(clientid anyelement) returns void
as $$
begin
for i in 1..15 loop
execute concat('update test.film_summary', i ,' set title=title ', 'where film_id
= ', clientid);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
end loop;
end;
$$
LANGUAGE plpgsql;
```

The function updates a row in 15 tables (16 is the maximum number of fastpath locks) with the value passed in the function parameter.

Test script for pgbench:
```
echo "select inittest1( :client_id);" > inittest1.sql
```

`client_id` - pgbench session number, set in a parameter that is available in test scripts.

Launch test :
```
pgbench -n -c 36 -T 10 -P 3 -f inittest1.sql
progress: 3.0 s, 1358.0 tps, lat 25.905 ms stddev 10.597, 0 failed
progress: 6.0 s, 1203.0 tps, lat 29.948 ms stddev 10.698, 0 failed
progress: 9.0 s, 1141.7 tps, lat 31.510 ms stddev 11.303, 0 failed
number of transactions actually processed: 12406
latency average = 28.895 ms
stddev latency = 11.061 ms
tps = 1242.842959 (without initial connection time)
```

The optimal number of sessions on 4 cores with default settings starts with ~36 sessions. With more than 1 session, HOT cleanup will not be able to run often and the number of blocks in the tables will increase. As the number of blocks increases , tps decreases.

Part 7. Example of creating programs for testing

1) The utilities pgbench, pg_test_fsync, fio discussed above are great for testing.
Simple tests written in the programming languages c, java , pyt h on can be used for testing . An example of a self-written test and the task it solves
https :// www . percona . com / blog / fsync - performance - storage - devices /
Create a file `fsync.py` :

```
postgres @ tantor :~$ mcedit fsync .py

#!/usr/bin/python
import os, sys, mmap
# Open a file
#fd = os.open( "testfile", os.O_RDWR|os.O_CREAT|os.O_DSYNC )
fd = os.open( "testfile", os.O_RDWR|os.O_CREAT )
fd2 = os.open( "testfile2", os.O_RDWR|os.O_CREAT )
m = mmap.mmap(-1, 8192)
for i in range(1,5000):
os.lseek(fd,os.SEEK_SET,0)
m[1] = 1
os.write(fd, m)
os.lseek(fd2,os.SEEK_SET,0)
os.write(fd2, m)
os.fsync(fd)
os.fsync(fd2)
# Close opened file
os.close( fd )
os.close( fd2 )
```

2) Run the test:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres@tantor:~$ time python3 ./fsync.py

real 0m5.379s
user 0m0.276s
sys 0 m 0.727 s
```

You can change the file opening parameters in the `os.open function` and check the program execution time.

3) Create a `gen .py file :`

```
postgres @ tantor :~$ mcedit gen.py

def main():
for count in [5000, 10000, 25000]:
with open(f'workload1c_{count}.sql', 'w') as f:
f.write('BEGIN;\n')
for i in range(1, count):
f.write(f"""create temporary table ttt{i} (
_C_1RRef byte,
_C_2RRef byte,
_C_3RRef byte,
_C_4 varchar(150),
_C_5 numeric(9,0),
_C_6RRef byte,
_C_7RRef byte,
_C_8RRef byte,
_C_9RRef byte,
_C_10 boolean,
_C_11 numeric(5,0),
_C_12 numeric(10,0),
_C_13 varchar(430),
_C_14 numeric(5,0),
_C_15RRef byte,
_C_16RRef byte,
_C_17RRef byte,
_C_18RRef byte,
_C_19_TYPE byte,
_C_19_S varchar(150),
_C_19_RTRef byte,
_C_19_RRRef byte,
_C_20 varchar,
_C_21 varchar,
_C_22 numeric(9,0),
_TTC_1 byte,
_TTC_2 numeric(9,0),
SDBL_IDENTITY int
);
""")
f.write('ROLLBACK;\n')

if __name__ == "__main__":
main ()
```

4) Run the commands in the terminal `postgres @ tantor :~$ :`

```
python3 ./gen.py
psql -c "vacuum full pg_class;"
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
psql -c "select pg_table_size('pg_class');"
time psql -f workload1c_5000.sql > /dev/null
psql -c "select pg_table_size('pg_class');"
time psql -f workload1c_10000.sql > /dev/null
psql -c "select pg_table_size('pg_class');"
time psql -f workload1c_25000.sql > /dev/null
psql -c "select pg_table_size('pg_class');"
```

The teams will give the result:

```
VACUUM
 pg_table_size
---------------
98304
(1 row)


real 0m10.947s
user 0m0.920s
sys 0m0.137s
pg_table_size
---------------
3932160
(1 row)


real 0m30.293s
user 0m2.695s
sys 0m0.384s
pg_table_size
---------------
10960896
(1 row)


real 1m27.637s
user 0m8.314s
sys 0m2.146s
pg_table_size
---------------
22822912
(1 row)
```

After vacuuming, the system catalog table takes up 98304 bytes.
The gen . py program created three command files: `workload1c_25000.sql` `workload1c_10000.sql` `workload1c_5000.sql` . In these scripts, 5000, 10000, 25000 temporary tables with 25 columns are created in one transaction, after which the transaction is rolled back.
After running these scripts, the size of the pg_class table is measured.
For 25000 objects the size of the pg_class table becomes 22MB.

5) Do it command :

```
postgres@tantor:~$ psql -c "select pg_table_size('pg_class');"
 pg_table_size
---------------
131072
```

`pg_class` table has decreased to 131K. What caused the table to decrease in size? As a result of autovacuum. If permanent tables had been created in another session after the tables had been created, would the `pg_class table` have decreased in size? Unlikely.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Oleg Ivanov

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Practice for Chapter 5

Part 1. Object locks

1) Number of object locks and advisory locks (advisory locks (which are not used automatically , their use requires programming) on an instance is determined by the product max_locks_per_transaction * max_connections (it is assumed that max_prepared_transactions = 0 and does not need to be changed) .
By default, **max_locks_per_transaction=64 and max_connections=100:**

```
postgres=# show max_locks_per_transaction;
max_locks_per_transaction
---------------------------
64
(1 row)
postgres=# show max_connections;
max_connections
-----------------
100
(1 row )
```

2) To prevent typos in commands inside a transaction from causing the transaction to fail, set the parameter:

```
postgres=# \set ON_ERROR_ROLLBACK INTERACTIVE
```

By default, any typo in the command puts the transaction into a failure state and the transaction cannot be continued, only rolled back. The **ON _ ERROR _ ROLLBACK command INTERACTIVE** specifies to implicitly set a savepoint after each command in a transaction and roll back to the last savepoint if the command in the transaction failed to execute (a typo is enough). The INTERACTIVE option sets this rule for interactive command entry and does not set savepoints when executing scripts. Therefore, this setting is both convenient and safe.

3) See how much memory is allocated to the server process:

```
postgres=# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
sum | sum | sum
---------+--------+--------
1329320 | 947688 | 381632
(1 row )
```

1-2MB is not that much. If more than 2MB is given out, reconnect.

4) Enable the output of command execution time to know the duration of execution:

```
postgres=# \timing on
Timing is on.
```

Measuring time is useful for visualizing how long it takes for commands to execute.

5) Create sectioned table :

```
postgres=# create table parttab(n numeric,k numeric, v varchar(100)) partition by
range (n);
CREATE TABLE
Time: 0.976 ms
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

6) Get started transaction :

```
postgres=# begin transaction;
BEGIN
Time: 0.150 ms
```

7) Perform anonymous plpgsql block :

```
postgres=*# do
$$
declare
cnt integer;
vvarchar(200);
begin
for i in 0..4000 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;

DO
Time: 67764.380 ms (01:07.764)
```

The block was executed for 67 seconds.

8) See how much memory is allocated to the server process:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum  |  sum  |  sum
----------+----------+---------
 57673256 | 47280560 | 10392696
(1 row)
```

57 MB allocated , which is quite a lot.

9) Look content tables blockings :

```
postgres=*# select locktype, relation, relation::regclass name, mode, granted,
fastpath from pg_locks order by relation limit 9;
locktype | relation | name | mode | granted
----------+----------+-------------------------------+---------------------+---------
relation | 12073 | pg_locks | AccessShareLock | t
relation | 12138 | pg_backend_memory_contexts | AccessShareLock | t
relation | 2943194 | parttab | AccessExclusiveLock | t
relation | 2943197 | parttab_0 | AccessExclusiveLock | t
relation | 2943200 | pg_toast.pg_toast_2943197 | ShareLock | t
relation | 2943201 | pg_toast.pg_toast_2943197_index | AccessExclusiveLock | t
relation | 2943202 | parttab_1 | AccessExclusiveLock | t
relation | 2943205 | pg_toast.pg_toast_2943202 | ShareLock | t
relation | 2943206 | pg_toast.pg_toast_2943202_index | AccessExclusiveLock | t
(9 rows)

Time: 27.150 ms
```

Installed blocking types `AccessShare` , `AccessExclusive` , `Share` .
10) Look number blockings :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
postgres=# select count(*) from pg_locks;
select count(distinct relation) from pg_locks;
count
-------
12009
(1 row)

Time: 11.052 ms
count
-------
 12006
(1 row )
```

One lock is set for each relation .

```
postgres=# select locktype, relation, mode, granted, fastpath from pg_locks where
fastpath=true;
locktype | relation | mode | granted | fastpath
---------------+----------+----------------+---------+----------
virtualxid | | ExclusiveLock | t | t
(1 row)
```

Only one lock is acquired via the fastpath . Locks acquired via the fastpath do not take up space in the overall lock structure.

```
postgres=# select mode, count(mode) from pg_locks group by mode;
mode | count
---------------------+-------
ShareLock | 4001
AccessExclusiveLock | 8003
ExclusiveLock | 2
AccessShareLock | 3
(4 rows)
```

Number blocking V common structure blocking `12008` And significantly exceeds **`max_locks_per_transaction * max_connections = 64*100 = 6400`** . Memory is allocated in powers of two. Also, fastpath locks are not included in the limit.

11) Roll back transaction :

```
postgres=*# rollback;
ROLLBACK
Time: 588.883 ms
```

12) Check that the locks have been removed:

```
postgres=# select locktype, relation::regclass relation, virtualxid,
virtualtransaction, mode, fastpath from pg_locks;
locktype | relation | virtualxid | virtualtransaction | mode | fastpath
------------+----------+-----------+-----+-----------------+----------
relation | pg_locks | | 3/11 | AccessShareLock | t
virtualxid | | 3/11 | 3/11 | ExclusiveLock | t
(2 rows )
```

Part 2. Monitoring the server process memory

1) Get started transaction :

```
postgres=# begin transaction;
```

```
BEGIN
Time: 0.150 ms

postgres= * # do
$$
declare
cnt integer;
vvarchar(200);
begin
in 0..4291 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;

ERROR: out of shared memory
HINT: You might need to increase max_locks_per_transaction.
CONTEXT: SQL statement "create table parttab _4290 partition of parttab for
values from (42900) to (42910)"
PL/pgSQL function inline_code_block line 8 at EXECUTE
Time : 72606.127 ms (01:12.606)
```

Anonymous block could not add section 4290 to table parttab because there was not enough space in the shared memory structure for locks.

2) Roll back transaction :

```
postgres= ! # rollback;
ROLLBACK
Time: 588.883 ms
```

Rolling back a transaction frees the process local memory contexts allocated in the transaction context.

3) `postgres=# \d parttab`

```
Partitioned table "public.parttab"
Column | Type | Collation | Nullable | Default
--------+---------+----------+----------+---------
n | numeric | | |
k | numeric | | |
v | character varying(100) | | |
Partition key: RANGE (n)
Number of partitions: 0
```

No sections were added.

4) Start the transaction:

```
postgres=# begin transaction;
BEGIN
Time: 0.150 ms
```

5) Reduce the number of sections to 2600 and execute the block:

```
postgres=*# do
$$
declare
cnt integer;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
vvarchar(200);
begin
for i in 0..2600 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;

CREATE TABLE
DO
Time: 35644.365 ms (00:35.644)

postgres=# \d parttab
Partitioned table "public.parttab"
Column | Type | Collation | Nullable | Default
--------+---------+----------+----------+---------
n | numeric | | |
k | numeric | | |
v | character varying(100) | | |
Partition key: RANGE (n)
Number of partitions: 2601 (Use \d+ to list them.)
```

2601 sections were created .

6) Try deleting the table:

```
postgres=*# drop table if exists parttab;
ERROR: out of shared memory
HINT: You might need to increase max_locks_per_transaction.
Time : 222.310 ms
```

too many locks. If the transaction were committed, the max_connections or max_locks_per_transaction parameter would have to be increased . Increasing either of these parameters requires first increasing them on the replicas and restarting the instance .

7) You can work with the table. Do the following: commands :

```
postgres=*# insert into parttab select generate_series(1, 10000-1),10000-
generate_series(1, 10000-1),'Initial value '||generate_series(1,
10000-1);
INSERT 0 9999
Time : 119.122 ms
```

8) See how much memory is allocated to the process:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
   sum | sum | sum
----------+----------+---------
57576184 | 36993936 | 20582248
(1 row)
```

The size of memory allocated to the process increased slightly - less than 1 MB from 57 **6** 73256 to 57 **5** 76184 .

9) Complete commands :

```
postgres=*# prepare prepstmt1 (int) as select v from parttab where n=$1;
PREPARE
Time: 0.700 ms

postgres=*# execute prepstmt1(1);
v
-----------------
Initial value 1
(1 row)

Time: 0.441 ms
```

10) See how much memory is allocated to the process:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
sum | sum | sum
----------+----------+---------
57598712 | 37014832 | 20583880
(1 row)

Time: 2.493 ms
```

The process memory size has increased slightly.

11) Create an index on the table and check how the amount of memory allocated to the process has changed:

```
postgres=*# create unique index parttabn on parttab(n);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
CREATE INDEX
Time: 28621.034 ms (00:28.621)
sum | sum | sum
----------+----------+----------
61139752 | 47002536 | 14137216
(1 row )

Time : 5.722 ms
```

Process memory size increased by ~4MB c 57598712 to 61139752 .

12) Run the command that is needed to make the planner think that there are more rows in the table than there are:

```
update pg_class set reltuples=reltuples*100 where relname like 'parttab%';
UPDATE 5204
Time : 245.317 ms
```

13) Next, execute the prepared command 6 times with a measurement of the memory allocated to the process:

```
postgres=*# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(1000);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(5000);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(10000);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(100);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(300);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
execute prepstmt1(2000);
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
sum | sum | sum
----------+----------+----------
61314488 | 47147896 | 14166592
(1 row)

Time: 5.532 ms
v
--------------------
Initial value 1000
(1 row)

Time: 28.541 ms
sum | sum | sum
----------+----------+----------
61470504 | 49987992 | 11482512
(1 row)

Time: 5.344 ms
v
--------------------
Initial value 5000
(1 row)

Time: 0.433 ms
sum | sum | sum
----------+----------+----------
61471528 | 49988816 | 11482712
(1 row)

Time: 5.606 ms
v
---
(0 rows)

Time: 0.370 ms
sum | sum | sum
----------+----------+----------
61471528 | 49989528 | 11482000
(1 row)

Time: 5.772 ms
v
-------------------
Initial value 100
(1 row)

Time: 0.388 ms
sum | sum | sum
----------+----------+----------
61472552 | 49990352 | 11482200
(1 row)

Time: 5.190 ms
v
-------------------
Initial value 300
(1 row)

Time: 208.855 ms
sum | sum | sum
----------+----------+----------
 72232560 | 58599344 | 13633216
(1 row)

Time: 5.309 ms
v
--------------------
Initial value 2000
(1 row)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
Time: 0.315 ms
sum | sum | sum
----------+----------+----------
 72232560 | 58599344 | 13633216
(1 row )

Time : 5.009 ms
```

On the sixth execution of the prepared query (after preparation), the server process switched to a generic execution plan. The plan size is 10 MB, which is larger than any of the private plans, since it includes all sections of the table. The process memory size increased from 61472552 to **72232560** bytes.

14) Look at what memory contexts there are and their size:

```
postgres=*# with recursive dep as (select name, total_bytes as total, left(ident,10)
ident, 1 as level, left(name,38) as path from pg_backend_memory_contexts where parent is
null
union all
select c.name, c.total_bytes, left(c.ident,10), p.level+1, left(p.path||'->'||c.name,38)
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select * from dep limit 40;
name | total | ident | level | path
-------------------------+----------+------------+-------+---------------------------------------
TopMemoryContext | 4259200 | | 1 | TopMemoryContext
Btree proof lookup cache | 8192 | | 2 | TopMemoryContext->Btree proof lookup c
Prepared Queries | 16384 | | 2 | TopMemoryContext->Prepared Queries
TopTransactionContext | 6394000 | | 2 | TopMemoryContext->TopTransactionContex
TableSpace cache | 8192 | | 2 | TopMemoryContext->TableSpace cache
RegexpCacheMemoryContext | 1024 | | 2 | TopMemoryContext->RegexpCacheMemoryCon
Type information cache | 24368 | | 2 | TopMemoryContext->Type information ca
Operator lookup cache | 24576 | | 2 | TopMemoryContext->Operator lookup cache
PLpgSQL cast expressions | 8192 | | 2 | TopMemoryContext->PLpgSQL cast express
CFuncHash | 8192 | | 2 | TopMemoryContext->CFuncHash
Rendezvous variable hash | 8192 | | 2 | TopMemoryContext->Rendezvous variable
PLpgSQL function hash | 8192 | | 2 | TopMemoryContext->PLpgSQL function has
Record information cache | 8192 | | 2 | TopMemoryContext->Record information c
RowDescriptionContext | 8192 | | 2 | TopMemoryContext->RowDescriptionContex
MessageContext | 131072 | | 2 | TopMemoryContext->MessageContext
Operator class cache | 8192 | | 2 | TopMemoryContext->Operator class cache
PgStat Shared Ref Hash | 394288 | | 2 | TopMemoryContext->PgStat Shared Ref Ha
PgStat Shared Ref | 573440 | | 2 | TopMemoryContext->PgStat Shared Ref
PgStat Pending | 2146304 | | 2 | TopMemoryContext->PgStat Pending
smgr relation table | 2097152 | | 2 | TopMemoryContext->smgr relation table
TransactionAbortContext | 32768 | | 2 | TopMemoryContext->TransactionAbortCont
Portal hash | 8192 | | 2 | TopMemoryContext->Portal hash
TopPortalContext | 8192 | | 2 | TopMemoryContext->TopPortalContext
Relcache by OID | 1048576 | | 2 | TopMemoryContext->Relcache by OID
CacheMemoryContext | 34046240 | | 2 | TopMemoryContext->CacheMemoryContext
WAL record construction | 50200 | | 2 | TopMemoryContext->WAL record construct
PrivateRefCount | 8192 | | 2 | TopMemoryContext->PrivateRefCount
MdSmgr | 131072 | | 2 | TopMemoryContext->MdSmgr
LOCALOCK hash | 2097152 | | 2 | TopMemoryContext->LOCALLOCK hash
GUCMemoryContext | 24576 | | 2 | TopMemoryContext->GUCMemoryContext
Timezones | 104112 | | 2 | TopMemoryContext->Timezones
ErrorContext | 8192 | | 2 | TopMemoryContext->ErrorContext
CurTransactionContext | 8192 | | 3 | TopMemoryContext->TopTransactionContex
Combo CIDs | 524288 | | 3 | TopMemoryContext->TopTransactionContex
RegexpMemoryContext | 13360 | ^(parttab) | 3 | TopMemoryContext->RegexpCacheMemoryCon
PortalContext | 1024 | <unnamed> | 3 | TopMemoryContext->TopPortalContext->Po
CachedPlan | 8551240 | prepare pr | 3 | TopMemoryContext->CacheMemoryContext->
partition descriptor | 153968 | parttab | 3 | TopMemoryContext->CacheMemoryContext->
partition key | 1024 | parttab | 3 | TopMemoryContext->CacheMemoryContext->
index info | 1024 | parttab_26 | 3 | TopMemoryContext->CacheMemoryContext->
(40 rows )
```

15) In another terminal, run psql:

```
astra@tantor:~$ psql
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru


```
psql
Type "help" for help.

postgres=#
```

16) In the third terminal, see how much memory is occupied by the operating system:

```
postgres@tantor:~$ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep postgres:
336508 191924 143715 3 2662 10224 101373 postgres : postgres postgres [local] idle in transaction
235100 94116 51708 3 24 22597 101196 postgres: checkpointer
230112 16804 3926 3 54 1 102783 postgres : postgres postgres [local] idle
229580 11812 2216 0 204 1 101200 postgres: autovacuum launcher
227480 10660 1879 1 10 1 101201 postgres: logical replication launcher
226296 14628 2800 0 29 1 101197 postgres: background writer
225908 31104 11337 0 28807 45318 101195 /opt/tantor/db/16/bin/postgres -D /var/lib/postgresql/tantor-se-16/data
225908 11940 2232 0 71 1554 101199 postgres : walwriter
```

In the operating system, the server process is allocated `336508 KB` of virtual address space. Of these, the resident size (together with the executable code) is the amount of memory allocated to the process and in physical memory RSS = `191924 KB` , the proportional set (the share of executable code of shared libraries) PSS = `144852 KB` .

The second server process, in which no commands were executed, was allocated `230112 KB` of virtual address space. Of these, the resident size in physical memory RSS= `16804 KB` , the proportional size PSS= `3926 KB` .

The `pg_backend_memory_contexts` view shows that the process is using `72232560 bytes` .

17) Roll back the transaction opened in the first terminal:

```
postgres=*# rollback;
ROLLBACK
Time: 488.252 ms
```

18) Delete table :

```
postgres=# drop table if exists parttab;
DROP TABLE
```

19) See if the memory has been freed:

```
postgres=# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from pg_backend_memory_contexts;
sum | sum | sum
----------+----------+----------
 56052848 | 12087192 | 43965656
(1 row )
```

Part of the memory was freed up and returned to the size measured in point 8 of part 1 of the practice.

20) See how much memory is allocated according to the operating system:

```
postgres@tantor:~$ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep postgres
3 32 408 189380 140903 2 2664 10224 101373 postgres: postgres postgres [local] idle
2 35 100 94244 51706 3 29 22597 101196 postgres: checkpointer
230112 16804 3922 1 61 1 102783 postgres: postgres postgres [local] idle
229580 11812 2216 0 229 1 101200 postgres: autovacuum launcher
```

```
227480 10788 1903 1 12 1 101201 postgres: logical replication launcher
226296 14628 2777 1 30 1 101197 postgres: background writer
225908 31104 11337 0 28807 45318 101195 /opt/tantor/db/16/bin/postgres -D
/var/lib/postgresql/tantor-se-16/data
225908 11940 2231 2 78 1566 101199 postgres : walwriter
```

It can be estimated that approximately 95 MB was freed up (`3 32 408 - 2 35 100 = 97308 KB`).

20) See how much memory is free and occupied:

```
postgres@tantor:~$ free
total used free shared buff/cache available
Mem : 4020796 12 4 5216 1 5 27140 135824 1675284 2 7 75580
Swap : 0 0 0
```

21) Close the psql utility in which you created the partitioned table to break the session and the server process to exit and free the local memory, which according to its data it occupies `56052848` and run the **`free command`** :

```
postgres=# exit
postgres@tantor:~$ free
total used free shared buff/cache available
Mem : 4020796 11 3 4344 1 6 38008 135824 1675288 2 8 86452
Swap : 0 0 0
```

110 MB freed up .

22) See how much memory the ps utility shows:

```
postgres@tantor:~$ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep postgres
2 35 100 94244 82711 3 29 22897 101196 postgres: checkpointer
230112 16804 4995 1 61 1 102783 postgres: postgres postgres [local] idle
229580 11812 2611 0 233 1 101200 postgres: autovacuum launcher
227480 10788 2146 1 12 1 101201 postgres: logical replication launcher
226296 14628 3345 2 30 1 101197 postgres: background writer
225908 31104 11337 0 28807 45318 101195 /opt/tantor/db/16/bin/postgres -D
/var/lib/postgresql/tantor-se-16/data
225908 11940 2787 0 78 1566 101199 postgres : walwriter
6980 2304 414 2 13 0 103846 grep postgres :
```

The local memory of the stopped server process was not counted in the memory of other processes. Local memory is not counted in the memory of other processes, only the shared memory that the process has access to. The stopped server process showed a memory of `3 32 408` . Of this, local memory according to the operating system was 110 MB, according to the process `56052848` bytes . When allocating memory, Linux uses buddy allocation.

Part 3. Temporary tables and files

1) Look at the parameters for temporary files and their values:

```
postgres=# \dconfig temp*
List of configuration parameters
Parameter | Value
-------------------+-------
temp_buffers | 8MB
temp_file_limit | -1
 temp_tablespaces |
(3 rows)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

By default, the local buffer cache size is 8MB. There is no limit on the size of temporary files and no tablespaces are set for temporary files.

2) Create temporary table :

```
postgres=# create temp table temp1 (id integer);
CREATE TABLE
```

3) Look at the path to the first file of the main layer of the temporary table:

```
postgres=# select pg_relation_filepath('temp1');
pg_relation_filepath
----------------------
base/5/t3_456929
(1 row )
```

The file is located in the pg_default tablespace directory.

4) Insert 10000 rows into temporary table temp 1:

```
postgres=# insert into temp1 select * from generate_series(1, 10000);
INSERT 0 10000
```

5) Look at the size of the temp1 table files:

```
postgres=# \! ls -al $PGDATA/base/5/t*
-rw------- 1 postgres postgres 368640 /var/lib/postgresql/tantor-se-
16/data/base/5/ t3_456932
-rw------- 1 postgres postgres 24576 /var/lib/postgresql/tantor-se-
16/data/base/5/ t3_456932_fsm
```

The lines are placed in a file consisting of 45 blocks of 8Kb: 368640/8192=45. The free space map consists of 3 blocks.

6) See how many rows are in the temporary table block:

```
postgres=# select count(*) from temp1 where ctid::text like '(43,%)';
 count
-------
226
(1 row)
```

`ctid` - this is a service column present in regular tables (heap table). ctid is the address of the row. The first number is the block number, the second number is the row number in the block. Block numbering starts with zero. All blocks except the last one contain 226 rows. The last block contains 56 rows. 226 is the maximum number of non-empty rows that can fit in a block of a regular Tantor SE DBMS table.

7) Run the command:

```
postgres=# explain (analyze, buffers) select * from temp1;
QUERY PLAN
-----------------------------------------------------------------------
Seq Scan on temp1 (cost=0.00..159.75 rows=11475 width=4) (actual
time=0.013..44.184 rows=10000 loops=1)
   Buffers: local hit =45
Planning Time: 0.028 ms
 Execution Time: 85.129 ms
```

```
(4 rows)
```

Reading from the local buffer cache (`hit`), reading from disk or Linux page cache (`read`), writing (`write`) are displayed in the command execution plan by the word **`local`** . For a shared buffer cache, by the word `shared` .

45 temporary table blocks fit into the local buffer cache, the default size of which is 8MB/8192=1024 buffers. One buffer stores the contents of one block.
The logic of working with a local buffer is similar to the logic of working with a shared buffer cache. There is no such thing that the lines are first in memory (temp _ buffers), and files are not created.

8) Set a limit on the maximum size of temporary files at the session level:

```
postgres=# set temp_file_limit = '1MB';
SET
```

9) Try inserting 1 million lines:

```
postgres=# insert into temp1 select * from generate_series(1, 1000000);
ERROR: temporary file size exceeds temp_file_limit (1024kB)
```

10) Reset the value to default (unlimited):

```
postgres=# reset temp_file_limit;
RESET
```

11) Insert 1 million rows into temporary table temp 1:

```
postgres=# insert into temp1 select * from generate_series(1, 1000000);
INSERT 0 1000000
```

12) Run the command several times:

```
postgres=# explain (analyze, buffers) select * from temp1;
QUERY PLAN
---------------------------------------------------
Seq Scan on temp1 (cost=0.00..15868.50 rows=1139850 width=4) (actual time=0.044..4123.828
rows=1010000 loops=1)
Buffers: local read=4470 dirtied=4426 written=4424
Planning Time: 0.028 ms
Execution Time: 8117.431 ms
(4 rows)

postgres=# explain (analyze, buffers) select * from temp1;
QUERY PLAN
---------------------------------------------------
Seq Scan on temp1 (cost=0.00..15868.50 rows=1139850 width=4) (actual time=0.046..4077.274
rows=1010000 loops=1)
Buffers: local read=4470 written=1024
Planning Time: 0.028 ms
Execution Time: 8069.024 ms
(4 rows)

postgres=# explain (analyze, buffers) select * from temp1;
QUERY PLAN
---------------------------------------------------
Seq Scan on temp1 (cost=0.00..15868.50 rows=1139850 width=4) (actual time=0.022..4077.606
rows=1010000 loops=1)
Buffers: local read=4470
Planning Time: 0.027 ms
Execution Time : 8064.546 ms
(4 rows )
```

13) View the amount of memory used by the server process:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
postgres=# select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
sum | sum | sum
---------+---------+--------
9919784 | 9563824 | 355960
(1 row)
```

Size used memory 9.3 MB .

```
postgres=# with recursive dep as
(select name, total_bytes as total, left(ident,10) ident, 1 as level, left(name,38) as path from
pg_backend_memory_contexts where parent is null
union all
select c.name, c.total_bytes, left(c.ident,10), p.level+1, left(p.path||'->'||c.name,38)
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select * from dep order by total desc limit 3;
name | total | ident | level | path
--------------------+---------+-------+-------+------------------------------------------------
-
 LocalBufferContext | 8425808 | | 2 | TopMemoryContext->LocalBufferContext
CacheMemoryContext | 524288 | | 2 | TopMemoryContext->CacheMemoryContext
MessageContext | 262144 | | 2 | TopMemoryContext->MessageContext
(3 rows)
```

Local cache buffers has Name context `LocalBufferContext` takes up 8 MB . This context takes up most of the server process's local memory.

You can use another terminal to see what the occupied memory looks like from the point of view of the operating system utilities:

```
postgres@tantor:~ $ ps -e -o vsz,rss,pss,psr,rops,wops,pid,cmd --sort -vsz | grep postgres
VSZ RSS PSS PSR ROPS WOPS PID CMD
238624 28316 15916 2 15040 14988 118376 postgres: postgres postgres [local] idle
235100 95140 82302 2 29 25157 101196 postgres: checkpointer
229580 11812 2593 1 1664 1 101200 postgres: autovacuum launcher
227480 10788 2107 1 12 1 101201 postgres: logical replication launcher
226296 14628 3374 2 39 1 101197 postgres: background writer
225908 31104 9787 0 91480 62149 101195 /opt/tantor/db/16/bin/postgres -D
/var/lib/postgresql/tantor-se-16/data
225908 11940 2639 3 106 1607 101199 postgres : walwriter
 79868 6332 937 1 486 538 116927 postgres : 15/ main : logger
```

Knowing that the server process is allocated `9919784` bytes, we can see which of the indicators can be useful if we monitor the memory with the operating system utilities. Perhaps `238624 - 225908 = 12716` , perhaps PSS = **`15916`** . None of the indicators accurately reflect the local memory allocated to the process. VSZ (virtual memory size) shows the memory to which the process has access. Almost all processes in the instance have access to a shared pool of 128 MB and other shared memory structures with a total size of about ~200 MB. Using the VSZ column value to estimate the memory allocated to the process is not informative. If we subtract the total memory of shared structures from the column value, we can get an estimate of the size of the local memory of the process. Which process should we take as a basis? Perhaps postmaster: `225908` .
In the example output of the background process `logger` `VSZ ( VmSize )= 79868` , which means that the process is not currently working with shared memory structures of the instance. At the same time, the `logger process VmPeak = 264500` .
You can use more detailed metrics on the process's memory usage:

```
postgres@tantor:~$ cat /proc/101195/status | head -33 | tail -16
VmPeak : 225908 kB
VmSize : 225908 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 31104 kB
VmRSS: 31104 kB
RssAnon: 2560 kB
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
RssFile: 17408 kB
RssShmem: 11136 kB
VmData: 1748 kB
VmStk: 132 kB
VmExe: 5240 kB
VmLib: 19096 kB
VmPTE: 180 kB
VmSwap: 0 kB
HugetlbPages : 0 kB
```

14) Look at the size of the files of the temp table 1:

```
postgres=# \! ls -al $PGDATA/base/5/t*
-rw------- 1 postgres postgres 36618240 /var/lib/postgresql/tantor-se-16/data/base/5/t3_456983
-rw------- 1 postgres postgres 32768 /var/lib/postgresql/tantor-se-16/data/base/5/t3_456983_fsm
```

15) Close the psql utility to end the session with the database:

```
postgres =# \ q
```

16) Look at the size of the temp1 table files:

```
postgres@tantor:~$ ls -al $PGDATA/base/5/t*
ls: cannot access '/var/lib/postgresql/tantor-se-16/data/base/5/t*': No such file
or directory
```

The files have been deleted. Temporary files are deleted automatically.

17) Send the command to execute, which will generate 5 GB of temporary files during execution:

```
postgres=# explain (analyze, buffers)
with recursive t as (
select 0 i, '' s
union all
select i + 1, repeat('a', i + 1) from t where i < 1e5 -- 100000
)
table t order by s desc limit 1;
```

18) in another terminal, run psql and execute the command at intervals of 1-2 minutes:

```
postgres=# select * from pg_ls_tmpdir();
name | size | modification
-------------------+-----------+-----------------------
pgsql_tmp119544.1 | 1073741824 | 2035-12-12 15: 12:59 +03
pgsql_tmp119544.2 | 22331392 | 2035-12-12 15:13:00+03
(2 rows)

postgres=# select * from pg_ls_tmpdir();
name | size | modification
-------------------+-----------+-----------------------
pgsql_tmp119544.1 | 1073741824 | 2035-12-12 15: 12:59 +03
pgsql_tmp119544.2 | 1073741824 | 2035-12-12 15:14:29+03
pgsql_tmp119544.3 | 1073741824 | 2035-12-12 15:15:08+03
pgsql_tmp119544.4 | 1073741824 | 2035-12-12 15: 16:17 +03
pgsql_tmp 119544.5 | 375881728 | 2035-12-12 15:16:45+03
(5 rows )
```

By default, temporary files up to 1GB in size appear in the tablespace's temporary files directory.

19) After the command completes its work after 300 seconds or is interrupted by the key combination < ctrl + c > the files will be deleted:

```
                            QUERY PLAN
-----------------------------------------------------------------------------------
 Limit (cost=3.82..3.82 rows=1 width=36) (actual time=296625.350..296625.398 rows=1 loops=1)
   Buffers: temp written=610577
CTE t
-> Recursive Union (cost=0.00..3.04 rows=31 width=36) (actual time=0.011..66081.120 rows=100001 loops=1)
-> Result (cost=0.00..0.01 rows=1 width=36) (actual time=0.004..0.011 rows=1 loops=1)
-> WorkTable Scan on t t_1 (cost=0.00..0.27 rows=3 width=36) (actual time=0.636..0.641 rows=1 loops=100001)
Filter: ((i)::numeric < '100000'::numeric)
Rows Removed by Filter: 0
-> Sort (cost=0.78..0.85 rows=31 width=36) (actual time=296625.343..296625.355 rows=1 loops=1)
Sort Key: ts DESC
Sort Method: top-N heapsort Memory: 122kB
         Buffers: temp written=610577
-> CTE Scan on t (cost=0.00..0.62 rows=31 width=36) (actual time=0.018..212744.403 rows=100001 loops=1)
               Buffers: temp written=610577
Planning Time: 0.061 ms
Execution Time : 299594.624 ms
(16 rows )
```

The request was executed in 299 seconds. 610577 blocks were written = ~5GB.

```
postgres=# SELECT * FROM pg_ls_tmpdir();
 name | size | modification
------+------+--------------
(0 rows)
```

20) Check that temporary files are limited. Set a maximum size limit for temporary files at the session level:

```
postgres=# set temp_file_limit = '1MB';
SET
```

21) Repeat command :

```
postgres=# explain (analyze, buffers)
with recursive t as (
select 0 i, '' s
union all
select i + 1, repeat('a', i + 1) from t where i < 1e5 -- 100000
)
table t order by s desc limit 1;
ERROR: temporary file size exceeds temp_file_limit (1024kB)
```

The restriction is in effect.

Part 4. Impact of configuration parameters on shared memory

1) Look at the size of the shared memory structures of the instance with the names < anonymous > and an empty name (NULL):

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations ORDER BY size DESC limit 4;
name | allocated_size | pg_size_pretty
---------------+---------------+----------------
Buffer Blocks | 134221824 | 128 MB
<anonymous> | 4946048 | 4830 kB
XLOG Ctl | 4208256 | 4110 kB
              | 1924224 | 1879 kB
(4 rows )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

2) Run a query so that you can later compare how the sizes of the instance's shared memory structures change when you change the configuration parameters:

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations where name like '%LOCK%' or name like '%roc%';
name | allocated_size | pg_size_pretty
-------------------------+----------------+----------------
PROCLOCK hash | 2944 | 2944 bytes
 Proc Header | 256 | 256 bytes
PREDICATELOCKTARGET hash | 2944 | 2944 bytes
ProcSignal | 11392 | 11 kB
Proc Array | 640 | 640 bytes
PREDICATELOCK hash | 2944 | 2944 bytes
LOCK hash | 2944 | 2944 bytes
(7 rows )
```

The sizes are not serious. The instance can run `max _ connections = 100` server processes. **`256 bytes`** `/100` = 2.56 bytes and even **`640 bytes`** `/100 =` 6.4 bytes can hardly save anything useful about the process.

3) Run the command to see the actual sizes of memory structures:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a where name like '%LOCK%' or name like
'%roc%';
name | off | size | allocated_size | true_size
-------------------------+-----------+-------+----------------+-----------
 LOCK hash | 142667776 | 2896 | 2944 | 695168
 PROCLOCK hash | 143362944 | 2896 | 2944 | 695168
PREDICATELOCKTARGET hash | 144062336 | 2896 | 2944 | 445312
PREDICATELOCK hash | 144507648 | 2896 | 2944 | 1260416
 Proc Header | 146621568 | 136 | 256 | 152192
Proc Array | 146773760 | 544 | 640 | 640
ProcSignal | 147177984 | 11272 | 11392 | 11392
(7 rows )
```

`'s check how increasing the value of the max_locks_per_transaction parameter` from 64 to 10000 will affect the size of shared memory `structures .`
Increase the parameter value and reload the instance:

```
postgres=# alter system set max_locks_per_transaction = 10000;
ALTER SYSTEM
postgres=# \q
astra@tantor:~$ sudo restart
astra @ tantor :~$ psql
```

5) Look how the size of the memory structures has changed:

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations ORDER BY size DESC limit 4;
name | allocated_size | pg_size_pretty
---------------+----------------+----------------
| 270582272 | 258 MB
<anonymous> | 228811520 | 218 MB
Buffer Blocks | 134221824 | 128 MB
XLOG Ctl | 4208256 | 4110 kB
(4 rows )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The size of memory structures named < anonymous > has increased from 4830 kB up to 218 MB .
< anonymous > is one of the types of memory allocation. Anonymous memory is allocated in RAM, has no mapping to files and devices.
Memory with an empty name (NULL) is written as "not used".

6) Look at the size of the LOCK and Proc structures :

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a where name like '%LOCK%' or name like
'%roc%';
name | off | size | allocated_size | true_size
--------------------------+------------+--------+----------------+-----------
 LOCK hash | 142667776 | 66384 | 66432 | 110935552
 PROCLOCK hash | 253603328 | 131920 | 131968 | 114512768
PREDICATELOCKTARGET hash | 368120320 | 2896 | 2944 | 445312
PREDICATELOCK hash | 368565632 | 2896 | 2944 | 1260416
 Proc Header | 370679552 | 136 | 256 | 152192
Proc Array | 370831744 | 544 | 640 | 640
ProcSignal | 371235968 | 11272 | 11392 | 11392
(7 rows)
```

Size structures PROCLOCK hash And LOCK HASH increased . The increase in hash-type memory structures is usually not critical. These structures are used as auxiliary ones: for fast searching of pointers to other structures. There are many ways of searching. For example, lists, b-tree, radix-tree. Hash structures are notable for the fact that the least number of processor cycles are used for searching them. The main disadvantage is that hash structures take up much more memory than other types of structures for searching. Therefore, if developers use a hash structure, this means that the size does not matter from the point of view of performance.

Proc structure size Header did not change when increased. This structure is often polled by processes and most likely it will occupy processor caches or actively "compete" for them. In addition to the size, the composition of the structure is important: the alignment of fields in the records of the structure.

7) Reset max_locks_per_transaction to default and increase max_connections up to 10000 :

```
postgres=# alter system reset max_locks_per_transaction;
ALTER SYSTEM
postgres=# alter system set max_connections = 10000;
ALTER SYSTEM
postgres=# \q
astra@tantor:~$ sudo restart
astra @ tantor :~$ psql
```

8) Look how the size of the memory structures has changed:

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations ORDER BY size DESC limit 4;
name | allocated_size | pg_size_pretty
------------------------+----------------+----------------
<anonymous> | 328024448 | 313 MB
| 156285824 | 149 MB
Buffer Blocks | 134221824 | 128 MB
Backend Activity Buffer | 10268672 | 10028 kB
(4 rows)
```

The size of memory structures named < anonymous > has increased from 4830 kB up to 313 MB .

be verified that increasing max_connections without increasing the number of sessions (server processes) does not affect TPS .

9) Look at the size of the LOCK and Proc structures :

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a where name like '%LOCK%' or name like
'%roc%';
name | off | size | allocated_size | true_size
--------------------------+-----------+--------+----------------+-----------
LOCK hash | 144093440 | 33616 | 33664 | 58106752
 PROCLOCK hash | 202200192 | 66384 | 66432 | 59769728
PREDICATELOCKTARGET hash | 261974144 | 33616 | 33664 | 39211904
PREDICATELOCK hash | 301186048 | 66384 | 66432 | 109208448
 Proc Header | 458218240 | 136 | 256 | 11982848
Proc Array | 470201088 | 40144 | 40192 | 40192
ProcSignal | 496702080 | 882472 | 882560 | 882560
(7 rows)
```

After increases max_connections structures shared memory Proc increased V sizes .

Part 5. max_connections parameter and performance

1) Let's check how the increase in the number of idle server processes will affect TPS.
Create test script :

```
postgres@tantor:~$ echo "select * from pg_sleep(1) ;" > test1.sql
```

2) Prepare the cluster for tests:

```
postgres @ tantor :~$
psql -p 5435 -c "alter system set max_connections=100"
psql -p 5435 -c "alter system set max_locks_per_transaction = 64"
psql -c "alter system set max_connections=100"
psql -c "alter system set max_locks_per_transaction = 64"
pgbench -i 2> /dev/null
pgbench -i -p 5435 2> /dev/null
sudo systemctl restart postgresql
sudo systemctl restart tantor-se-server-16
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
```

3) Get baseline metrics:

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6709 .693060 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 7119 .172960 (without initial connection time)
```

U DBMS Tantor SE tps = 7119 .
PostgreSQL DBMS , which comes standard with Astralinux tps = 6709 .

Only the SELECT test will be used, since tests with table changes will reduce accuracy during repeated runs. Since the virtual machine has little memory (4 GB), the effect of memory structure sizes begins at about the same time that Huge Pages begin to have an effect (~100 GB), so the differences are expected to be minimal. Test intervals are chosen to be small (30 seconds), since there is no point in wasting course time on waiting.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Practice is useful not for absolute values, but for the procedure of measurement.

4) Currently the maximum number of sessions is 100. Create 90 sessions using pgbench :

```
postgres@tantor:~$ pgbench -T 10000 -P 5 -c 90 -f test1.sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 72.1 tps, lat 1002.351 ms stddev 1.874, 0 failed
progress: 10.0 s, 90.0 tps, lat 1001.731 ms stddev 0.528, 0 failed
progress: 15.0 s, 90.0 tps, lat 1001.525 ms stddev 0.493, 0 failed
```

If you don't need database connections, you can interrupt the test with the key combination **< ctrl + c >**
tps in the test should be shown as identical integers , equal to the number of sessions with a second delay . If they are not shown as integers , this means that there is competition for processor cores and the accuracy of the tests will be lower.

5) In the second window, run a test that only performs SELECT:

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6940 .512696 (without initial connection time)
```

6) Stop the load and start it with another database:

```
<ctrl+c>
postgres@tantor:~$ pgbench -T 10000 -P 5 -c 90 -f test1.sql -p 5435
pgbench (16.2, server 15.6 (Debian 15.6-astra.se2))
starting vacuum...end.
progress: 5.0 s, 71.9 tps, lat 1002.589 ms stddev 2.201, 0 failed
progress: 10.0 s, 90.1 tps, lat 1002.279 ms stddev 1.216, 0 failed
progress: 15.0 s, 90.0 tps, lat 1002.023 ms stddev 1.286, 0 failed
```

7) In the second window, run the test:

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6649 .501401 (without initial connection time)
```

There was some decrease in tps. It can be assumed that the slowdown is caused by checking the status of 90 processes when executing each select from the test with a frequency equal to tps. To check the status, a shared lock on the memory structure is needed. At the same time, every second the server process gets a lock to change its status (transaction number, if set).

8) Let's change the command for the load so that the load sessions have the transaction open for 1 second:

```
postgres@tantor:~$ echo "select *, pg_sleep(1) from pg_current_xact_id();" > test1.sql
```

Next, we'll repeat the tests. In real testing, it's worth restarting the instances to clear the memory structures.

9) Run a load of 90 sessions:

```
postgres @ tantor :~$ pgbench - T 10000 - P 5 - c 90 - f test 1.sql
pgbench (16.2)
starting vacuum...end.
progress: 5.0 s, 71.7 tps, lat 1003.276 ms stddev 2.201, 0 failed
progress: 10.0 s, 90.2 tps, lat 1020.448 ms stddev 36.319, 0 failed
progress: 15.0 s, 90.1 tps, lat 1002.278 ms stddev 1.112, 0 failed
progress: 20.0 s, 90.0 tps, lat 1001.755 ms stddev 0.542, 0 failed
```

10) Launch test :

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6854 .086439 (without initial connection time)
```

11) Stop the load and start it with another database:

```
<ctrl+c>
postgres@tantor:~$ pgbench -T 10000 -P 5 -c 90 -f test1.sql -p 5435
pgbench (16.2, server 15.6 (Debian 15.6-astra.se2))
starting vacuum...end.
progress: 5.0 s, 72.0 tps, lat 1005.050 ms stddev 5.612, 0 failed
progress: 10.0 s, 89.9 tps, lat 1002.030 ms stddev 0.824, 0 failed
progress: 15.0 s, 90.0 tps, lat 1002.266 ms stddev 1.887, 0 failed
```

12) Launch test :

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6763 .148130 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6573 .925660 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6649 .730773 (without initial connection time)
```

The spread of values in PostgreSQL has increased. Several measurements can be made.
Overall, tps has decreased.

13) Stop the load and start it with another database:

```
< ctrl + c >
```

14) Increase max_connections up to 2000 and a delay of up to 10 seconds:

```
postgres@tantor:~$
echo "select *, pg_sleep( 10 ) from pg_current_xact_id();" > test1.sql
psql -p 5435 -c "alter system set max_connections= 2000 "
psql -c "alter system set max_connections= 2000 "
ALTER SYSTEM
ALTER SYSTEM
```

The delay is increased so that the awakening of processes is rare and does not affect the test result. Scheduler timeslots are 10 milliseconds. 4 processors. Without waiting for service, 400 processes can be awakened per second, 4000 processes every 10 seconds. 4000 is more than 2000 processes that can be started with the parameter value `max _ connections = 2000` . Keeping the horizon does not matter, since there are no commands in the test that change the contents of the tables.

15) Stop the instances. The instance memory size has increased by hundreds of megabytes. To avoid running out of memory, instances will be started one at a time.

```
sudo systemctl stop postgresql
sudo systemctl stop tantor-se-server-16
sudo systemctl start tantor-se-server-16
```

16) Run a load of 990 sessions:

```
pgbench - T 10000 - P 20 - c 990 - f test 1.sql
pgbench (16.2)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
starting vacuum...end.
progress: 20.0 s, 49.5 tps, lat 10096.327 ms stddev 27.359, 0 failed
progress: 40.0 s, 99.0 tps, lat 10013.549 ms stddev 6.854, 0 failed
progress: 60.0 s, 99.0 tps, lat 10010.448 ms stddev 3.005, 0 failed
```

17) Launch test :

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6844 .133619 (without initial connection time)
```

18) Stop the load and start it with another database:

```
<ctrl+c>
sudo systemctl stop tantor-se-server-16
sudo systemctl start postgresql
pgbench -T 10000 -P 20 -c 990 -f test1.sql -p 5435
```

19) Launch test :

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6632 .468942 (without initial connection time)
```

There is little or no change when increasing the number of sessions from 90 to 990. PostgreSQL is optimized to work with 4000 sessions.
More interesting are 10,000 sessions, but they require about 20GB of memory.

Example of a test with 10,000 sessions:

```
echo "select *, pg_sleep(100) from pg_current_xact_id();" > test1.sql
psql -p 5435 -c "alter system set max_connections=10240"
psql -c "alter system set max_connections=10240"
sudo systemctl stop postgresql
sudo systemctl stop tantor-se-server-16
sudo systemctl start tantor-se-server-16
```

```
postgres@tantor:~$ free
total used free shared buff/cache available
Mem: 24020472 1280012 22022956 373344 1421396 22740460
Swap: 0 0 0
```

```
postgres@tantor:~$ pgbench -T 10000 -P 30 -c 10000 -f test1.sql
pgbench (16.2)
starting vacuum...end.
progress: 81.6 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 90.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 120.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 150.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed

total used free shared buff/cache available
Mem: 24020472 17913240 5365736 384800 1479104 6107232
Swap: 0 0 0
```

```
postgres@tantor:~$ pgbench -b select-only -T 30 2> /dev/null | grep tps
tps = 6409 .405397 (without initial connection time)
```

With 10000 inactive sessions tps dropped to `6409` .

For PostgreSQL, the instability of the result remains the same:

```
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
```

```
tps = 6692 .845789 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6436 .574882 (without initial connection time)
postgres@tantor:~$ pgbench -b select-only -T 30 -p 5435 2> /dev/null | grep tps
tps = 6547 .961317 (without initial connection time)
```

Part 6. Buffer Cache Size and Buffer Release

1) We automate testing of the instance operation using the example of changing the buffer cache size. Changing some of the configuration parameters requires restarting the instance. In the previous part of the practice, you repeatedly restarted the instance, and this is labor-intensive and leads to errors. Create a script file for the test. We will test the execution speed of commands that create and delete tables.

```
postgres@tantor:~$ mcedit run.sql

CREATE TABLE x(id int);
INSERT INTO x VALUES (1);
DROP TABLE x;
```

2) Create file test :

```
postgres@tantor:~$ mcedit x.sh

#/bin/sh
pg_ctl stop
for x in '128MB' '1GB' '2GB' '4GB' '8GB' '16GB' '18GB'
do
pg_ctl -l /dev/null -o '--shared_buffers='$x'' start
sleep 1
echo tps for $x
psql -c 'select * from (select *, lead(off) over(order by off) - off as true from
pg_shmem_allocations) as a where a.true<>a.allocated_size order by 1;'
pgbench --file= run.sql -j 1-T 10 2> /dev/null | grep tps
pg_ctl stop
 sleep 1
done
```

Remove the values `'4 GB ' '8 GB ' '16 GB ' '18 GB '` as the virtual machine has little memory.

3) Change the execution permissions of the script file:

```
postgres@tantor:~$ chmod +x x.sh
```

4) Run the test file:

```
postgres@tantor:~$ ./x.sh
waiting for server to start.... done
server started
tps for 128MB
name | off | size | allocated_size | true
---------------------------+----------+-------+---------------+----------
LOCK hash | 144128000 | 33616 | 33664 | 59396992
PREDICATELOCK hash | 304476928 | 66384 | 66432 | 111420288
PREDICATELOCKTARGET hash | 264527744 | 33616 | 33664 | 39949184
PredXactList | 415897216 | 88 | 128 | 19703168
Proc Header | 464834560 | 136 | 256 | 12269696
PROCLOCK hash | 203524992 | 66384 | 66432 | 60998528
RWConflictPool | 439937792 | 24 | 128 | 24628992
SERIALIZABLEXID hash | 435600384 | 4944 | 4992 | 4337408
Shared Buffer Lookup Table | 143166464 | 2896 | 2944 | 961408
(9 rows)
```

```
tps = 79 .221247 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 1GB
name | off | size | allocated_size | true
----------------------------+-----------+-------+----------------+-----------
LOCK hash | 1121593088 | 33616 | 33664 | 59396992
PREDICATELOCK hash | 1281942016 | 66384 | 66432 | 111420288
PREDICATELOCKTARGET hash | 1241992832 | 33616 | 33664 | 39949184
PredXactList | 1393362304 | 88 | 128 | 19703168
Proc Header | 1442299648 | 136 | 256 | 12269696
PROCLOCK hash | 1180990080 | 66384 | 66432 | 60998528
RWConflictPool | 1417402880 | 24 | 128 | 24628992
SERIALIZABLEXID hash | 1413065472 | 4944 | 4992 | 4337408
Shared Buffer Lookup Table | 1114202880 | 9040 | 9088 | 7390080
(9 rows)

tps = 74 .513412 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 2GB
name | off | size | allocated_size | true
----------------------------+-----------+-------+----------------+-----------
LOCK hash | 2224302592 | 33616 | 33664 | 59396992
PREDICATELOCK hash | 2384651520 | 66384 | 66432 | 111420288
PREDICATELOCKTARGET hash | 2344702336 | 33616 | 33664 | 39949184
PredXactList | 2496071808 | 88 | 128 | 19703168
Proc Header | 2545009152 | 136 | 256 | 12269696
PROCLOCK hash | 2283699584 | 66384 | 66432 | 60998528
RWConflictPool | 2520112384 | 24 | 128 | 24628992
SERIALIZABLEXID hash | 2515774976 | 4944 | 4992 | 4337408
Shared Buffer Lookup Table | 2209564160 | 17232 | 17280 | 14738304
(9 rows)

tps = 71 .288637 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 4GB
name | off | size | allocated_size | true
----------------------------+-----------+-------+----------------+-----------
LOCK hash | 4429721984 | 33616 | 33664 | 59396992
PREDICATELOCK hash | 4590070912 | 66384 | 66432 | 111420288
PREDICATELOCKTARGET hash | 4550121728 | 33616 | 33664 | 39949184
PredXactList | 4701491200 | 88 | 128 | 19703168
Proc Header | 4750428544 | 136 | 256 | 12269696
PROCLOCK hash | 4489118976 | 66384 | 66432 | 60998528
RWConflictPool | 4725531776 | 24 | 128 | 24628992
SERIALIZABLEXID hash | 4721194368 | 4944 | 4992 | 4337408
Shared Buffer Lookup Table | 4400287104 | 33616 | 33664 | 29434752
(9 rows)

tps = 66 .595864 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 8GB
name | off | size | allocated_size | true
----------------------------+-----------+-------+----------------+-----------
LOCK hash | 8806510976 | 33616 | 33664 | 59396992
PREDICATELOCK hash | 8966859904 | 66384 | 66432 | 111420288
PREDICATELOCKTARGET hash | 8926910720 | 33616 | 33664 | 39949184
PredXactList | 9078280192 | 88 | 128 | 19703168
Proc Header | 9127217536 | 136 | 256 | 12269696
PROCLOCK hash | 8865907968 | 66384 | 66432 | 60998528
RWConflictPool | 9102320768 | 24 | 128 | 24628992
SERIALIZABLEXID hash | 9097983360 | 4944 | 4992 | 4337408
Shared Buffer Lookup Table | 8747683200 | 66384 | 66432 | 58827648
(9 rows)

tps = 59 .421421 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
```

```
tps for 16gb
name | off | size | allocated_size | true
---------------------------+------------+--------+---------------+-----------
LOCK hash | 17560088960 | 33616 | 33664 | 59396992
PREDICATELOCK hash | 17720437888 | 66384 | 66432 | 111420288
PREDICATELOCKTARGET hash | 17680488704 | 33616 | 33664 | 39949184
PredXactList | 17831858176 | 88 | 128 | 19703168
Proc Header | 17880795520 | 136 | 256 | 12269696
PROCLOCK hash | 17619485952 | 66384 | 66432 | 60998528
RWConflictPool | 17855898752 | 24 | 128 | 24628992
SERIALIZABLEXID hash | 17851561344 | 4944 | 4992 | 4337408
Shared Buffer Lookup Table | 17442475392 | 131920 | 131968 | 117613440
(9 rows)

tps = 48.098817 (without initial connection time)
waiting for server to shut down..... done
server stopped
waiting for server to start.... done
server started
tps for 18gb
name | off | size | allocated_size | true
---------------------------+------------+--------+---------------+-----------
LOCK hash | 19744272768 | 33616 | 33664 | 59396992
PREDICATELOCK hash | 19904621696 | 66384 | 66432 | 111420288
PREDICATELOCKTARGET hash | 19864672512 | 33616 | 33664 | 39949184
PredXactList | 20016041984 | 88 | 128 | 19703168
Proc Header | 20064979328 | 136 | 256 | 12269696
PROCLOCK hash | 19803669760 | 66384 | 66432 | 60998528
RWConflictPool | 20040082560 | 24 | 128 | 24628992
SERIALIZABLEXID hash | 20035745152 | 4944 | 4992 | 4337408
Shared Buffer Lookup Table | 19616173440 | 131920 | 131968 | 128099200
(9 rows)

tps = 46.381753 (without initial connection time)
waiting for server to shut down..... done
server stopped
```

The speed of table deletion was tested depending on the size of the buffer cache.
The speed of table deletion significantly decreases with increasing buffer cache size, even when the buffer cache is empty.
This happens after deleting an object, when the buffers containing its blocks are invalidated, or when deleting a separate file, or when truncating a file (including vacuum), or when deleting a database.

5) Reset the parameters to default values and restart the instance:

```
postgres@tantor:~$
sudo systemctl start tantor-se-server-16
sudo systemctl start postgresql
psql -p 5435 -c "alter system reset max_locks_per_transaction"
psql -p 5435 -c "alter system reset max_connections"
psql -c "alter system reset max_connections"
sudo systemctl restart tantor-se-server-16
sudo systemctl restart tantor-se-server-16
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

tantor

Practice for Chapter 6

Part 1. Free space map

1) Install pageinspect extension :

```
postgres=# create extension if not exists pageinspect;
CREATE EXTENSION
```

2) Create two tables with different column orders and insert rows:

```
postgres=#
begin;
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 varchar(1), c2 bigserial , c3 date, c4 timestamp);
create table t2 (c1 bigserial , c2 timestamp, c3 date, c4 varchar(1));
insert into t1 values('A', 1, now(), current_timestamp);
insert into t2 values(1, current_timestamp, now(), 'A');
commit;
BEGIN
CREATE TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
COMMIT
```

3) Look at the contents of the rows inserted into the tables using the pageinspect extension functions:

```
postgres=# select t_data, lp_len, t_hoff from
heap_page_items(get_raw_page('t1','main',0));
t_data | lp_len | t_hoff
-----------------------------------------------------------------+--------+--------
\x 0541 000000000000 0100000000000000 9a230000 00000000 9e7a078838cc0200 | 56 | 24
(1 row)
```

```
postgres=# select t_data, lp_len, t_hoff from
heap_page_items(get_raw_page('t2','main',0));
t_data | lp_len | t_hoff
------------------------+--------+--------
\x0100000000000000 9e7a078838cc0200 9a230000 0541 | 46 | 24
(1 row)
```

In table t1, the row takes up more space than in table t2. The values in the row fields in both tables are the same, only the order of the columns is different.

4) Look at the contents of the first four blocks of the free space map of the pg_class table . `This table` has many data blocks and rows in them:

```
postgres=#
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 1));
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 2));
SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 3));
fsm_page_contents
-------------------
0:236+
1:236+
3:236+
7:236+
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
15:236+
31:236+
63:236+
127:236+
255:236+
511:236+
1023:236+
2047:236+
4095:236+
fp_next_slot: 0 +

(1 row)
...
4095:189+
4096: 115 +
4097:2+
4098:2+
4099:2+
4100:2+
4101:154+
4102:236+
4103:12+
4104:3+
4105: 6+
4106: 10+
4107: 8+
4108:11+
4109:189+
fp_next_slot: 1 +

(1 row)
ERROR: block number 3 is out of range for relation "pg_class"
```

There is no fourth block .
For each data block, the map stores 1 byte. A byte represents the free space divided by 32 rounded down. To quickly find a free block, the map is not just a list, but has a tree structure above the list. The tree is stored on each FSM page as an array and is not ideal. There is no practical point in reading the values of numbers in FSM blocks.

The map algorithm provides for different blocks to be issued to different server processes in order to reduce contention for access to data blocks. Different server processes will insert rows (or row versions) into different blocks. The algorithm takes into account that, if possible, data blocks are filled sequentially, since this allows for prefetching of blocks .

5) Completely vacuum the `pg_class system catalog table` and look at the map of free FSM space and VM visibility :

```
postgres=# vacuum full pg_class;
VACUUM
postgres=# SELECT * from fsm_ page_contents(get_raw_page('pg_class', ' fsm ',
0));
ERROR: could not open file "base/5/535511_fsm": No such file or directory
postgres=# SELECT * from page_header (get_raw_page('pg_class', ' vm ', 0));
ERROR: could not open file "base/5/535511_vm": No such file or directory
```

Full vacuum did not create FSM and VM layers, this is normal.

6) Perform normal vacuuming:

```
postgres =# vacuum pg_class ;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
VACUUM
```

7) Look at the FSM and VM map :

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('pg_class', 'fsm', 0));
fsm_page_contents
-------------------
0:159+
1:159+
3:159+
7:159+
15:159+
31:159+
63:159+
127:159+
255:159+
511:159+
1023:159+
2047: 159 +
4095:159+
fp_next_slot: 0 +

(1 row)
postgres=# SELECT * from page_header(get_raw_page('pg_class', 'vm', 0));
lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----------+---------+-------+-------+-------+---------+----------+---------+----------
0/8BEFFDB0 | 0 | 0 | 24 | 8192 | 8192 | 8192 | 5 | 0
(1 row)
```

Normal vacuuming created layers (files) FSM and VM.

FSM blocks are not explicitly logged, as the algorithm has self-correction procedures. Increasing the size of an FSM file is not logged, and size discrepancies do not cause an error.
When damage is detected, the process that detected the inconsistencies in the tree attempts to correct the errors and rebuild part of the tree.
Regular vacuum and autovacuum update the FSM tree leaf pages and recreate the FSM and VM if the FSM files are missing.

8) Let's check if and when the FSM server process can be created.
See if table t1 has a free space map:

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('t1', 'fsm', 0));
ERROR: could not open file "base/5/535438_fsm": No such file or directory
```

There is no free space map file. As soon as the block is full and the server process wants to make a change to the free space map, it will create one.

9) Execute an anonymous block to insert 100 rows into table t1:

```
postgres=#
DO $$
DECLARE
BEGIN
FOR r IN 1..100 LOOP
insert into t1 values('A', 1, clock_timestamp(), current_timestamp);
END LOOP ;
END $$;
```

DO

10) Check if the map has been created:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('t1', 'fsm', 0));
ERROR: could not open file "base/5/535438_fsm": No such file or directory
```

The FSM map was not created by the server process.

11) Why wasn't the map created? Check how many blocks are in the table. Run request :

```
postgres=# select max(ctid) from t1;
max
---------
(0,100)
(1 row)
```

One block and it has 100 lines.

12) Insert another 100 rows and check how many blocks are in the main layer (data files):

```
postgres=#
DO $$
DECLARE
BEGIN
FOR r IN 1..100 LOOP
insert into t1 values('A', 1, clock_timestamp(), current_timestamp);
END LOOP;
END$$;

DO
postgres=# select max(ctid) from t1;
max
--------
(1.65)
(1 row )
```

Two blocks. The second block has 65 lines.

13) Check that the FSM file has been created:

```
postgres=# SELECT * from fsm_page_contents(get_raw_page('t1', 'fsm', 0));
fsm_page_contents
--------------------
fp_next_slot: 0 +

(1 row )
```

14) Check that the contents of the first lock are really empty :

```
postgres=# SELECT * from page_header(get_raw_page('t1', 'fsm', 0));
lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----------+----------+-------+-------+-------+---------+----------+---------+-----------
0/0 | 0 | 0 | 24 | 8192 | 8192 | 8192 | 5 | 0
(1 row)
```

`upper` indicates the end of a block.
Although the server process created the FSM file, it did not fill it. Why? Because to fill it, all the data blocks of the table must be read, and this is not the job of the server process and would delay the execution of the command. VM file was not created.
Only regular vacuum and autovacuum will fill the FSM and VM. Server processes make changes to the filled FSM and fix damage to a part of the FSM tree.

Oleg Ivanov

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Part 2. Changing the order of columns

1) If the virtual machine has network access, download the archive with the demo database:

```
postgres@tantor:~$ wget https://edu.postgrespro.com/demo-medium-en.zip
Saving to: 'demo-medium.zip'
'demo-medium-en.zip' saved [64544920/64544920]
```

The demo database is distributed under the PostgreSQL license.
The demo base is available in three versions : **demo - small .zip demo - medium .zip demo - big .zip**
The demo base is also located at:

```
postgres@tantor:~$ wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/example-guided-project/flights_RUSSIA_small.sql
'flights_RUSSIA_small.sql' saved [103865229/103865229]
postgres=# \i flights_RUSSIA_small.sql
```

2) Import base data :

```
postgres@tantor:~$ time zcat demo-medium.zip | psql
SET
SET
...
real 1m 3.339s
user 0m2.168s
sys 0 m 0.255 s
```

3) Let's use a script that can show the optimal order of columns. Download the script:

```
postgres @ tantor :~$ wget https :// raw . githubusercontent . com / NikolayS / postgres_dba / refs / heads / master / sql / p 1_alignment_padding . sql
'p1_alignment_padding.sql' saved [6598/6598]
postgres@tantor:~$ cat p1_alignment_padding.sql
```

4) Connect to the demo database:

```
postgres @ tantor :~$ psql - d demo
```

5) Execute the script:

```
demo=# \i p1_alignment_padding.sql
Table | Size | Wasted | Suggested Columns Reorder
------------------------+-------+---------------+------------------------------------
bookings.ticket_flights| 155 MB| ~18 MB (11.61%)| amount, fare_conditions, flight_id, ticket_no
bookings.boarding_passes| 110 MB| ~14 MB (13.13%)| boarding_no, flight_id, seat_no, ticket_no
bookings.tickets | 109 MB| ~6.4MB (5.79%) | book_ref, contact_data, passenger_id,
                                              passenger_name, ticket_no
bookings.bookings | 30 MB | |
bookings.flights | 6.7MB | |
bookings.seats | 96 kB | |
bookings.airports_data | 48 kB |~832 b (1.69%) | airport_code, airport_name, city, timezone,
coordinates
bookings.aircrafts_data| 8192 b| |
(8 rows )
```

For 4 tables, the script found a more optimal order of columns . Expected space savings according to the script calculations are ~11%.

demo database objects :

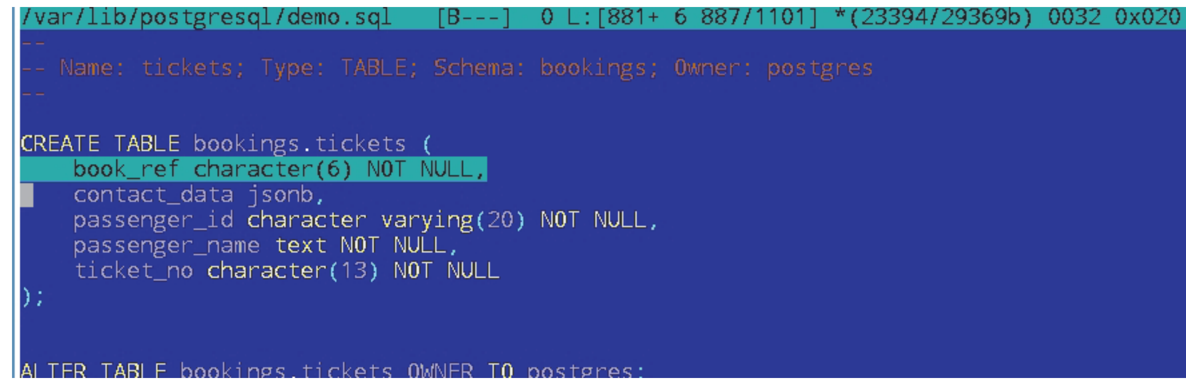
"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

postgres@tantor:~$ **pg_dump -d demo -s -f demo.sql**

The -s option specifies not to dump table rows.
Parameter -f is the name of the file where commands for creating and modifying objects are dumped (dump file)
The -d parameter specifies which database to connect to in order to dump its contents.

7) Edit the script by changing the order of columns for `the tables ticket_flights , boarding_passes , tickets :`

postgres@tantor:~$ **mcedit demo.sql**



8) Create a database called demo 2, connect to it and run the edited `demo.sql script :`

```
postgres=#
create database demo2;
\c demo2 \\
\i demo.sql
CREATE DATABASE
You are now connected to database "demo2" as user "postgres".
...
ALTER TABLE
demo 2=#
```

demo database to the demo 2 database :

```
postgres@tantor:~$ time pg_dump -d demo -a | psql -d demo2
...
real 3m38 .040s
user 0m1.981s
sys 0 m 0.259 s
```

The overload is performed in one thread and will last ~3 minutes .
During these three minutes, complete the following two practice points.

10) While the data overload command is running, execute the query:

demo2=# **select * from pg_stat_progress_copy;**

```
pid | dated | datname | release | command | type |bytes_processed |bytes_total|tup_processed | excluded
-------+--------+--------+--------+-----------+-----+----------------+----------+-------------+--------
59160 | 535928 | demo | 535978 | COPY TO | PIPE | 217467 | 0| 5654 | 0
59161 | 536055 | demo2 | 536110 | COPY FROM | PIPE | 83424852 | 0| 829071 | 0
(2 rows )
```

The query allows you to evaluate the speed of execution of COPY commands . There is no separate view for monitoring the progress of the unloading-loading, since the execution of demo . sql from the point of view of the server process is a set of separate commands.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

In addition to the pg_stat_progress_copy representation There are also a few views to monitor the progress of some typically long-running commands:

```
postgres=# \dv * progress *
List of relations
Schema | Name | Type | Owner
------------+------------------------------+------+----------
pg_catalog | pg_stat_progress_analyze | view | postgres
pg_catalog | pg_stat_progress_basebackup | view | postgres
pg_catalog | pg_stat_progress_cluster | view | postgres
pg_catalog | pg_stat_progress_copy | view | postgres
pg_catalog | pg_stat_ progress _create_index | view | postgres
pg_catalog | pg_stat_progress_vacuum | view | postgres
(6 rows)
```

11) While the data overload command is running, execute the query:

```
postgres=# select * from pg_stat_activity where query like 'COPY%' \gx
-[ RECORD 1 ]----+------------------------------------------------------------
datid | 535645
datname | demo2
pid | 57607
leader_pid |
usesysid | 10
usename | postgres
application_name | psql
client_addr |
client_hostname |
client_port | -1
backend_start | 17:50:39.986712+03
xact_start | 17:59:27.486498+03
query_start | 17:59:27.486498+03
state_change | 17:59:27.4865+03
wait_event_type | IO
wait_event | DataFileWrite
state | active
backend_xid | 406744
backend_xmin | 406744
query_id |
query | COPY bookings.boarding_passes (ticket_no, flight_id, boarding_no, seat_no) FROM stdin;
backend_type | client backend
```

12) After the data is reloaded, execute the query:

```
postgres=# select pg_size_pretty(a), pg_size_pretty(b), 100*(ab)/(a+b) "%" from (select pg_database_size('demo') a , pg_database_size('demo2') b);
pg_size_pretty | pg_size_pretty | %
----------------+----------------+----
706 MB | 815 MB | -7
(1 row)
```

The query shows the sizes of the original database, the optimized one, and the percentage of space saved. The space has increased. Why did this happen?
Because the data was loaded with the indexes created. Because of this, the data loading time increased significantly and the index sizes became larger.

13) Look at the sizes of indexes in two databases demo and demo2:

```
demo2=# \di+ bookings.*
List of relations
Schema | Name | Table | Size |
----------+---------------------------------------+----------------+--------+
bookings | aircrafts_pkey |aircrafts_data | 16 kB |
bookings | airports_data_pkey |airports_data | 16 kB |
bookings | boarding_passes_flight_id_boarding_no_key |boarding_passes | 281 MB |
bookings | boarding_passes_flight_id_seat_no_key |boarding_passes | 274 MB |
bookings | boarding_passes_pkey |boarding_passes | 404 MB |
bookings | bookings_pkey |bookings | 45 MB |
```

```
bookings | flights_flight_no_scheduled_departure_key |flights | 9552 kB |
bookings | flights_pkey |flights | 7328 kB |
bookings | seats_pkey |seats | 48 kB |
bookings | ticket_flights_pkey |ticket_flights | 423 MB |
bookings | tickets_pkey |tickets | 99 MB |
(11 rows)
```

```
demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# \di+ bookings.*
List of relations
Schema | Name | Table | Size |
----------+-------------------------------------------+----------------+---------+
bookings | aircrafts_pkey |aircrafts_data | 16 kB |
bookings | airports_data_pkey |airports_data | 16 kB |
bookings | boarding_passes_flight_id_boarding_no_key |boarding_passes | 170 MB |
bookings | boarding_passes_flight_id_seat_no_key |boarding_passes | 170 MB |
bookings | boarding_passes_pkey |boarding_passes | 307 MB |
bookings | bookings_pkey |bookings | 45 MB |
bookings | flights_flight_no_scheduled_departure_key |flights | 6648 kB |
bookings | flights_pkey |flights | 4744 kB |
bookings | seats_pkey |seats | 48 kB |
bookings | ticket_flights_pkey |ticket_flights | 325 MB |
bookings | tickets_pkey |tickets | 89 MB |
(11 rows)
```

Seven indexes increased in size. Index `bookings _ pkey` did not change size. How does it differ from other indexes? In that the data of the column by which it was created was filled with monotonically increasing values:

```
demo2=# select * from bookings.bookings limit 3;
book_ref | book_date | total_amount
----------+------------------------+--------------
00000F | 2017-07-05 03:12:00+03 | 265700.00
000012 | 2017-07-14 09:02:00+03 | 37900.00
00002D | 2017-05-20 18:45:00+03 | 114700.00
(3 rows)
```

The structure of b-tree indexes in PostgreSQL is filled optimally when inserting monotonically increasing values. This happens if the index is created on a column that is filled with an increasing sequence.

14) Rebuild indexes. This can be done in several ways.
The first way to execute the command:

```
demo =# \ c demo 2
demo2=# SELECT 'REINDEX INDEX ' || indexrelid::regclass || ';' FROM pg_index
where indexrelid::regclass::text like 'bookings.%'; \gexec
?column?
-----------------------------------------------------------------
REINDEX INDEX bookings.aircrafts_pkey;
REINDEX INDEX bookings.airports_data_pkey;
REINDEX INDEX bookings.boarding_passes_flight_id_boarding_no_key;
REINDEX INDEX bookings.boarding_passes_flight_id_seat_no_key;
REINDEX INDEX bookings.boarding_passes_pkey;
REINDEX INDEX bookings.bookings_pkey;
REINDEX INDEX bookings.flights_flight_no_scheduled_departure_key;
REINDEX INDEX bookings.flights_pkey;
REINDEX INDEX bookings.seats_pkey;
REINDEX INDEX bookings.ticket_flights_pkey;
REINDEX INDEX bookings.tickets_pkey;
(11 rows)
```

The command is convenient because you can add a predicate (WHERE condition) and filter out those indexes that need to be rebuilt.

The second and third way to execute commands:

```
demo2=# \timing on
Timing is on.
demo2=# REINDEX SCHEMA bookings;
REINDEX
Time: 27179.519 ms (00:27.180)
demo2=# REINDEX DATABASE;
REINDEX
Time: 28331.634 ms (00:28.332)
demo2=# \timing off
Timing is off.
```

It is assumed that tables are not modified in parallel sessions, so REINDEX is optimal over REINDEX INDEX CONCURRENTLY. REINDEX INDEX CONCURRENTLY commands perform two passes over the indexed data, which is slower than the single pass that REINDEX performs.

15) Compare the time of index rebuild with CONCURRENTLY option and normal rebuild.
To do this, run the commands:

```
postgres @ tantor :~$ sudo restart
postgres@tantor:~$ time psql -d demo2 -c "REINDEX INDEX CONCURRENTLY
bookings.boarding_passes_pkey;"
REINDEX

real 0m 9.278 s
user 0m0.009s
sys 0m0.000s
postgres@tantor:~$ sudo restart
postgres@tantor:~$ time psql -d demo2 -c "REINDEX INDEX
bookings.boarding_passes_pkey;"
REINDEX

real 0m 7.982 s
user 0m0.009s
sys 0m0.000s
```

Instance restart was performed to clear the buffer cache. **Index rebuild with CONCURRENTLY is 16% slower.**

16) How to prevent indexes from being created before loading data? pg_dump does not have such parameters. When editing the demo.sql script, at the end of the script there are commands:

```
ALTER TABLE ONLY ..
ADD CONSTRAINT ..
```

These commands create indexes. You can copy the script, auto-replace " ADD CONSTRAINT " with " --ADD CONSTRAINT ". At the end of the set of commands with commented parts add ";". Example:

```
postgres @ tantor :~$ cp demo .sql demo - withoutindexes .sql
postgres@tantor:~$ mcedit demo-withoutindexs.sql
<F4>
postgres@tantor:~$ cat demo-withoutindexs.sql | tail -16
ALTER TABLE ONLY bookings.ticket_flights
   -- ADD CONSTRAINT tickets_flights_ticket_no_fkey FOREIGN KEY (ticket_no) REFERENCES
bookings.tickets(ticket_no);


--
-- Name: tickets tickets_book_ref_fkey; Type: FK CONSTRAINT; Schema: bookings; Owner: postgres
--

ALTER TABLE ONLY bookings.tickets
   -- ADD CONSTRAINT tickets_book_ref_fkey FOREIGN KEY (book_ref) REFERENCES bookings.bookings(book_ref);
;

--
-- PostgreSQL database dump complete
```

--

Similarly with index creation commands, if there are any, but there are no such commands, since there are no "analytical indexes" (indexes that are not used by integrity constraints, but are used to speed up queries). Sequence of actions:
a) execute the modified `script demo - withoutindexs .sql`
b) reload the data `with the command pg_dump -d demo - s - f demo .sql` **The data overload command will be executed in 10 seconds instead of 3 minutes.** The presence of indexes significantly (12 times) slowed down the introduction of changes to tables.
c) execute the script `demo . sql` . When executing the script `demo . sql` All commands will return errors except adding integrity constraints.

17) Complete request :

```
demo2=# select pg_size_pretty(a), pg_size_pretty(b), 100*(ab)/(a+b) "%" from
(select pg_database_size('demo') a , pg_database_size('demo2') b);
pg_size_pretty | pg_size_pretty | %
-----------------+----------------+---
706 MB | 706 MB | 0
(1 row )
```

The size of the bases became the same.
However, rearranging the columns as recommended by the script did not lead to success. The size of the largest table remained the same:

```
demo 2=# select pg_table_size ( ' bookings.ticket_flights ' ) ;
 pg_table_size
---------------
162660352
(1 row)

demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# select pg_table_size('bookings.ticket_flights');
pg_table_size
---------------
162660352
(1 row )
```

18) List of functions that have size in their names:

```
demo2=# \df pg*size*
List of functions
Schema | Name | Result data type | Argument data types | Type
------------+----------------------+------------------+---------------------+------
pg_catalog | pg_column_size | integer | "any" | func
pg_catalog | pg_database_size | bigint | name | func
pg_catalog | pg_database_size | bigint | oid | func
pg_catalog | pg_indexes_size | bigint | regclass | func
pg_catalog | pg_relation_size | bigint | regclass | func
pg_catalog | pg_relation_size | bigint | regclass, text | func
pg_catalog | pg_size_bytes | bigint | text | func
pg_catalog | pg_size_pretty | text | bigint | func
pg_catalog | pg_size_pretty | text | numeric | func
pg_catalog | pg_table_size | bigint | regclass | func
pg_catalog | pg_tablespace_size | bigint | name | func
pg_catalog | pg_tablespace_size | bigint | oid | func
pg_catalog | pg_total_relation_size | bigint | regclass | func
(13 rows )
```

There is a function `pg_column_size ( value )` that shows how many bytes will be occupied when saving value . If you pass the table column name to the `function arguments, the function will return how much space the field actually occupies in the table` block, taking into account compression, padding, and displacement in TOAST . If the field is stored in compressed form, it will return how much space the field occupies in compressed form.
There is no function to measure the size of a string.

19) Using the function pg_column_size (table .*) you can see the size of the entire row together with the header. In this case, the function returns the size of the row inside the block without padding the entire row . An example of measuring the real size of the data area of table rows without padding the entire row :

```
demo2=# select pg_column_size(bookings.ticket_flights. * ) from bookings.ticket_flights limit 1;
pg_column_size
----------------
58 -> 64
(1 row)

demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# select pg_column_size(bookings.ticket_flights.*) from bookings.ticket_flights limit 1;
pg_column_size
----------------
60 -> 64
(1 row)

demo=# select pg_column_size(bookings.tickets.*) from bookings.tickets limit 1;
pg_column_size
----------------
103 -> 104
(1 row)

demo=# \c demo2
You are now connected to database "demo2" as user "postgres".
demo2=# select pg_column_size(bookings.tickets.*) from bookings.tickets limit 1;
pg_column_size
----------------
103 -> 104
(1 row)

demo2=# select pg_column_size(bookings.boarding_passes.*) from bookings.boarding_passes limit 1;
pg_column_size
----------------
49 -> 56
(1 row)

demo2=# \c demo
You are now connected to database "demo" as user "postgres".
demo=# select pg_column_size(bookings.boarding_passes.*) from bookings.boarding_passes limit 1;
pg_column_size
----------------
51 -> 56
(1 row )
```

The difference is small - just a few bytes.
This difference is compensated by the padding of the entire line: the length of the entire line (header plus data area) must be a multiple of 8 bytes.
In addition to padding of individual columns, the entire row is ALWAYS aligned to 8 bytes. The row header plus the data area is the full size of the row and is aligned to 8 bytes.
Rearranging columns for tables in the demo database had no effect.
The size of the entire string must be a multiple of 8, which means it is: .., 48, 56 , 64 ,.., 96, 104,...
The five previous SELECT statements returned the numbers: 58 - complements to 64 ; 60 - complements to 64 ; 49 and 51 complement to 56 .
If the text of a point is not clear, reread the point again. It can be difficult to understand, like any new information. Once you have a sense of understanding, you will easily remember it later. Even highly qualified specialists miss "padding of the whole line". In particular, in a complex script **p 1_ alignment _ padding . sql** "padding of the whole line" is not taken into account.

What's interesting is that if the pg_column_size(table.*) function on tables with swapped rows returned values of 64 and 65, then the space savings on each row would be 7 bytes. But if the function returns 57 and 63, then there is no difference.
Note: on 64-bit operating systems, the alignment is 8 bytes, on 32-bit systems, 4 bytes. The alignment is saved in the control file and can be viewed using the command line utility:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres@tantor:~$ pg_controldata | grep align
Maximum data alignment : 8
```

19) Let's check if the script also reacts to those tables where the column rearrangement will have an effect. Let's create two tables with different column orders and the same data. One table will take up 18% more space due to padding.

```
demo=#
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 char(1), c2 integer, c3 char(1), c4 integer, c5 char(1), c6
integer);
create table t2 (c1 integer, c2 integer, c3 integer, c4 char(1), c5 char(1), c6
char(1));
ALTER TABLE t1 SET ( fillfactor = 100);
ALTER TABLE t2 SET ( fillfactor = 100);
DO
$$
BEGIN
FOR i IN 1 .. 500000 LOOP
INSERT INTO t1 VALUES ('1', 1,'1', 1,'1', null);
INSERT INTO t2 VALUES (null, 1, 1, '1','1','1');
END LOOP;
END;
$$;
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
DO
demo=# \i p1_alignment_padding.sql
Table | Table Size | Wasted * | Suggested Columns Reorder
--------------------------+-----------+------+--------------------------------------
bookings.ticket_flights | 155 MB | ~18 MB (11.61%) | amount, fare_conditions, flight_id +
| | | ticket_no
bookings.boarding_passes | 110 MB | ~14 MB (13.13%) | boarding_no, flight_id, seat_no +
| | | ticket_no
bookings.tickets | 109 MB | ~6477 kB (5.79%) | book_ref, contact_data, passenger_id+
| | | passenger_name, ticket_no
bookings.bookings | 30 MB | |
bookings.t1 | 25 MB | |
bookings.t2 | 21 MB | |
bookings.flights | 6688 kB | |
bookings.seats | 96 kB | |
bookings.airports_data | 48 kB | ~832 bytes (1.69%) | airport_code, airport_name, city +
| | | timezone, coordinates
bookings.aircrafts_data | 8192 bytes | |
(10 rows )
```

The sizes of tables t1 and t2 are different, although the script did not recommend rearranging the columns of table t1.

20) Let's look at an example where the script gives an accurate estimate. Run commands :

```
postgres=#
drop table if exists t;
create table t(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int8, c8 int8);
DO
$$
BEGIN
FOR i IN 1 .. 100000 LOOP
INSERT INTO t VALUES (1,2,3,4,5,6,7,8);
END LOOP;
```

```
END;
$$
;
drop table if exists t1;
create table t1(c1 int8, c2 int4, c3 int4, c4 int4, c5 int8, c6 int4, c7 int4, c8 int4);
DO
$$
BEGIN
FOR i IN 1 .. 100000 LOOP
INSERT INTO t1 VALUES (1,2,3,4,5,6,7,8);
END LOOP;
END;
$$
;
select pg_table_size('t'), pg_table_size('t1'), 100*(pg_table_size('t1')-
pg_table_size('t'))/pg_table_size('t') "%";
DROP TABLE
CREATE TABLE
DO
DROP TABLE
CREATE TABLE
DO
pg_table_size | pg_table_size | %
---------------+--------------+----------
6914048 | 7684096 | 11
(1 row )
```

Space saving 11% (size increase 10% if t is specified in the denominator instead of t1 table).

```
postgres=# \i p1_alignment_padding.sql
Table | Table Size | Wasted * | Suggested Columns Reorder
----------------------+-----------+------------------+------------
pgbench_accounts | 13 MB | |
t1 | 7512 kB | ~781 kB (10.40%) | c2, c3, c4 +
                      | | | c6, c7, c8 +
| | | c1, c5
t | 6760 kB | |
```

The script correctly estimated the space savings.

21) In the theoretical part, examples were given on the slide " Aligning ".
An example was given for the second picture:

```
create table t ( a boolean, b int4);
insert into t values (true, 1);
select t_data, lp_len, t_hoff from heap_page_items(get_raw_page('t','main',0));
t_data | lp_len | t_hoff
--------------------+--------+--------
\ x0100000001000000 | 32 | 24
```

Create an example for the fourth picture on the slide:

The first field is 1-byte type " char ", the second field is bigint.
The result should be similar to what is shown on the slide. Example :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres=# drop table if exists t;
DROP TABLE
postgres =# write the command to create the table
CREATE TABLE
postgres=# insert into t values ( '1' , 1 );
INSERT 0 1
postgres=# select t_data, lp_len, t_hoff from
heap_page_items(get_raw_page('t','main',0));
t_data | lp_len | t_hoff
--------------------------+--------+--------
\ x 31 00000000000000 0100000000000000 | 40 | 24
(1 row )
```

There are 7 unused bytes , which is quite a lot. If you swap the columns, will the unused bytes disappear between the fields? They will disappear between the fields, but the same 7 unused bytes will appear between the rows in the block, since the rows are aligned to 8 bytes.

Part 3. Table Block Contents

1) Create a table:

```
postgres=#
drop table if exists t;
create table t(s text);
insert into t values ('a');
select * from page_header(get_raw_page('t', 0));
NOTICE: table "t" does not exist, skipping
DROP TABLE
CREATE TABLE
INSERT 0 1
lsn | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
9/5507B2B8 | 0 | 0 | 28 | 8144 | 8176 | 8192 | 5 | 0
(1 row)
```

lower indicates the end of the block header
special at the end of the data area, also the beginning of the special area, which is used to support 64-bit transaction identifiers
pagesize - block size, in Tantor DBMS it is always 8192.
checksum - block checksum. It is calculated when saving the block to the file. When the block is in the buffer, the checksum does not change.
prune_xid - defaults to 0. xmax of the oldest unpurged row in the block. Used How hint (it's like a hint-bit, non-WAL-logged hint field that helps determine whether pruning will be useful. It is currently unused in index pages) to the process that will search place V block to tried clear block (First check whether there's any chance there's something to prune).

Of greatest interest are **lower** and **upper** . These fields can be used to determine the size of the block header, the size of the data area, and how much free space is in the block.

2) To see a non-zero value in the checksum field, perform a full vacuum of the table:

```
postgres=# vacuum full t;
select * from page_header(get_raw_page('t', 0));
VACUUM
lsn | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
9/ BCF 15670 | -7207 | 0 | 28 | 8144 | 8176 | 8192 | 5 | 0
(1 row )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

3) See which data types are preferred in your category :

```
postgres=# select distinct typname, typalign , typstorage, typlen from pg_type
where typname not like 'pg_%' and typname not like '\_%' and typtype='b' and
typispreferred = true and typisdefined = true order by typname;
typname | typalign | typstorage | typlen
-------------+----------+------------+--------
bool | c | p | 1
float8 | d | p | 8
inet | i | m | -1
interval | d | p | 16
oid | i | p | 4
text | i | x | -1
timestamptz | d | p | 8
varbit | i | x | -1
(8 rows)
```

`typispreferred = true` means that type is preferred type V his categories (typcategory). 8 rows were returned, which means there are 8 categories of data types. If you remove this condition, the query will return 62 rows.
`typtype = b` (base) base data type. Other types are created separately: c (composite) composite type (table row image), d (domain) domain (restriction), e (enum) enumeration, p (pseudo-type) pseudo-type, r (range) range type, m (multirange) multi-range type.
typisdefined = true the type is formed. false - the type is a blank and is not ready for use.

`typalign` - alignment. Accepts values:
`c` (char), 1 byte, i.e. without alignment
`s` (short), 2 bytes
`i` (int), 4 bytes
`d` (double) 8 bytes

`typlen` - the number of bytes that store a field of this type. For variable-length types (called `varlena`), `typlen = -1` .

`typstorage` - the default column storage strategy. For fixed-width data types (`typlen >0`) the only possible strategy is " `p` ". Possible values:
`p` (plain): the field is stored in the table block without compression
`e` (external): the field value is moved to TOAST
`m` (main, in the main layer): the field is compressed and stored in the table block
`x` (extended): The field can be uncompressed/compressed, stored in a table block, or the field value can be TOASTed.

Part 4. Aligning fields in table rows

1) Create tables and view table column characteristics:

```
postgres=#
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 varchar(1), c2 bigserial , c3 date, c4 timestamp);
create table t2 (c1 bigserial , c2 timestamp, c3 date, c4 varchar(1));
SELECT a.attname, t.typname, t.typalign, t.typlen
FROM pg_class c
JOIN pg_attribute a ON (a.attrelid = c.oid)
JOIN pg_type t ON (t.oid = a.attypid)
WHERE c.relname = 't1' AND a.attnum >= 0;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
SELECT a.attname, t.typname, t.typalign, t.typlen
FROM pg_class c
JOIN pg_attribute a ON (a.attrelid = c.oid)
JOIN pg_type t ON (t.oid = a.attypid)
WHERE c.relname = 't2' AND a.attnum >= 0;
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
attname | typname | typalign | typlen
---------+-----------+----------+--------
c1 | varchar | i | -1
c2 | int8 | d | 8
c3 | date | i | 4
c4 | timestamp | d | 8
(4 rows)

attname | typname | typalign | typlen
---------+-----------+----------+--------
c1 | int8 | d | 8
c2 | timestamp | d | 8
c3 | date | i | 4
c4 | varchar | i | -1
(4 rows )
```

The last column in table t1 is a fixed-width column of 8 bytes. This means that all previous fixed-width columns will be aligned to 8 bytes. Variable-width columns varlena (text, bytea, numeric, and many others) have i alignment (less often d). Fields of such types will be aligned and force previous columns to be aligned **if such fields store more than 126 bytes.** This phrase is difficult to understand, it can be detailed:
In table t1, all four columns are aligned to 8 bytes, including column c1, which takes up at least 8 bytes. But in table t2, column c4 is not aligned and does not force the previous columns to be aligned if the size of the c4 field is less than 126 bytes.

2) Look at the list of variable width types varlena :

```
postgres=# select distinct typname, typalign, typstorage, typlen from pg_type
where typname not like 'pg_%' and typname not like '\_%' and typtype='b' and
typis defined = true and typlen= -1 order by typname;
typname | typalign | typstorage | typlen
---------------+----------+------------+--------
bit | i | x | -1
bpchar | i | x | -1
 byte | i | x | -1
cidr | i | m | -1
gtsvector | i | p | -1
 inet | i | m | -1
int2vector | i | p | -1
json | i | x | -1
 jsonb | i | x | -1
jsonpath | i | x | -1
 numeric | i | m | -1
oidvector | i | p | -1
path | d | x | -1
 polygon | d | x | -1
refcursor | i | x | -1
 text | i | x | -1
 tsquery | i | p | -1
 tsvector | i | x | -1
txid_snapshot | d | x | -1
varbit | i | x | -1
 varchar | i | x | -1
 xml | i | x | -1
(22 rows )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Alignment speeds up data processing significantly, so it is used despite the increase in storage space. The same is true for memory structures - alignment is used whenever possible.

one row into tables t 1 and t 2:

```
postgres=#
insert into t1 values('A', 1, now(), current_timestamp);
insert into t2 values(1, current_timestamp, now(), 'A');
INSERT 0 1
INSERT 0 1
```

4) Look at the size of the line the function shows `pg _ column _ size (table.*)` :

```
postgres=#
select pg_column_size(t1.*) from t1 limit 1;
select pg_column_size(t2.*) from t2 limit 1;
row_size
----------
56
(1 row)

row_size
----------
       46
(1 row)
```

The difference in the size of the lines according to the function is 56-46=10 bytes. The real difference in the number of bytes for storing the line will be different. What? You can go back to point 18 of the previous part of the practice (where it is written that the point is difficult to understand) and calculate the real difference.
Answer: 46 will take 6*8=48 bytes, 56 will take 7*8=56 bytes. The real difference in the string size after aligning the entire string: 56-48=8 bytes.

5) What is in these 46 and 56 bytes? Complete requests :

```
postgres=# SELECT lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t1','main',0))\gx
-[ RECORD 1 ]---------------------------------------------
lp_off | 8120
lp_len | 56
t_hoff | 24
t_data | \x 0541000000000000 0100000000000000 9b23000000000000 015fcf714ccc0200
```

Size data lines (t_data) 32 bytes .
Row header size: 56-32=24 bytes.
All columns are 8 bytes and will be 8-byte aligned.

```
postgres=# SELECT lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t2','main',0))\gx
-[ RECORD 1 ]--------------------------------------------
lp_off | 8128
lp_len | 46
t_hoff | 24
t_data | \x 0100000000000000 a27fcf714ccc0200 9b230000 0541
```

Size data lines (t_data) 22 bytes .
Row header size: 46-22=24 bytes.
The last varchar column is not aligned and occupies 2 bytes. If the varchar field length exceeds 126 bytes, the field will be aligned to 4 bytes.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Fields c1 and c2 occupy 8 bytes each because that is their size.

lp_len does not show the total size of the string that the string occupies in the block. lp_len does not include the alignment of the entire string. This means that if lp_len is not divisible by 8, then 1 to 7 empty bytes will be added to the end of the string so that the size of the space occupied by the string becomes a multiple of 8 bytes. lp_len is stored in the block header.
In the second query, lp_len = 46 (not divisible by 8). The string takes up 48 bytes in the block (divisible by 8).
Why doesn't lp_len store the full string size, since it would be easier to calculate the actual string size? Because lp_len is used by instance processes to determine which byte the last field in a string ends on. The **pageinspect** extension simply outputs what is stored in **the** block.

In Tantor DBMS the minimum size of the row header is 24 bytes, as in PostgreSQL . This size can be increased by 8 bytes at once due to the size of the empty value map. The empty value map (NULL) is stored in the row header. 1 bit is used for each column.

, the row header stores the service field t_maclabel (as well as the bits t_infomaskpgac , t_hasmac) and the minimum size of each row header is 32 bytes. The heap_page_items(...) function shows the t_maclabel field .

Part 5. Aligning Rows in Table Blocks

1) Recreate tables :

```
postgres=#
BEGIN;
drop table if exists t1;
drop table if exists t2;
create table t1 (c1 serial, c2 timestamp);
create table t2(c2 timestamp, c1 serial);
insert into t1 (c2) values(current_timestamp);
insert into t1 (c2) values(current_timestamp);
insert into t2 (c2) values(current_timestamp);
insert into t2 (c2) values(current_timestamp);
COMMIT;
BEGIN
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
COMMIT
```

Two rows are inserted into the tables so that the actual size of the rows can be calculated with the entire row aligned.

2) Run a query on table t1:

```
postgres=# SELECT lp, t_ctid, lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t1','main',0));
lp | t_ctid | lp_off | lp_len | t_hoff | t_data
----+--------+--------+--------+--------+-----------------------------------
1 | (0,1) | 8136 | 40 | 24 | \x01000000000000008cb97ce24ecc0200
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
2 | (0,2) | 8096 | 40 | 24 | \x0200000000000000008cb97ce24ecc0200
(2 rows)
```

`lp` - (line pointer) number slot V title block (size 4 byte slots), in which is stored meaning `lp_off` .
`t_ctid` - string address. Consists of two numbers: the block sequence number in the main layer, the slot number (`lp`) in the block header.
`lp_off` - address of the beginning of the row header (offset in bytes from the beginning of the block).
`lp_len - the length of the string in bytes ,` excluding alignment of the entire string.
`t_hoff` - the size of the row header in bytes. The row header is always aligned to 8 bytes (`t _ hoff` multiple of 8). In `the example t_hoff` has a minimum size of 24 bytes. In Astralinux PostgreSQL the minimum size is 32 bytes. The row header size can be increased by the map of empty values in the row fields. The map of empty values (NULL) is stored in the row header. For each column 1 bit is used.

Size of the second line: `8136-8096=40` bytes.
The size of the first line is exactly the same. How to calculate it? The address of the special area is not visible in the query result. For Tanator DBMS SE is 16 bytes. Block size 8Kb = `8192` bytes. Formula: `8192` (block size) – `16` (size of special area at the end of block) – `8136` (`lp_off of` first line) = `40` bytes.

3) Run a query on table t2:

```
postgres=# SELECT lp, t_ctid, lp_off,lp_len,t_hoff,t_data FROM
heap_page_items(get_raw_page('t2','main',0));
lp | t_ctid | lp_off | lp_len | t_hoff | t_data
----+--------+--------+--------+--------+----------------------------
1 | (0,1) | 8136 | 36 | 24 | \x8cb97ce24ecc020001000000
2 | (0,2) | 8096 | 36 | 24 | \x8cb97ce24ecc020002000000
(2 rows)
```

Size second lines : `8136-8096=40` byte . The size occupied by the row is the same as that of the table t1. We found out that in terms of storage space there is no difference between the tables.

4) Let's make sure there really is no difference. Insert 600 rows into each table and see the maximum ctid:

```
postgres=#
DO
$$
BEGIN
FOR i IN 1 .. 600 LOOP
insert into t1 (c2) values(current_timestamp);
insert into t2 (c2) values(current_timestamp);
END LOOP;
END;
$$
;
DO
postgres=# select max(ctid) from t1;
max
--------
(3.47)
(1 row)

postgres=# select max(ctid) from t2;
max
--------
(3.47)
(1 row )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

In both tables the maximum ctid is the same, which means that the first two blocks fit the same number of rows.

5) How many rows fit into the first block of both tables? Complete command :

```
postgres=# SELECT count(lp) FROM heap_page_items(get_raw_page('t1','main',0));
count
-------
185
(1 row)
postgres=# SELECT count(lp) FROM heap_page_items(get_raw_page('t1','main',0));
count
-------
185
(1 row )
```

The first block of each table contained 185 rows.

6) How much space does the block header take up?

```
postgres=# select * from page_header(get_raw_page('t2','main',0));
lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
A/3D28EAE0 | 0 | 0 | 764 | 776 | 8176 | 8192 | 5 | 0
(1 row)

postgres=# select * from page_header(get_raw_page('t1','main',0));
lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
A/3D28EA98 | 0 | 0 | 764 | 776 | 8176 | 8192 | 5 | 0
(1 row)
```

In both tables, the block header takes up 764 bytes, which is 9.3% of the block size.
Between the header and the data area, 776-764=12 bytes are left unused, which is not much. The first block (the second block too) of the tables is completely filled.

In this part of the practice, you learned functions that can be used to see how rows are actually arranged in blocks. By checking a table block that takes up a lot of space, you may be able to optimize its storage. Not only by rearranging rows, but, for example, by changing the dimensions, types, or number of fields.
This part of the chapter is useful for those who design or reorganize data storage schemes in PostgreSQL. This is a small part of what is worth knowing when designing schemes. Later in the course we will consider storing null values, field displacement in TOAST, choosing data types, storing records in indexes.

Part 6. Storing empty (NULL) values in table rows

If at least one field in a row is empty (has a NULL value), then space is allocated in the header of this row for a bitmap. In a bitmap, one bit means one column. If there are a thousand columns, then there will be a thousand bits in the bitmap. The header size will increase by 1000 bits and will then be aligned to 8 bytes.
The minimum header size is 24 bytes. I would like to determine how many columns can be in a table so that the header size does not increase to 32 bytes when an empty value appears in any column.

1) Create tables with the number of columns 8,9,24,25 and insert two rows into each table. In the first row, all fields are non-empty, in the second row, at least one field is empty:

```
postgres=#
drop table if exists t1;
drop table if exists t2;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
drop table if exists t3;
drop table if exists t4;
create table t1(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4);
create table t2(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4,c9 int4);
create table t3(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4, c9 int4, c10 int4, c11 int4, c12 int4, c13 int4, c14 int4, c15 int4,
c16 int4, c17 int4, c18 int4, c19 int4, c20 int4, c21 int4, c22 int4, c23 int4,
c24 int4);
create table t4(c1 int4, c2 int4, c3 int4, c4 int4, c5 int4, c6 int4, c7 int4,
c8 int4, c9 int4, c10 int4, c11 int4, c12 int4, c13 int4, c14 int4, c15 int4,
c16 int4, c17 int4, c18 int4, c19 int4, c20 int4, c21 int4, c22 int4, c23 int4,
c24 int4, c25 int4);
INSERT INTO t1 VALUES (1,2,3,4,5,6,7,8);
INSERT INTO t1 VALUES (1,NULL,3,4,5,6,7,8);
INSERT INTO t2 VALUES (1,2,3,4,5,6,7,8,9);
INSERT INTO t2 VALUES (1,NULL,3,4,5,6,7,8,9);
INSERT INTO t3 VALUES
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24);
INSERT INTO t3 VALUES
(1,NULL,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24);
INSERT INTO t4 VALUES
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25);
INSERT INTO t4 VALUES
(1,NULL,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25);

DROP TABLE
DROP TABLE
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

2) Run queries to determine the size of the header of each row in all four tables:

```
postgres=#
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t 1 ','main',0));
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t 2 ','main',0));
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t 3 ','main',0));
select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t 4 ','main',0));

lp | lp_off | lp_len | t_ctid | t_hoff | t_bits
----+--------+--------+--------+--------+----------
1 | 8120 | 56 | (0.1) | 24 |
2 | 8064 | 52 | (0,2) | 24 | 10111111
(2 rows)
```

```
lp | lp_off | lp_len | t_ctid | t_hoff | t_bits
----+--------+--------+--------+--------+-----------------
1 | 8112 | 60 | (0.1) | 24 |
2 | 8048 | 64 | (0.2) | 32 | 10111111 10000000
(2 rows)

lp | lp_off | lp_len | t_ctid | t_hoff | t_bits
----+--------+--------+--------+--------+----------
1 | 8056 | 120 | (0.1) | 24 |
2 | 7928 | 124 | (0,2) | 32 | 101111111111111111111111
(2 rows)

lp | lp_off | lp_len | t_ctid | t_hoff | t_bits
----+--------+--------+--------+--------+---------------------------------
1 | 8048 | 124 | (0.1) | 24 |
2 | 7920 | 128 | (0,2) | 32 | 10111111 11111111 11111111 10000000
(2 rows )
```

If the table has up to 8 columns inclusive , the row header size is 24 bytes.
If the table has 9 or more columns, then the size of the row header, if at least one field contains an empty value (NULL), increases sharply in size and becomes 32 bytes .

The bitmap is aligned to one byte. This means that if the table has up to 9 columns, the bitmap takes up 1 byte (8 bits). If the table has 9-16 columns, the bitmap takes up 2 bytes (16 bits). If 17-24, then 3 bytes. Starting with 25 columns - 4 bytes , and so on.
40 bytes due to the empty value bitmap ? 8*8 (64 columns is 8 bytes in the map)+8 (an 8-column bitmap fits in a 24-byte header)+1 (an extra column that increases the row header size by 8 bytes)= 73 columns .

Don't overestimate the increase in row header size. The point is that storing NULL does not take up a single byte in the data area, i.e. it is highly efficient. For example, if you store zero instead of NULL in a field **and this field is aligned to 8 bytes, then using NULL instead of any other value will save 8 bytes** . This is evident from the **lp_len** column :
52 is less than 56 in table t1. In other tables, one more column needs to be made empty so that the row size is reduced by 4 bytes:

```
postgres=# INSERT INTO t3 VALUES
(1,NULL,3,NULL,5,NULL,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24);
INSERT 0 1
postgres=# select lp, lp_off, lp_len, t_ctid, t_hoff,t_bits from
heap_page_items(get_raw_page('t3','main',0));
lp | lp_off | lp_len | t_ctid | t_hoff | t_bits
----+--------+--------+--------+--------+----------
1 | 8056 | 120 | (0.1) | 24 |
2 | 7928 | 124 | (0,2) | 32 | 101111111111111111111111
3 | 7808 | 116 | (0.3) | 32 | 101010111111111111111111
(3 rows)
```

A string with three empty fields has a size of 116 bytes , which is less than the `120` bytes of `a` string with no empty values. `t_hoff` is included in `lp_len` . The actual size of the rows (taking into account the alignment of the entire row) is the same: 120 bytes. This is easy to check by the lp_off column **: the first row** takes 8192-8056-16=120 bytes; the third row takes 7928-7808=120 bytes.

Using NULL is inconvenient for writing queries, but in PostgreSQL NULL saves storage space.
PostgreSQL btree indexes index NULL.

Part 7. Number of rows in a table block

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Let's see how many rows can be stored in a block. The idea of how many rows can be and usually is useful to imagine how the storage of rows in tables is organized.

1) Depending on the line size, when the block is completely filled, the number of lines in the Tantor SE block will be:

```
postgres=# select string_agg(a::text,',') rows from (SELECT distinct
trunc(2038/(2*generate_series(0, 1015)+7)) a order by a desc) a;
rows
-----------------------------------------------------------------------291,226,185, 156 , 135 , 119
,107,97,88,81,75,70,65,61,58,55,52,49,47,45,43,41,39,38,37,35,34,33,32,31,30,
29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1
(1 row )
```

In builds with a 32-bit transaction counter, the special area at the end of a 16-byte block is missing. This will only affect three numbers: instead of `156` , `135` , `119` , such builds will fit `157` , `136` , `120` lines.

2) Check how many rows will fit in a block for a table with one column of type **`serial`** (**`int 4 , integer`**):

```
postgres=#
drop table if exists t;
create table t(c serial );
insert into t select * from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select count(*) from heap_page_items(get_raw_page('t','main',0)) where
t_ctid::text like '(0,%';
select * from page_header(get_raw_page('t','main',0));
DROP TABLE
CREATE TABLE
INSERT 0 600
count
-------
   226
(1 row)
lp | lp_off | lp_len | t_ctid | t_hoff | t_data
-----------+--------+--------+---------+--------+------------
226 | 944 | 28 | (0.226) | 24 | \xe2000000
225 | 976 | 28 | (0.225) | 24 | \xe1000000
(2 rows)

count
-------
   226
(1 row)

lsn | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
A/55E5F568 | 0 | 0 | 928 | 944 | 8176 | 8192 | 5 | 0
(1 row)
```

The block of such a table will fit 226 rows. By replacing the table definition, you can check how many rows will fit in the block.

How did you get 226 bytes? The formula for calculating the maximum number of rows in a Tantor SE block is:
8192 (block size) = 24 (block header) + 4*x + 24 (row header)*x + 8 (row length is a multiple of 8)*x + 16 (special area).

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

8152 = 36* x + 16
x = 226

3) In Astralinux PostgreSQL The block fits 185 lines, not 226 lines . Check this by connecting to port 5435 :

```
postgres @ tantor :~$ psql - p 5435
psql (16.2, server 15.6 ( Debian 15.6- astra . se 2))
Type "help" for help.

postgres=#
drop table if exists t;
create table t(c serial );
insert into t select * from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select * from page_header(get_raw_page('t','main',0));
DROP TABLE
CREATE TABLE
INSERT 0 600
count
-------
   185
(1 row)

lp | lp_off | lp_len | t_ctid | t_hoff | t_data
-----------+--------+--------+---------+--------+------------
185 | 792 | 36 | (0.185) | 32 | \xb9000000
184 | 832 | 36 | (0.184) | 32 | \xb8000000
(2 rows)

lsn | checksum | flags | lower | upper | special | pagesize | version |prune_xid
-----------+----------+-------+-------+-------+---------+----------+---------+---------
0/8D9C5A8 | 0 | 0 | 764 | 792 | 8192 | 8192 | 4 | 0
(1 row)
```

4) Replace serial with `big serial` (`in t 8`) and check how many lines fit in the block:

```
postgres=#
drop table if exists t;
create table t(c big serial );
insert into t select * from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select * from page_header(get_raw_page('t','main',0));
DROP TABLE
CREATE TABLE
INSERT 0 600
count
-------
   226
(1 row)

lp | lp_off | lp_len | t_ctid | t_hoff | t_data
----------+--------+--------+---------+--------+-----
226 | 944 | 32 | (0.226) | 24 | \xe200000000000000
225 | 976 | 32 | (0.225) | 24 | \xe100000000000000
(2 rows)

lsn | checksum | flags | lower | upper | special | pagesize | version |prune_xid
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
------------+----------+-------+-------+-------+---------+----------+---------+---------
A/55E20128 | 0 | 0 | 928 | 944 | 8176 | 8192 | 5 | 0
(1 row)
```

Number lines Also `226` . Why? Because the table row with the `serial column` aligned to 8 bytes. This does not mean that serial is not different from bigserial. You can add columns to tables and in some cases there will be a difference.
However, what is important is that b-tree indexes on `int4 ( integer , serial )` and `int8 ( big serial ) columns` always have the same size .

5) Number of rows with maximum data size per row:

```
postgres=# SELECT trunc(2038/(2*generate_series+7)) rows , max(generate_series*8) size FROM generate_series(0, 1015) group by rows order by 1 desc;
rows | size
------+------
291 | 0
226 | 8
185 | 16
156 | 24
135 | 32
119 | 40
107 | 48
97 | 56
88 | 64
81 | 72
75 | 80
70 | 88
65 | 96
61 | 104
58 | 112
55 | 120
52 | 128
49 | 136
47 | 144
45 | 152
43 | 160
41 | 168
39 | 176
38 | 184
37 | 192
35 | 200
34 | 208
33 | 216
32 | 224
31 | 232
30 | 240
29 | 248
28 | 256
27 | 272
26 | 280
25 | 296
24 | 304
23 | 320
22 | 336
21 | 360
20 | 376
19 | 400
18 | 424
17 | 448
16 | 480
15 | 512
14 | 552
13 | 592
12 | 648
11 | 712
10 | 784
9 | 872
8 | 984
7 | 1136
6 | 1328
5 | 1600
4 | 2008
3 | 2688
2 | 4048
1 | 8120
(60 rows)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

6) Let's check value 185, which follows for 266:

```
postgres=#
drop table if exists t;
create table t( c bigserial , c1 text default 'a' );
insert into t select *, 'a' from generate_series(1, 600);
select count(*) from t where (ctid::text::point)[0] = 0;
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) where t_ctid::text like '(0,%' order
by lp_off limit 2;
select count(*) from heap_page_items(get_raw_page('t','main',0)) where
t_ctid::text like '(0,%';
select * from page_header(get_raw_page('t','main',0));
DROP TABLE
CREATE TABLE
INSERT 0 600
count
-------
   185
(1 row)

lp | lp_off | lp_len | t_ctid | t_hoff | t_data
----------+--------+--------+---------+--------+-----------------------
185 | 776 | 34 | (0.185) | 24 | \xb90000000000000000561
184 | 816 | 34 | (0.184) | 24 | \xb80000000000000000561
(2 rows)

count
-------
   185
(1 row)

lsn | checksum | flags | lower | upper | special | pagesize | version |prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+---------
A/55E8F8B0 | 0 | 0 | 764 | 776 | 8176 | 8192 | 5 | 0
(1 row)
```

For two bigserial columns and a variable-width column storing one character and aligned to 4 bytes, the maximum number of rows in a block is 185.
Formula:
8192 = 24 (block header) + 4*x + 24 (row header)*x + 16 (row length is a multiple of 8)*x + 16 (special area)
8152 = 44*x + 12
x = 185

7) The most optimal full line size is a multiple of 64 bytes (cache line size). Number of lines and line data area size for Tantor SE:

```
postgres=# SELECT distinct trunc(2038/(16*generate_series+4)) rows,
max(generate_series*64-24) size FROM generate_series(1, 64) group by rows order
by 1 desc;
 rows | size
------+------
101 | 40
56 | 104
39 | 168
29 | 232
24 | 296
20 | 360
17 | 424
15 | 488
13 | 552
12 | 616
11 | 680
10 | 744
9 | 808
```

```
8 | 936
7 | 1064
6 | 1256
5 | 1576
4 | 1960
3 | 2664
2 | 4008
1 | 4072
(21 rows)
```

With a data area size equal to the size column, there are no losses due to padding and the rows are aligned to the cache line, i.e. they are processed with minimal overhead.

Practice for Chapter 7

Part 1. Access methods

1) Look at the available data access methods :

```
postgres=# \d A +
List of access methods
Name | Type | Handler | Description
--------+-------+---------------------+----------------------------------------
brin | Index | brinhandler | block range index (BRIN) access method
btree | Index | bthandler | b-tree index access method
gin | Index | ginhandler | GIN index access method
gist | Index | gisthandler | GiST index access method
hash | Index | hashhandler | hash index access method
heap | Table | heap_tableam_handler | heap table access method
spgist | Index | spghandler | SP-GiST index access method
(7 rows )
```

There are two types of access methods (ways): table and index.

) Install pg_columnar extensions and bloom :

```
postgres=#
create extension pg_columnar;
create extension bloom;
CREATE EXTENSION
CREATE EXTENSION
```

These extensions add access methods. You can add both table and index access methods. Table access methods define how data is stored in tables.

3) See what access methods have been added:

```
postgres=# \dA+
List of access methods
Name | Type | Handler | Description
----------+-------+---------------------+------------------------------
bloom | Index | blhandler | bloom index access method
brin | Index | brinhandler | block range index (BRIN) access method
btree | Index | bthandler | b-tree index access method
columnar | Table | columnar.columnar_handler |
gin | Index | ginhandler | GIN index access method
gist | Index | gisthandler | GiST index access method
hash | Index | hashhandler | hash index access method
heap | Table | heap_tableam_handler | heap table access method
spgist | Index | spghandler | SP-GiST index access method
(9 rows)

postgres=# select * from pg_am;
oid | amname | amhandler | amtype
--------+----------+------------+--------
2 | heap | heap_tableam_handler | t
403 | btree | bthandler | i
405 | hash | hashhandler | i
783 | gist | gisthandler | i
2742 | gin | ginhandler | i
4000 | spgist | spghandler | i
3580 | brin | brinhandler | i
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
 544775 | columnar | columnar.columnar_handler | t
 544924 | bloom | blhandler | i
(9 rows )
```

4) When creating an index, the index type and operator class are specified for each index column. Example :

```
postgres=# drop table if exists t;
NOTICE: table "t" does not exist, skipping
DROP TABLE
postgres=# create table t(id int8, s text);
CREATE TABLE
postgres=# create index t_idx on t using btree (id int8_ops, s text_pattern_ops);
CREATE INDEX
```

If you do not specify the index type (index access method), `btree is used` .
If you do not specify an operator class, the default operator class set for the data type on which the index is being created is used.

5) See what operator classes are available for the btree method. While typing the command, you can press the <TAB> key twice and a list of words that can be used to continue the command will be displayed:

```
postgres=# \dA <TAB><TAB>
\dA \dA c \dAf \dAo \dAp
postgres=# \dA c + btree int <TAB><TAB>
int2 int2vector int4 int4multirange int4range int8 int8multirange int8range
integer internal
postgres=# \dAc+ btree int8
List of operator classes
AM | Input type | Storage type | Operator class | Default? | Operator family | Owner
-------+------------+--------------+----------------+----------+-----------------+----------
btree | bigint | | int8_ops | yes | integer_ops | postgres
(1 row)
postgres=# \dAc+ btree bigint
List of operator classes
AM | Input type | Storage type | Operator class | Default? | Operator family | Owner
-------+------------+--------------+----------------+----------+-----------------+----------
btree | bigint | | int8_ops | yes | integer_ops | postgres
(1 row)
```

For `the int8 (bigint) data type` , there is one operator class `int8_ops` , which is part of `the integer_ops family` .
For int4 (integer) there is one operator class `int4_ops` , which is part of the same family.
A family depends on the presence of an access method and is associated with it. Access methods do not depend on the presence of a family. Access methods do not use families, but operator classes.

Instead of psql commands, you can use queries to system catalog tables. For example, for the **`\dAc+ * command The bigint`** equivalent would be the query:

```
postgres=# SELECT am.amname "AM", format_type(c.opcintype, NULL) "type" ,
 c.opcname "op_class", c.opcdefault "d", of.opfname "op_family"
FROM pg_opclass c JOIN pg_am am on am.oid = c.opcmethod JOIN pg_opfamily of ON
of.oid = c.opcfamily where format_type(c.opcintype, NULL) = ' bigint '
ORDER BY 1, 2, 4;
AM | type | op_class | d | op_family
-------+---------+--------+---+-------------------------
brin | bigint | int8_minmax_multi_ops | f | integer_minmax_multi_ops
brin | bigint | int8_bloom_ops | f | integer_bloom_ops
brin | bigint | int8_minmax_ops | t | integer_minmax_ops
 btree | bigint | int8_ops | t | integer_ops
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
hash | bigint | int8_ops | t | integer_ops
(5 rows)
```

6) Look at the family of operators available for integer types:

```
postgres=# \dAf btree int4
List of operator families
AM | Operator family | Applicable types
--------+-----------------+-------------------------
btree | integer_ops | smallint, integer, bigint
(1 row)

postgres=# \dAf btree int8
List of operator families
AM | Operator family | Applicable types
--------+-----------------+-------------------------
btree | integer_ops | smallint, integer, bigint
(1 row)
```

For integer types there is the same family: integer_ops
Operator classes are grouped into a family so that execution plans can be generated with expressions of different (but compatible, convertible) types without using explicit type casts.

7) Look at (no need to execute commands) the following examples, which show that using data types not included in the family is difficult and requires explicit type casting:

Let's select the values of the backend_xmin column :

```
postgres=# select backend_xmin from pg_stat_activity where backend_xmin is not null limit 1;
backend_xmin
--------------
422122
(1 row )
```

The transaction counter value looks like an integer.
Integers can be compared and sorted. This is meaning sort it is forbidden :

```
postgres=# select backend_xmin from pg_stat_activity where backend_xmin is not null order by 1 limit 1;
ERROR: could not identify an ordering operator for type xid
LINE 1: ..._activity where backend_xmin is not null order by 1 limit 1;
^
HINT: Use an explicit ordering operator or modify the query.
```

The max function does not work with this type:

```
postgres=# select max(backend_xmin) from pg_stat_activity where backend_xmin is not null order by 1 limit 1;
ERROR: function max(xid) does not exist
LINE 1: select max(backend_xmin) from pg_stat_activity where backend...
^
HINT: No function matches the given name and argument types. You might need to add explicit type casts.
```

xid data type to an integer , since there is no type casting (type casts are created by the CREATE CAST command), nor are there any casting functions:

```
postgres=# select backend_xmin::int8 from pg_stat_activity where backend_xmin is not null order by 1 limit 1;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
ERROR: cannot cast type xid to bigint
LINE 1: select backend_xmin::int8 from pg_stat_activity where backen...
^

postgres=# select backend_xmin::text from pg_stat_activity where backend_xmin is
not null order by 1 limit 1;
backend_xmin
--------------
422122
(1 row )
```

Sorting works with the text type .

8) After the previous point, you may have forgotten what operator classes exist for integers and what family they belong to. You can reread points 5 and 6 of this part of the practice. You will remember that operator families are needed so that queries are executed without explicit casting of compatible types. Let's look at the functions used to support the btree access method:

```
postgres=# \dAp+ btree integer_ops
List of support functions of operator families
AM | Operator family | Registered left type | Registered right type | Number | Function
-------+-----------------+---------------------+----------------------+--------+--------------------------
btree | integer_ops | bigint | bigint | 1 | btint8cmp(bigint,bigint)
btree | integer_ops | bigint | bigint | 2 | btint8sortsupport(internal)
btree | integer_ops | bigint | bigint | 3 | in_range(bigint,bigint,
bigint,boolean,boolean)
btree | integer_ops | bigint | bigint | 4 | btequalimage(oid)
btree | integer_ops | integer | integer | 1 | btint4cmp(integer,integer)
btree | integer_ops | integer | integer | 2 | btint4sortsupport(internal)
btree | integer_ops | integer | integer | 3 | in_range(integer,integer,
integer, boolean, boolean)
btree | integer_ops | integer | integer | 4 | btequalimage(oid)
btree | integer_ops | smallint | smallint | 1 | btint2cmp(smallint,smallint)
btree | integer_ops | smallint | smallint | 2 | btint2sortsupport(internal)
btree | integer_ops | smallint | smallint | 3 | in_range(smallint,smallint,
smallint,boolean,boolean)
btree | integer_ops | smallint | smallint | 4 | btequalimage(oid)
btree | integer_ops | bigint | integer | 1 | btint84cmp(bigint,integer)
btree | integer_ops | bigint | smallint | 1 | btint82cmp(bigint,smallint)
btree | integer_ops | integer | bigint | 1 | btint48cmp(integer,bigint)
btree | integer_ops | integer | bigint | 3 | in_range(integer,integer,
bigint,boolean,boolean)
btree | integer_ops | integer | smallint | 1 | btint42cmp(integer,smallint)
btree | integer_ops | integer | smallint | 3 | in_range(integer,integer,
smallint,boolean,boolean)
btree | integer_ops | smallint | bigint | 1 | btint28cmp(smallint,bigint)
btree | integer_ops | smallint | bigint | 3 | in_range(smallint,smallint,
bigint,boolean,boolean)
btree | integer_ops | smallint | integer | 1 | btint24cmp(smallint,integer)
btree | integer_ops | smallint | integer | 3 | in_range(smallint,smallint,
integer, boolean, boolean)
(22 rows)
```

Functions a lot , so How There is options functions that take different types data : `integer, bigint, smallint` .

For indexing by the btree method , it is enough that the data type is comparable. For this purpose, the operator class had a supporting ("reference") function **Number =1** , which could compare two values of the data type for which the operator class was created. There is a function with such a number and it is called **btint 8 cmp (..)** . The result of the function: negative, positive value or zero if the values are equal.
For efficient sorting (ORDER BY) it is desirable that the operator class has a second (**Number =2**) function for quick sorting of values. A function with such a number exists and is called **btint 8 sortsupport (..)**
To enable the planner to use the index in the "RANGE" expressions of window functions, a third (**Number =3**) function is needed.
To support deduplication, a fourth (**Number =4**) function is needed.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

9) The operator class binds operators that will play roles (strategies) in the methods that the index logic uses to order (compare, sort, measure distances, associations, etc.) data. See which operators are bound to the "strategies" of the access methods:

```
postgres =# \ dAo btree integer_ops
                List of operators of operator families
AM | Operator family | Operator | Strategy | Purpose
-------+--------+---------------------+----------+---------
btree | integer_ops | <(bigint,bigint) | 1 | search
btree | integer_ops | <=(bigint,bigint) | 2 | search
btree | integer_ops | =(bigint,bigint) | 3 | search
btree | integer_ops | >=(bigint,bigint) | 4 | search
btree | integer_ops | >(bigint,bigint) | 5 | search
btree | integer_ops | <(integer,integer) | 1 | search
btree | integer_ops | <=(integer,integer) | 2 | search
btree | integer_ops | =(integer,integer) | 3 | search
btree | integer_ops | >=(integer,integer) | 4 | search
btree | integer_ops | >(integer,integer) | 5 | search
btree | integer_ops | <(smallint,smallint) | 1 | search
btree | integer_ops | <=(smallint,smallint) | 2 | search
btree | integer_ops | =(smallint,smallint) | 3 | search
btree | integer_ops | >=(smallint,smallint) | 4 | search
btree | integer_ops | >(smallint,smallint) | 5 | search
btree | integer_ops | <(bigint,integer) | 1 | search
btree | integer_ops | <=(bigint,integer) | 2 | search
btree | integer_ops | =(bigint,integer) | 3 | search
btree | integer_ops | >=(bigint,integer) | 4 | search
btree | integer_ops | >(bigint,integer) | 5 | search
btree | integer_ops | <(bigint,smallint) | 1 | search
btree | integer_ops | <=(bigint,smallint) | 2 | search
btree | integer_ops | =(bigint,smallint) | 3 | search
btree | integer_ops | >=(bigint,smallint) | 4 | search
btree | integer_ops | >(bigint,smallint) | 5 | search
btree | integer_ops | <(integer,bigint) | 1 | search
btree | integer_ops | <=(integer,bigint) | 2 | search
btree | integer_ops | =(integer,bigint) | 3 | search
btree | integer_ops | >=(integer,bigint) | 4 | search
btree | integer_ops | >(integer,bigint) | 5 | search
btree | integer_ops | <(integer,smallint) | 1 | search
btree | integer_ops | <=(integer,smallint) | 2 | search
btree | integer_ops | =(integer,smallint) | 3 | search
btree | integer_ops | >=(integer,smallint) | 4 | search
btree | integer_ops | >(integer,smallint) | 5 | search
btree | integer_ops | <(smallint,bigint) | 1 | search
btree | integer_ops | <=(smallint,bigint) | 2 | search
btree | integer_ops | =(smallint,bigint) | 3 | search
btree | integer_ops | >=(smallint,bigint) | 4 | search
btree | integer_ops | >(smallint,bigint) | 5 | search
btree | integer_ops | <(smallint,integer) | 1 | search
btree | integer_ops | <=(smallint,integer) | 2 | search
btree | integer_ops | =(smallint,integer) | 3 | search
btree | integer_ops | >=(smallint,integer) | 4 | search
btree | integer_ops | >(smallint,integer) | 5 | search
(45 rows )
```

The operator class specifies the names of the "supporting" functions , which, **if encountered in a query, can use the index access method** when searching (Purpose = search) or sorting (Purpose = ordering) data.
The numbers in Strategy are predefined in the code implementing the access methods. For example, for btree the strategies are defined as `1 (<), 2 (<=), 3 (=), 4 (>=), 5 (>) .`
For hash indexes there is one strategy with number `1 (=) :`

```
postgres=# \dAo+ hash integer_ops
List of operators of operator families
AM | Operator family | Operator | Strategy | Purpose | Sort opfamily
------------+-----------------+----------+----------+---------+---------------
hash | integer_ops | =(bigint,bigint) | 1 | search |
hash | integer_ops | =(integer,integer) | 1 | search |
hash | integer_ops | =(smallint,smallint) | 1 | search |
hash | integer_ops | =(bigint,integer) | 1 | search |
hash | integer_ops | =(bigint,smallint) | 1 | search |
hash | integer_ops | =(integer,bigint) | 1 | search |
hash | integer_ops | =(integer,smallint) | 1 | search |
hash | integer_ops | =(smallint,bigint) | 1 | search |
hash | integer_ops | =(smallint,integer) | 1 | search |
```

```
(9 rows )
```

Hash indexes can only be used with exact match queries: equality (=).
The course does not include tasks like "let's look at everything there is". On the contrary, the course does not study the gist , gin , brin indices so as not to have unnecessary information.
Examples are given for one of the most commonly used index types: btree. Once you understand how the btree index works, you can understand how all other index types work by analogy. Lists of strategies for access methods are in the documentation
https://docs.tantorlabs.ru/tdb/ru/16_4/se/xindex.html
We'll cover operators in detail because it will help us understand what types of indexes can be created to serve queries. If a query has an operator, then indexes that have that operator in the list of strategies for that index type and data type can be used.

Part 2. Using Indexes with Integrity Constraints

1) Create a table with a primary key:

```
postgres=# drop table if exists t3;
DROP TABLE
postgres=# create table t3 (n int4 primary key, m int4);
CREATE TABLE
```

When the table was created, an index named `t 3_ pkey was created.` to support the primary key:

```
postgres=# \d+ t3
Table "public.t3"
Column | Type | Collation | Nullable | Default | Storage |Compression|Stats target|
--------+---------+-----------+----------+---------+---------+-----------+------------
n | integer | | not null | | plain | |
m | integer | | | | plain | |
Indexes:
"t3_pkey" PRIMARY KEY, btree (n)
Access method: heap
```

Table access method for a table: `heap` .

2) Create a composite index on columns **m , n** :

```
postgres=# create unique index concurrently t3_pkey1 on t3 (m,n);
CREATE INDEX
```

An index can be created with the CONCURRENTLY option. This option places a SHARE UPDATE EXCLUSIVE lock on the table for the duration of its operation (i.e., for the duration of index creation). The option allows SELECT, WITH, INSERT, UPDATE, DELETE, MERGE commands to be executed and allows the use of a fast path to object locking.
Creating an index can take a long time. Without CONCURRENTLY, the table is scanned once, with CONCURRENTLY, the table is scanned twice and three transactions are used to reduce the database horizon retention time.
Autovacuum is incompatible with creating, dropping, recreating indexes, regardless of whether CONCURRENTLY is used or not. Autovacuum skips tables if it cannot immediately obtain a lock.
Multiple indexes with the CONCURRENTLY option on the same table cannot be created simultaneously.
Multiple indexes without the CONCURRENTLY option can be created simultaneously on the same table and on different tables. Index creation commands can be run in different sessions.

3) Replace the integrity constraint with one command:

```
postgres=# ALTER TABLE t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pkey PRIMARY KEY USING INDEX t3_pkey1;
NOTICE: ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index "t3_pkey1" to "t3_pkey"
```

```
ALTER TABLE

postgres=# \d+ t3
Table "public.t3"
Column | Type | Collation | Nullable | Default | Storage |Compression|Stats target|
--------+---------+-----------+----------+---------+---------+-----------+------------
n | integer | | not null | | plain | |
m | integer | | not null | | plain | |
Indexes:
"t3_pkey" PRIMARY KEY, btree ( m, n)
Access method : heap
```

Compare with the output of the same command executed earlier. The integrity constraint **not has been added to column m null** and a new index was used for the primary key. The old index on column n was dropped and its space was freed. The index **t 3_ pkey 1** was renamed to index t 3_ pkey .
The command execution time consists of:
a) waiting for an exclusive lock to be obtained on a table
b) waiting for all buffers in the buffer cache that were used by the blocks of the index being deleted to be freed (the buffers will be added to the free list) . When using an unreasonably large buffer cache, searching for each block in the buffer cache may take a long time. Unreasonable growth of the buffer cache is caused by increasing the number of hash bucket slots in the Shared Buffer Lookup Table in some PostgreSQL forks. Tantor DBMS does not increase this number (NUM _ BUFFER _ PARTITIONS).

4) Create an index on column n:

```
postgres=# create index concurrently t3_pkey1 on t3 (n);
CREATE INDEX

postgres=# \d t3
Table "public.t3"
Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+---------
n | integer | | not null |
m | integer | | not null |
Indexes:
"t3_pkey" PRIMARY KEY, btree (m, n)
    "t3_pkey1" btree (n)
```

Index created. There are two indexes on the table. Both indexes will be updated simultaneously if you change rows in the table. Additional indexes reduce the performance of changing rows in tables. Why then are additional indexes created?
Additional indexes are created if the execution time of commands using additional indexes is significantly less than the execution time of commands without using these additional indexes. The lower time correlates with the cost of the execution plan and the CPU and I/O resources.

5) Try replacing the integrity constraint with one command:

```
postgres =# alter table t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pkey
PRIMARY KEY USING INDEX t3_pkey1;
ERROR: "t3_pkey1" is not a unique index
LINE 1: ALTER TABLE t3 DROP CONSTRAINT t3_pkey, ADD CONSTRAINT t3_pk...
^
DETAIL: Cannot create a primary key or unique constraint using such an index .
```

The created index cannot be used by integrity constraints. Why?
Because the index is non-unique. Non-unique indexes cannot be used by integrity constraints .

6) Try adding an integrity constraint in the NOT VALID state:

```
postgres=# alter table t3 ADD CONSTRAINT t3_pkey PRIMARY KEY USING INDEX t3_pkey1
NOT VALID;
ERROR: PRIMARY KEY constraints cannot be marked NOT VALID
```

In PostgreSQL, `PRIMARY KEY there is no NOT` state `VALID` . In DBMSs of other manufacturers, integrity constraints may have the `NOT state. VALID` .

7) If the index is not suitable for use by integrity constraints and does not speed up query execution, it should be dropped. Drop the index:

```
postgres =# drop index t 3_ pkey 1;
DROP INDEX
```

8) Try dropping the index used by the integrity constraint:

```
postgres=# drop index t3_pkey;
ERROR: cannot drop index t3_pkey because constraint t3_pkey on table t3 requires it
HINT: You can drop constraint t3_pkey on table t3 instead.
```

An index used by an integrity constraint cannot be dropped. The integrity constraint can be dropped.

9) Try creating a foreign key with the command:

```
postgres=# alter table t3 add constraint fk foreign key (m) references t3(n) not valid;
ERROR: there is no unique constraint matching given keys for referenced table "t3"
```

A foreign key can only be created on PRIMARY integrity constraints. KEY or UNIQUE .

10) Create external key by command :

```
postgres=# alter table t3 add constraint fk foreign key (n,m) references t3(m,n) not
valid;
ALTER TABLE
postgres=# \d+ t3
Table "public.t3"
Column | Type |Collation| Nullable |Default| Storage | Compression |Stats target|
--------+---------+---------+----------+-------+---------+-------------+-------------
n | integer | | not null | | plain | | |
m | integer | | not null | | plain | | |
Indexes:
"t3_pkey" PRIMARY KEY, btree (m, n)
Foreign key constraints:
"fk" FOREIGN KEY (n, m) REFERENCES t3(m, n) NOT VALID
Referenced by:
TABLE "t3" CONSTRAINT "fk" FOREIGN KEY (n, m) REFERENCES t3(m, n) NOT VALID
Access method : heap
```

Foreign key created without checking existing rows.
There is no index on the foreign key columns.

11) In this case, the foreign key works and is checked when inserting, deleting, changing rows in the table. Execute insert V table :

```
postgres=# insert into t3 values (2,3);
ERROR: insert or update on table "t3" violates foreign key constraint "fk"
DETAIL: Key (n, m)=(2, 3) is not present in table "t3".
postgres=# insert into t3 values (1,1);
INSERT 0 1
```

Inserting a row that violates a foreign key cannot be performed.
If the changes do not violate integrity constraints, the command is executed .

12) Perform a check on the rows for compliance with the integrity constraint:

```
postgres=# alter table t3 validate constraint fk;
ALTER TABLE
```

When validating rows, a ShareRowExclusive lock is requested on the tables:
a) on which a FOREIGN KEY is created (child table)
b) on the parent table that has PRIMARY KEY or UNIQUE referenced by the FOREIGN KEY.
In the example the table is the same.
The check may take a long time and depends on the number of rows in the child table. The ShareRowExclusive lock is held during the check.

If partitioned tables are used, the integrity constraint check status can be used by the planner to exclude partitions from scanning.

The index on the foreign key columns was not created during the check.
An index on FK is created if:
1) in the master table (where PK is) the PK column value is often updated or rows are deleted. These actions are undesirable and are avoided when designing applications.
2) when using PK-FK table joins. This is used very often, because this is what FK is for: it defines a connection (join) between tables.

Part 3. Characteristics of btree indexes

1) Create a table:

```
postgres=#
drop table if exists t;
create table t (id int8, s text storage plain);
create index t_idx on t (s text_ops) include (id);
insert into t values (1, repeat('a',2700));
DROP TABLE
CREATE TABLE
CREATE INDEX
```

2) Try inserting a row with a field with 2700 characters into the table:

```
postgres=# insert into t values (1, repeat('a',2700));
ERROR: index row size 2720 exceeds btree version 4 maximum 2704 for index "t_idx"
DETAIL: Index row references tuple (0,1) in relation "t".
HINT: Values larger than 1/3 of a buffer page cannot be indexed .
Consider a function index of an MD5 hash of the value, or use full text indexing.
```

The maximum size of an index record is 2704 bytes , which is approximately one third of the size of a block without a block header: 2704*3=8112 bytes.
Values greater than 1/3 of the block cannot be indexed .

To calculate the size of a row in a table, you can use the function:

```
postgres=# select pg_column_size(row(1::int4, repeat('a',2700))),
pg_column_size(row(repeat('a',2700)));
pg_column_size | pg_column_size
----------------+----------------
2732 | 2728
(1 row)
```

There are no functions for calculating the size of a record in an index block.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

3) See how many records fit into the intermediate index block:

```
postgres=# drop table if exists t3;
create table t3 (id bigserial primary key, s int4) with (autovacuum_enabled=off);
insert into t3 (s) select * from generate_series(1, 1000000);
select pg_indexes_size('t3');
select pg_relation_size('t3_pkey', 'main');
select * from bt_page_stats('t3_pkey',3);
DROP TABLE
CREATE TABLE
INSERT 0 1000000
pg_indexes_size
-----------------
22487040
(1 row)
pg_relation_size
------------------
22487040
(1 row)
blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------------+------------+-----------+--------------+-----------+-----------+-----------+------------+
3 | i | 286 | 0 | 15 | 2436 | 0 | 411 | 1 |
(1 row )
```

The number of records in the intermediate (internal) index block is 286.
The percentage of filling of intermediate blocks of the btree index is 70% and does not change.
The size of the main index layer is 22487040 bytes.
A large number of rows are inserted into the table so that the index has leaf, intermediate and root levels.

4) By default, the percentage of leaf blocks is 90% and can change. See how many records are in the index leaf block:

```
postgres =# select * from bt_page_stats ( ' t 3_pkey ' , 274 ) ;
 blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------------+------------+-----------+--------------+-----------+-----------+-----------+------------+
274 | l | 367 | 0 | 16 | 808 | 273 | 275 | 0 |
(1 row)
```

When filled (fillfactor = 90) by 90%, the number of records in the index block is 286*90%/70% = 367. All leaf blocks have 808 bytes free, except for the rightmost block. The indexed column is filled with a monotonically increasing sequence. New values are added to the rightmost leaf block. The rightmost block has **btpo_next=0 at each level** . For leaf blocks, type=l , and the numbering of levels also starts from the leaf level, so for leaf blocks, btpo_level = 0 . Data By sheet block :

```
postgres=# select * from bt_page_stats('t3_pkey',275);
blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------------+------------+-----------+--------------+-----------+-----------+-----------+------------+
275 | l | 82 | 0 | 16 | 6508 | 274 | 0 | 0 |
```

There are 82 records in the right leaf block, 6508 bytes are free in the block.

5) See how many records will fit in a leaf block if you set **fillfactor=100** :

```
postgres=# drop table if exists t3;
create table t3 (id bigserial primary key, s int4) with (autovacuum_enabled=off,
fillfactor=100);
alter index t3_pkey set ( fillfactor=100 );
insert into t3 (s) select * from generate_series(1, 1000000);
select pg_indexes_size('t3');
select pg_relation_size('t3_pkey', 'main');
select * from bt_page_stats('t3_pkey',3);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
select * from bt_page_stats('t3_pkey',4);
pg_indexes_size
------------------
20275200
(1 row)
pg_relation_size
-------------------
20275200
(1 row)
blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level
-------+------------+------------+-----------+---------------+-----------+-----------+-----------+-----------+
3 | i | 286 | 0 | 15 | 2436 | 0 | 411 | 1 |
(1 row)

blkno | type | live_items | dead_items | avg_item_size | free_size | btpo_prev | btpo_next | btpo_level |
-------+------------+------------+-----------+---------------+-----------+-----------+-----------+------------+
4 | l | 407 | 0 | 16 | 8 | 2 | 5 | 0 |
(1 row )
```

The number of records in the intermediate index blocks has not changed and remains 286, since the percentage of filling the intermediate btree index blocks is 70% and does not change. 30% will be used in case the leaf blocks are divided. When inserting rows into the t3 table (filling the indexed column with a monotonically increasing sequence), only the right blocks of the leaf and intermediate levels are divided.
The number of records that fit into leaf blocks increased by ~11% from 367 to 407 , which corresponds to an increase in `fillfactor` from 90% to 100%.
The index file size has decreased by ~10% from 22487040 to 20275200 bytes, which corresponds to an increase in `fillfactor` from 90% to 100%.

The length of a record in a leaf block is 16 bytes .

6) Look at the data from the header of a fully populated sheet block:

```
postgres=# select * from page_header(get_raw_page('t3_pkey', 4));
lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+----------
B/3DD2D598 | 0 | 0 | 1652 | 1664 | 8176 | 8192 | 5 | 0
(1 row)
```

The data area takes 407*16 = 6152 bytes. 8192-6152=1680 is overhead. The main part of the overhead is the variable part of the block header: 4 bytes per index record. 407*4 bytes (slot size in the block header)=1628 bytes. 1680-1628=52 bytes, which are used by the fixed part of the block header 24 bytes, unoccupied space 1664 - 1652 =12 bytes , special area of 8192-8176 = 16 bytes.

7) See how many blocks of different types are in the index:

```
postgres=# select type, count(*) from bt_multi_page_stats('t3_pkey',1,-1) group
by type order by 2;
type | count
------+-------
r | 1
i | 9
 l | 2464
(3 rows)
```

99.6%, that is, the overwhelming majority of blocks are sheet .
The percentage of leaf blocks will decrease if the length of indexed fields is larger. In this case, there will be fewer records in the leaf and intermediate blocks and the number of index levels will increase. Deduplication will increase the number of records in the leaf blocks of the index.

8) Run a handy query that outputs the index sizes and the index creation command :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
postgres=# select i.relname "table", indexrelname "index",
pg_INDEXES_size(relid) "indexes_size",
pg_RELATION_size(relid) "table_size ",
pg_TOTAL_RELATION_size(relid) "total",
pg_RELATION_size(indexrelid) "index_size",
reltuples::bigint "rows",
ii.indexdef ddl
from pg_stat_all_indexes i join pg_class c on (i.relid = c.oid)
join pg_indexes ii on (i.indexrelname = ii.indexname)
where i.schemaname not like 'pg_%' -- do not output service objects
order by pg_INDEXES_size(relid) desc, pg_RELATION_size(indexrelid) desc limit 1;
table | index | indexes_size | table_size | total | index_size | rows | ddl
-----------------+--------------------------+------------------+-------------+-----------+-----------+--------+
t3 | t3_pkey | 20275200 | 44285952 | 64593920 | 20275200 | -1 | CREATE UNIQUE INDEX t3_pkey ON public.t3 USING
btree (id) WITH (fillfactor='100')
(1 row)
```

Why is the number of rows in the table `rows = -1` ?
Because statistics have not been collected.

9) Collect statistics on table t3:

```
postgres=# analyze t3;
ANALYZE
```

10) Repeat request :

```
postgres=# select i.relname "table", indexrelname "index",
pg_INDEXES_size(relid) "indexes_size",
pg_RELATION_size(relid) "table_size ",
pg_TOTAL_RELATION_size(relid) "total",
pg_RELATION_size(indexrelid) "index_size",
reltuples::bigint "rows",
ii.indexdef ddl
from pg_stat_all_indexes i join pg_class c on (i.relid = c.oid)
join pg_indexes ii on (i.indexrelname = ii.indexname)
where i.schemaname not like 'pg_%' -- do not output service objects
order by pg_INDEXES_size(relid) desc, pg_RELATION_size(indexrelid) desc limit 1
\gx
-[ RECORD 1 ]+-------------------------------------------------------
table | t3
index | t3_pkey
indexes_size | 20275200
table_size | 44285952
total | 64593920
index_size | 20275200
rows | 1000000
ddl | CREATE UNIQUE INDEX t3_pkey ON public.t3 USING btree (id) WITH
                ( fillfactor = '100')
```

Now the number of rows in the table is `rows= 1000000` .
`limit 1` from the query , the query is convenient for quickly assessing the sizes of tables and indexes; find out which indexes to pay attention to. For example, large indexes, tables with a large number of indexes on one table.

Part 4. Navigating the btree index structure

1) The pageinspect extension has functions for viewing the structure of index blocks. Functions for btree indexes have the `bt_` **prefix** . The number of levels in a btree index tree is not included in the

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

statistics tables. The number of levels is stored in the metadata block and can be viewed using the bt_metap function . **view** the metadata for the t3_pkey index that was created in the previous part of the practice:

```
postgres=# select * from bt_metap('t3_pkey');
magic | version | root | level | fastroot | fastlevel |last_cleanup_|last_cleanup|allequalimage
| | | | | |num_delpages | _num_tuples|
--------+---------+------+-------+----------+-----------+-------------+-----------+------------
340322 | 4 | 412 | 2 | 412 | 2 | 0 | -1 | t
(1 row)
```

In the results of the function, only **level is of interest** and **root** .
Number of levels **level =2** . Numbering starts from zero. The index tree grows from bottom to top and zero corresponds to leaf blocks.
magic and version fields are used to quickly verify that an object is a btree index of a supported version. The "magic" number for btree indexes is 340322 (0 x 0531162).
Starting with PostgreSQL version 12, version 4 of indexes is used. An older version of the index may be encountered if the DBMS was updated from older versions. Indexes of older versions should be rebuilt, otherwise the innovations that appeared in new versions will not be used.
The root block of the index **is root=412** . The number is the ordinal number of the block from the beginning of the first file of the index data layer.
fastroot and fastlevel are used for minor optimization of index search. You can ignore these columns. If you delete all the rows in the table, the number of index levels does not decrease. In this case, the root block will have one heir. In this case, fastroot will become a block that has several heirs and from which you can start searching. In the example, fastroot points to root.

2) Function bt_page_stats is needed to navigate the index structure and produces one row for each index block. Look at the data for the root index block t3_pkey :

```
postgres =# select * from bt_page_stats ( ' t 3_pkey ' , 412 ) ;
blkno|type|live_items|dead_items|avg_item_size|free_size| btpo_prev | btpo_next | btpo_level | btpo_flags
-----+----+----------+----------+-------------+---------+-----------+-----------+------------+-----------
412 | r | 9 | 0 | 15 | 7976 | 0 | 0 | 2 | 2
(1 row)
```

For navigation are used fields btpo_prev And btpo_next . These fields indicate the numbers of blocks to the left and right on the same level. btpo_prev=0 means that the block is the leftmost one, btpo_ next =0 means that the block is the rightmost one on its level. The root block is the only one on its level, so the values are zero.
The block type is specified in the field type . The values in this column are: r - root ; i - internal ; l - leaf (list), e - ignored , d - deleted leaf (deleted leaf), D - deleted internal (deleted internal).
avg_item_size shows the calculated value of the average size of an index entry in this block. Entries are aligned to 8 bytes.
live_items - how many entries are in this block.

3) To view index records, use the bt_page_items function . Look at the index records in the root block of the t3_pkey index :

```
postgres =# select itemoffset , ctid , itemlen , nulls , vars , dead , htid , tids , data
from bt _ page _ items (' t 3_ pkey ', 412) order by 1;
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+----------+---------+-------+------+------+------+------+------------------------
1 | ( 3 , 0 ) | 8 | f | f | | | |
2 | (411, 1 ) | 16 | f | f | | | | ff from 3 01 00 00 00 00 00
         3 | ( 698 , 1 ) | 16 | f | f | | | | fd 87 03 00 00 00 00 00
4 | (984, 1 ) | 16 | f | f | | | | fb 4 b 05 00 00 00 00 00
5 | (1270, 1 ) | 16 | f | f | | | | f 9 0 f 07 00 00 00 00 00
6 | (1556, 1 ) | 16 | f | f | | | | f 7 d 3 08 00 00 00 00 00
7 | (1842, 1 ) | 16 | f | f | | | | f 5 97 0 a 00 00 00 00 00
8 | (2128, 1 ) | 16 | f | f | | | | f 3 5 b 0 c 00 00 00 00 00
          9 | ( 2414 , 1 ) | 16 | f | f | | | | f 1 1 f 0 e 00 00 00 00 00
( 9 rows )
```

tantor

The root block contains 9 records that point to index blocks. The ctid column can store references to index blocks or table rows. When storing references to index blocks, the row identifier does not matter: in the example given, the row identifier has the values 0 or 1 . If 1 , then the data field stores the minimum value that is present in the child leaf block. If zero, then the data field is empty (treated as "minus infinity", i.e. the boundary is unknown) and this reference leads to the leftmost child block.

Since the block is root, the first entry (itemoffset=1) does not store HighKey. HighKey is not stored in the rightmost blocks of each level, since there is no point in designating the right boundary - the block is already the rightmost. HighKey is used to check whether it is necessary to read a block to the right.

data field of the current and next (itemoffset+1) records specifies the range that must contain the value that is searched in the index .

Index records are stored in an ordered manner, the order is given in the itemoffset column . The first row itemoffset =1 , ctid=(3,0) points to the lower-level block, which will be the leftmost at its level. The last row itemoffset =9 , ctid=(2414,1) points to the lower-level block, which will be the rightmost at its level.

The index uses the suffix truncation optimization and the truncation of indexed columns in the data field is its consequence. Because of this optimization, the btree index used in PostgreSQL can be called "Simple Prefix B-Tree". Simple because whole fields are truncated (whole "attribute" truncation). For a single-column index, there is an empty space left, which is treated as minus infinity.

The length of the first record due to the absence of a value in the data field is 8 bytes: itemlen=8 .

4) See what **the bt_page_stats function produces** by blocks with itemoffset =1,9,2 :

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size
size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('t3_pkey', 3 );
blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
     3 | i | 286 | 0 | 15 | 2436 | 0 | 411 | 1 | 0
(1 row )
```

3 is the leftmost block on its level, as indicated by btpo_prev=0 .
in block 3. The block is 70% full, which is the default fill percentage for intermediate blocks.

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size
size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('t3_pkey', 2414 );
blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
  2414 | i | 184 | 0 | 15 | 4476 | 2128 | 0 | 1 | 0
(1 row )
```

2414 is the rightmost block on its level, as indicated by btpo_next=0 .
In 2414 block 184 records. This is the rightmost block of its level, it is filled not by 70%, but less. This happened because when filling with a monotonically increasing sequence, insertions are performed in the right leaf block and it is this block that is divided. Insertion of a record with a reference to a new leaf block is performed in the rightmost block of the higher level. The rightmost blocks of each level are divided. After division, the records are redistributed between the two blocks. In the block "to the left" there remain 70% of the records for the intermediate blocks. When dividing a leaf block, in the block "to the left" there remain fillfactor records. The remaining records of both the intermediate and leaf blocks remain in the "right" block and there are fewer of them than in the block "to the left" of it. That is why in the "right" 2414 block 184 records, which is less than 286.

```
postgres =# select blkno , type , live _ items live , dead_items dead ,
avg_item_size size , free_size free , btpo_prev , btpo_next , btpo_level l , btpo
_ flags f from bt_page_stats ( ' t 3_pkey ' , 411 ) ;
```

```
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
411 | i | 286 | 0 | 15 | 2436 | 3 | 698 | 1 | 0
(1 row )
```

Block `411 is to the right of block 3` , as indicated by `btpo_ prev = 3` .
Values `of btpo_prev` and `btpo _ next` correspond to the order of the entries in the `itemoffset column` of the block above. In the example, `btpo_next = 698 , which` corresponds to the entry with `itemoffset =3` in the block above.
In all intermediate blocks, except for the "right" ones, the number of records (`286`) and free space (`free`) correspond to 70% filling because the insertions were in the right blocks. As a result of dividing the right blocks (when there was no space left for insertion), the left blocks were filled by 70% (intermediate) or up to fillfactor (leaf).

5) Block 2414 is the rightmost block on the intermediate level. There are 184 records in the block. Look at the first and last records:

```
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2414 ) order by 1 limit 2 ;
itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+----------+---------+-------+------+------+------+------+-----------
1 | (2290.0) | 8 | f | f | | | |
2 | (2291.1) | 16 | f | f | | | | 87 21 0e 00 00 00 00 00
(2 rows)
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2414) order by 1 desc limit 2 ;
itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+----------+---------+-------+------+------+------+------+-----------
184 | (2474.1) | 16 | f | f | | | | 2b 42 0f 00 00 00 00 00
183 | (2473.1) | 16 | f | f | | | | 95 40 0f 00 00 00 00 00
(2 rows)
```

There are three levels in the block. What is the number of the rightmost sheet block? 2474.

6) Look at the statistics and contents of the rightmost leaf block:

```
postgres =# select blkno , type , live _ items live , dead_items dead ,
avg_item_size size , free_size free , btpo_prev , btpo_next , btpo_level l , btpo
_ flags f from bt_page_stats ( ' t 3_pkey ' , 2474 ) ;
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
2474 | l | 22 | 0 | 16 | 7708 | 2473 | 0 | 0 | 1
(1 row )
```

In statistics `type = l` , it means that the block is leaf. `btpo _ next =0` means that the block is the rightmost.

```
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2474);
itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+-----------+---------+-------+------+-------+------+-----------+------+------------------------
1 | (5405.54) | 16 | f | f | f | (5405.54) | | 2b 42 0f 00 00 00 00 00
2 | (5405.55) | 16 | f | f | f | (5405.55) | | 2c 42 0f 00 00 00 00 00
3 | (5405.56) | 16 | f | f | f | (5405.56) | | 2d 42 0f 00 00 00 00 00
4 | (5405.57) | 16 | f | f | f | (5405.57) | | 2e 42 0f 00 00 00 00 00
5 | (5405.58) | 16 | f | f | f | (5405.58) | | 2f 42 0f 00 00 00 00 00
6 | (5405.59) | 16 | f | f | f | (5405.59) | | 30 42 0f 00 00 00 00 00
7 | (5405.60) | 16 | f | f | f | (5405.60) | | 31 42 0f 00 00 00 00 00
8 | (5405.61) | 16 | f | f | f | (5405.61) | | 32 42 0f 00 00 00 00 00
9 | (5405.62) | 16 | f | f | f | (5405.62) | | 33 42 0f 00 00 00 00 00
10 | (5405.63) | 16 | f | f | f | (5405.63) | | 34 42 0f 00 00 00 00 00
11 | (5405.64) | 16 | f | f | f | (5405.64) | | 35 42 0f 00 00 00 00 00
12 | (5405.65) | 16 | f | f | f | (5405.65) | | 36 42 0f 00 00 00 00 00
13 | (5405.66) | 16 | f | f | f | (5405.66) | | 37 42 0f 00 00 00 00 00
14 | (5405.67) | 16 | f | f | f | (5405.67) | | 38 42 0f 00 00 00 00 00
15 | (5405.68) | 16 | f | f | f | (5405.68) | | 39 42 0f 00 00 00 00 00
16 | (5405.69) | 16 | f | f | f | (5405.69) | | 3a 42 0f 00 00 00 00 00
17 | (5405.70) | 16 | f | f | f | (5405.70) | | 3b 42 0f 00 00 00 00 00
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
18 | (5405.71) | 16 | f | f | f | (5405.71) | | 3c 42 0f 00 00 00 00 00
19 | (5405.72) | 16 | f | f | f | (5405.72) | | 3d 42 0f 00 00 00 00 00
20 | (5405.73) | 16 | f | f | f | (5405.73) | | 3e 42 0f 00 00 00 00 00
21 | (5405.74) | 16 | f | f | f | (5405.74) | | 3f 42 0f 00 00 00 00 00
22 | (5405.75) | 16 | f | f | f | (5405.75) | | 40 42 0f 00 00 00 00 00
(22 rows)
```

This is the content of the rightmost leaf block. All ctids of the rightmost leaf block contain references to the table rows. The **htid** column (heap tuple id) stores the same value as `ctid` . Why is the value duplicated in ctid and htid ? The btree algorithm is optimized to work under conditions of minimum locks and to minimize block splits. When one process reads index blocks, other processes can change its structure. When navigating through blocks, there is no single picture ("read integrity"). Additional fields are used to detect contradictions.

7) Check that ctid refers to a table row. Run a query on table t3 using the `ctid service column` :

```
postgres=# select * from t3 where ctid=' (5405.54 )';
id | s
--------+--------
999979 | 999979
(1 row)

postgres=# select * from t3 where ctid=' (5405.75) ';
id | s
---------+---------
1000000 | 1000000
(1 row )
```

The row **ctid = ' (5405,75) '** was the most recently added to the table.

8) Insert a row into the table and check that a record has been inserted into the right leaf block :

```
postgres=# insert into t3 values(default);
INSERT 0 1
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2474) where itemoffset>21;
itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+-----------+---------+-------+------+------+-----------+------+------------------------
22 | (5405.75) | 16 | f | f | f | (5405.75) | | 40 42 0f 00 00 00 00 00
    23 | (5405.76) | 16 | f | f | f | (5405.76) | | 41 42 0f 00 00 00 00 00
(2 rows)
```

9) Look at the first entries in any leaf block except the rightmost one:

```
postgres=# select itemoffset, ctid, itemlen, nulls, vars, dead, htid, tids, data
from bt_page_items('t3_pkey', 2473) limit 3;
itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+-----------+---------+-------+------+------+-----------+------+----------------------
1 | (5405.1) | 16 | f | f | | | | 2b 42 0f 00 00 00 00 00
2 | (5403.18) | 16 | f | f | f | (5403.18) | | 95 40 0f 00 00 00 00 00
3 | (5403.19) | 16 | f | f | f | (5403.19) | | 96 40 0f 00 00 00 00 00
(3 rows)
```

The first row (`itemoffset=1`) in leaf blocks, except for the "rightmost" one, always stores a service value called "High key". When inserting a new block into the index structure, High keys and references to neighboring blocks of the same level are updated. **High key stores the largest value that occurs in this index block** .

High key (the first row of a leaf block except the rightmost leaf block) is always checked when searching by index. Why? During the descent from the previous level to the leaf level, another process could have already split the block to which they are descending and redistributed references to table rows, which means that the value being sought is in the block (or even blocks) to the right of the one they are descending to. If the High key value of a leaf block differs from the value in the reference to the next leaf block , then the process must move to the right along the leaf level and check whether the

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

desired value is there. The reference to the next leaf block is in the higher block in the itemoffset +1 entry (in the example 184).

Part 5. Deduplication in btree indexes

Deduplication appeared in PostgreSQL version 13 and uses fields to store information that are intended for other purposes without deduplication. This made it possible not to radically change the index structure and not require rebuilding indexes when migrating to a new version of PostgreSQL.

1) Create a table with one column of type int4 or int8 . Insert 407 rows with the same value 1 into the table . Look at the contents of the index block:

```
postgres =# drop table if exists td ;
create table td(id int8 ) with (autovacuum_enabled=off);
create index td_idx on td(id);
insert into td select 1 from generate_series(1, 407);
select magic, root, level, fastroot, fastlevel, allequalimage from
bt_metap('td_idx');
select blkno, type, live_items live, dead_items dead, avg_item_size size,
free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('td_idx',1);
select itemoffset o, ctid, itemlen, htid, data, substring(tids::text for 34) tids
from bt_page_items('td_idx',1) limit 3;
DROP TABLE
CREATE TABLE
CREATE INDEX
INSERT 0 407
magic | root | level | fastroot | fastlevel | allequalimage
--------+-----------+-------+---------+-----------+---------------
340322 | 1 | 0 | 1 | 0 | t
(1 row )
```

The root block is block number 1 . To support deduplication, function number 4 (BTEQUALIMAGE_PROC) must be defined in the operator class for the data type being indexed . If **allequalimage=t** , then the function is defined and deduplication is supported.
Deduplication is not supported with the following data types: numeric, jsonb, float4, float8 , arrays, composite, range types. Indexes with INCLUDE columns do not support deduplication. Such indexes have **allequalimage= f** .

```
 blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+---
1 | l | 407 | 0 | 16 | 8 | 0 | 0 | 0 | 3
(1 row )
```

Block type number 1 is specified as leaf in the block itself. There are no blocks to the left and right of this block (btpo_prev=0 and btpo_next=0), the block is the only one at its level. There are 407 records in the block (live =407). There are 8 bytes free in the block (free=8).
The index consists of two blocks: a zero block with metadata and a single block.

```
 o | ctid | itemlen | htid | data | tids
---+-------+---------+-------+----------+-----------
1 | (0,1) | 16 | (0,1) | 01 00 00 00 00 00 00 00 |
2 | (0,2) | 16 | (0,2) | 01 00 00 00 00 00 00 00 |
3 | (0.3) | 16 | (0.3) | 01 00 00 00 00 00 00 00 |
(3 rows)
```

Block has format right sheet block . In each record, the ctid field stores a reference to a table row, and the data field stores an indexed value. Since the block is right, it is completely filled so that there is no room to insert a new record.

2) Insert one line :

```
postgres=# insert into td values(1);
INSERT 0 1
```

3) Look how the contents of block 1 of the index have changed:

```
postgres=# select blkno, type, live_items live, dead_items dead, avg_item_size
size, free_size free, btpo_prev, btpo_next, btpo_level l, btpo_flags f from
bt_page_stats('td_idx',1);
 select itemoffset o, ctid, itemlen, htid, data, substring(tids::text for 34)
tids from bt_page_items('td_idx',1);

blkno | type | live | dead | size | free | btpo_prev | btpo_next | l | f
-------+------+------+------+------+------+-----------+-----------+---+--
1 | l | 3 | 0 | 832 | 5640 | 0 | 0 | 0 | 3
(1 row )
```

Deduplication has been performed. Deduplication is performed if the block would have to be divided: there is no space in the block or the fillfactor is exceeded.
The block had 407 records, but now it has 3 records.
There are 5640 bytes free in the block.

```
 o | ctid | itemlen | htid | data | tids
---+-----------+---------+----------+----------+-----------------------------------
1 | ( 16.8414 ) | 1352 | (0,1) | 01 00 00 00 00 00 00 00 | {"(0.1)","(0.2)","(0.3)","(0.4)","
2 | ( 16.8377 ) | 1128 | (0.223) | 01 00 00 00 00 00 00 00 | {"(0.223)","(0.224)","(0.225)","(0
3 | (1.182) | 16 | (1.182) | 01 00 00 00 00 00 00 00 |
(3 rows )
```

The data field stores the value of the indexed columns. In the example, this is the integer 1 .
The values of the indexed columns are stored in the index record, and references to the table rows are stored in the tids column (tuple ids, table row identifiers) as a sorted array of ctid values.
There is no deduplication in the itemoffset=3 line (tids is empty) and the table row is pointed to by ctid=(1,182).
In the first and second lines, tids point to table rows.
The first tid of tids is stored in htid.
The ctid of the first and second entries do not store references to blocks, but service data about tids. For example, the ctid of intermediate blocks will store the block number in the index, referring to the lower level, and the second part of the ctid will store the number of elements in tids.

tids array is specified in the `itemlen column` :

```
postgres=# select itemoffset o, ctid, itemlen, htid, pg_column_size(tids) size,
cardinality(tids) from bt_page_items('td_idx',1);
 o | ctid | itemlen | htid | size | cardinality
---+-----------+---------+---------+------+-------------
1 | (16.8414) | 1352 | (0,1) | 1356 | 222
2 | (16.8377) | 1128 | (0.223) | 1134 | 185
3 | (1.182) | 16 | (1.182) | |
(3 rows)
```

Size type ctid 6 bytes . The number of elements in the array is given by **the cardinality (tids) function** . Calculating the size of the array in the tids column: 222*6=1332. 185*6=1110.
This corresponds to the values returned by the pg_column_size(tids) function . The function returns values increased by 24 bytes (1332+24= 1356 and 1110+24=1134), since it is intended to return the size of a table row. 24 bytes is the minimum size of a row header.

4) Let's see how to determine whether an index supports deduplication.
For indexes that do not support deduplication, **allequalimage = f** .

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Deduplication is not supported with data types: `numeric, jsonb, float4, float8` , arrays, composite types, range types :

```
postgres=# drop table if exists td;
create table td(id float8 );
create index td_idx on td(id);
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
allequalimage
---------------
 f
(1 row)

postgres=# drop table if exists td;
create table td(id int8 [] );
create index td_idx on td(id);
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
allequalimage
---------------
 f
(1 row)

postgres=# drop table if exists td;
create table td(id jsonb );
create index td_idx on td(id);
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
allequalimage
---------------
 f
(1 row)

postgres=# drop table if exists td;
create table td(n timestamp , n1 date , n2 integer , n3 char , n4 text , n5
varchar );
create index td_idx on td (n,n1,n2,n3,n4,n5);
select allequalimage from bt_metap('td_idx');
DROP TABLE
CREATE TABLE
CREATE INDEX
allequalimage
---------------
 t
(1 row )
```

A composite index supports deduplication if the key data types support it.

```
postgres=# create index td1_idx on td (n) include (n1);
select allequalimage from bt_metap('td1_idx');
CREATE INDEX
allequalimage
---------------
 f
(1 row)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Indexes with INCLUDE columns do not support deduplication even if the data types do.

Part 6. Indexes in descending order

By default, the index is built in ascending order, i.e. smaller values are on the "left" and larger values are on the "right". When creating an index, you can specify the reverse order: DESC . You should not do this for indexes filled with an ascending sequence. The ASC and DESC property when creating an index does not affect the efficiency of the index use by the planner (for example, ORDER BY ASC or DESC). This property affects the index filling: the right blocks in the index differ from the rest in that they are optimized for inserts. It is desirable that inserts are performed mainly in the right leaf block of the index.

1) Enable command execution time measurement:

```
postgres=# \timing
Timing is on.
```

2) Create a table and index in direct and reverse sort order:

```
postgres=# drop table if exists t3;
create table t3 (id bigserial, s int4) with (autovacuum_enabled=off,
fillfactor=100);
create unique index if not exists t3_pkey on t3 using btree (id) include (s)
with (fillfactor=100, deduplicate_items=off);
insert into t3 (s) select * from generate_series(1, 1000000);
select pg_relation_size('t3_pkey', 'main');

drop table if exists t3;
create table t3 (id bigserial, s int4) with (autovacuum_enabled=off,
fillfactor=100);
create unique index if not exists t3_pkey on t3 using btree (id DESC nulls first
) include (s) with (fillfactor=100, deduplicate_items=off);
insert into t3 (s) select * from generate_series(1, 1000000);
select pg_relation_size('t3_pkey', 'main');

DROP TABLE
Time: 29.181 ms
CREATE TABLE
Time: 5.667 ms
CREATE INDEX
Time: 15.279 ms
INSERT 0 1000000
Time: 4351.432 ms (00:04.351)
pg_relation_size
------------------
         28467200
(1 row)
Time: 0.229 ms

DROP TABLE
Time: 23.951 ms
CREATE TABLE
Time: 7.504 ms
CREATE INDEX
Time: 11.651 ms
INSERT 0 1000000
Time: 5740.328 ms (00:05.740)
pg_relation_size
------------------
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
         56401920
(1 row)
Time: 0.307 ms
```

Changing the order significantly impacted insertion speed and index size.
The index size has increased by 2 times. The insertion speed has decreased by 32%.
If the values were filled in descending order, then the DESC index would be more optimal than ASC .

3) Rebuild the index and see if the index size has decreased:

```
postgres=# reindex index t3_pkey;
REINDEX
postgres=# select pg_relation_size('t3_pkey', 'main');
pg_relation_size
------------------
28475392
(1 row)
Time: 1077.214 ms (00:01.077)
```

The index size has decreased. The index rebuild was efficient. The index was updated inefficiently when inserting rows into the table. Updates occurred in the left blocks of the index, which are not optimized for inserts, unlike the right blocks of the btree index.

When inserting rows into a table with values that do not fall into the right block, the index structure is filled inefficiently and inserts are slow.
The reason for the slowdown in inserts is not so much the excessive division of blocks, but the lack of fastpath optimization. The process that inserted into the right leaf block remembers the reference to it during the next insert if the new value is greater than the previous one (or empty) and does not go from the root to the leaf block. Fastpath optimization is also used when inserting into a datetime column filled by DEFAULT with the insertion time. The process forgets the block address and starts searching from the root again if for some reason it inserted (records are only inserted into the index, they are not changed, and are deleted only by vacuum) not into the rightmost block. Fastpath is used when the number of levels in the index is 2 or more.

In addition to the values that are filled by the sequence, you also need to remember about empty values.
By default, empty values are stored "on the right" (in the right index blocks). This can be overridden by specifying NULLS FIRST .
When redefining the order, application developers usually assume that what should be returned preferentially first when sorting. This idea is incorrect, since the server process slides through blocks in both directions with equal efficiency.
Using NULLS FIRST can affect performance: if NULL is inserted into the index when inserting rows into a table (when inserting a row into a table, the value of the indexed column is not set, but is updated later, and the updates are distributed over time, not bulk), then fastpath optimization stops working, since NULLs will be in the leftmost leaf block, and fastpath only works with the right block. Inserting rows with NULLs will slow down when using NULLS FIRST . The performance degradation is the same as using DESC in the example given, so the example for NULLS FIRST is not given.

4) Turn it off measurement time :

```
postgres=# \timing
Timing is off.
```

Part 7. Covering indices and Index Only Scan

t 3_ pkey index was created with the `include ( s ) option` , meaning that the leaf blocks of the index store the values of column s. Storing the column values increases the size of the index.

the Index access method can be used. Only Scan . This method is used if the columns referenced in the query are present in the index. We can say that the query is "covered" by the index. Then the index is called "covering" for the query. The complexity of executing the command using the Index method Only Scan is significantly reduced because there is no need to access table blocks. All values needed by the query are taken from the index.

Columns added **by include (..)** do not affect the index structure.

Why not add the column to the key columns? If the column data type does not support the comparison operation, then such data type cannot be added to the key columns. Also **include (..)** columns are missing from the intermediate blocks, which slightly reduces the size of the index.

1) Look plan execution commands :

```
postgres=# explain select * from t3 where id=1;
QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using t3_pkey on t3 (cost=0.42..8.44 rows=1 width=12)
Index Cond: (id = 1)
(2 rows)

postgres=# explain select * from t3 where id=4 and s>3;
QUERY PLAN
-----------------------------------------------------------------------
 Index Only Scan using t3_pkey on t3 (cost=0.42..8.45 rows=1 width=12)
Index Cond: (id = 4)
Filter: (s > 3)
(3 rows )
```

Index method is used **Only Scan** .

2) Look at the command execution plan:

```
postgres=# explain select s from t3 where id>4 and id < 10;
QUERY PLAN
--------------------------------------------------------------------------
Bitmap Heap Scan on t3 (cost=123.67..5717.65 rows= 5000 width=4)
Recheck Cond: ((id > 4) AND (id < 10))
-> Bitmap Index Scan on t3_pkey (cost=0.00..122.42 rows=5000 width=0)
Index Cond: ((id > 4) AND (id < 10))
(4 rows )
```

`Index Only Scan` is not used. The planner is very wrong about the number of rows: according to its estimate, the query returns `5000` rows.

3) Collect statistics:

```
postgres =# analyze t 3;
ANALYZE
```

4) Repeat command :

```
postgres=# explain select s from t3 where id>4 and id < 10;
QUERY PLAN
----------------------------------------------------------------------
Index Only Scan using t3_pkey on t3 (cost=0.42..8.53 rows=5 width=4)
Index Cond: ((id > 4) AND (id < 10))
(2 rows)
```

Index has started to be used Only Scan .
The row count estimate becomes correct **rows =5** .

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

5) Only columns can be present in `include , but not` expressions :

```
postgres=# create unique index if not exists t3_pkey1 on t3 using btree (id)
include ( UPPER(s) );
ERROR: expressions are not supported in included columns
```

Part 8. Partial indices

Partial indexes are created on a portion of the table rows. The portion of the rows is determined by the WHERE predicate , `which` is specified when creating the index and makes the index partial.
Partial indexes are useful because they avoid indexing the most frequently occurring values. A most frequently occurring value is a value that is present in a significant percentage of all rows in a table. When searching for the most frequently occurring values, the index will not be used anyway, since it would be more efficient to scan all rows in the table. There is no point in indexing rows with the most frequently occurring values. By excluding such rows from the index, you can reduce the size of the index, which will speed up the vacuuming of the table. It also speeds up changes to table rows if the index is not affected.
The second reason why a partial index is used is when there are no requests to some of the table rows. If there are requests, then not index access is used, but a full table scan.
A partial index can be unique .
It is not worth creating a large number of partial indexes that index different rows. The more indexes on a table, the lower the performance of commands that change data; autovacuum; the probability of using the fast path of locks decreases.

1) Look at the size of the `t3_pkey index` :

```
postgres=# select pg_relation_size('t3_pkey', 'main');
pg_relation_size
------------------
28475392
(1 row )
```

2) Drop the index and create a partial index on rows where `s <1000` :

```
postgres=# drop index t3_pkey;
DROP INDEX
postgres=# create unique index t3_pkey on t3 using btree (id) include (s) WHERE
s<1000 ;
CREATE INDEX
```

3) Look at the size of the index:

```
postgres=# select pg_relation_size('t3_pkey', 'main');
pg_relation_size
------------------
49152
(1 row )
```

The size of the index is significantly smaller than the index for all rows.

4) Perform request :

```
postgres=# explain select * from t3 where id=4 and s>3;
QUERY PLAN
------------------------------------------------------------
Gather (cost=1000.00..12656.10 rows=1 width=12)
Workers Planned: 2
```

```
-> Parallel Seq Scan on t3 (cost=0.00..11656.00 rows=1 width=12)
Filter: ((s > 3) AND (id = 4))
(4 rows)
```

Although the query returns rows that are referenced in the index, the planner does not know about this.

5) Execute a query with an explicit condition s<3:

```
postgres=# explain select * from t3 where id=4 and s<3;
QUERY PLAN
------------------------------------------------------------------------
Index Only Scan using t3_pkey on t3 (cost=0.28..8.29 rows=1 width=12)
Index Cond: (id = 4)
Filter: (s < 3)
(3 rows)
```

Partial index is used.

Part 9. Studying the structure of the btree index

The chapter provided an example of the index structure.

1) Create a table, index, insert three rows:

```
postgres=# drop table if exists t;
create table t(s text storage plain) with (autovacuum_enabled=off,
fillfactor=10);
create index t_idx on t (s) with (fillfactor=10, deduplicate_items = off);
insert into t values (repeat('a',2500));
insert into t values (repeat('b',2500));
insert into t values (repeat('c',2500));
```

The size of the fields is chosen so that one row fits into the table block. The index block fits 3-4 rows.

2) The address of the root block of the index and the number of levels can be viewed using the query:

```
postgres=# select root, level, fastroot, fastlevel, allequalimage from
bt_metap('t_idx');
root | level | fastroot | fastlevel | allequalimage
------------+-------+----------+-----------+---------------
    1 | 0 | 1 | 0 | t
(1 row)
```

Statistics for all index blocks can be viewed using the query:

```
postgres=# select blkno, type, live_items live, avg_item_size size, free_size
free, btpo_prev prev, btpo_next next, btpo_level level, btpo_flags fl from
bt_multi_page_stats('t_idx',1,-1);
blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
1 | l | 3 | 2512 | 600 | 0 | 0 | 0 | 3
(1 row)
```

The contents of the index block can be viewed using the query:

```
postgres=# select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',1);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
3 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
(3 rows )
```

Based on these queries, we can draw the index structure:

0 root 1
level 0
fastroot 1
fastlevel 0

1(0,1)a
(1,1)b
(2,1)c

There are two blocks in the index: block zero always contains metadata; block number 1 contains three rows.
For compactness, the value is displayed with one letter, not 2500 letters. The letter a in hexadecimal is 61, the letter b is 62, the letter c is 63.

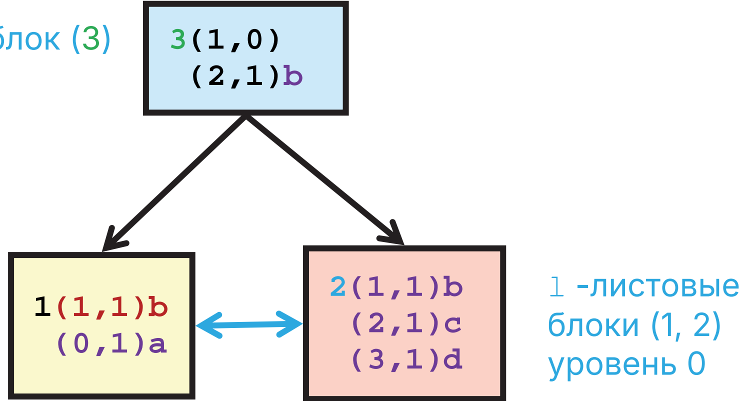
Next, inserting one row at a time, queries will be executed and based on the queries, the index structure will be drawn. The queries in the following points do not necessarily have to be executed, it is enough to view the query results and compare them with the drawing. This is useful to understand how the index grows.

4) After inserting the fourth row, you can run the following queries and, based on their results, draw the index structure:

```
postgres=# insert into t values (repeat('d',2500));
select ctid, left(s, 24) from t;
select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev, btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
INSERT 0 1
ctid | left
--------+------------------------
(0,1) | aaaaaaaaaaaaaaaaaaaaaaaa
(1,1) | bbbbbbbbbbbbbbbbbbbbbbbb
(2,1) | cccccccccccccccccccccccc
(3.1) | dddddddddddddddddddddd
(4 rows)

blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
1 | l | 2 | 2512 | 3116 | 0 | 2 | 0 | 1
2 | l | 3 | 2512 | 600 | 1 | 0 | 0 | 1
3 | r | 2 | 1260 | 5620 | 0 | 0 | 1 | 2
(3 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (1,1) | 2512 | | 20 27 00 00 62 62 | b
2 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
(2 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
2 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
3 | (3.1) | 2512 | (3.1) | 20 27 00 00 64 64 | d
(3 rows)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (1.0) | 8 | | |
2 | (2,1) | 2512 | | 20 27 00 00 62 62 | b
(2 rows)
```

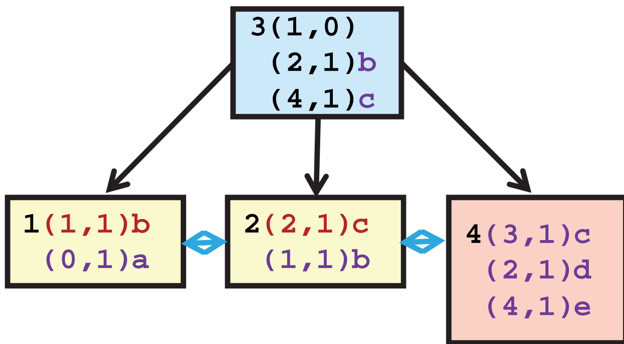


Root became block 3. With further insertions , the root block will also change.

5) After inserting the fourth row, you can repeat the queries by adding a query for block 4. Blocks are added to the index one by one and block 4 will be added to the index.

```
postgres=# insert into t values (repeat('e',2500));
select ctid, left(s, 24) from t;
select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev, btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
INSERT 0 1
ctid | left
--------+------------------------
(0,1) | aaaaaaaaaaaaaaaaaaaaaaaa
(1,1) | bbbbbbbbbbbbbbbbbbbbbbbb
(2,1) | cccccccccccccccccccccccc
(3.1) | dddddddddddddddddddddd
(4.1) | eeeeeeeeeeeeeeeeeeeeeeee
(5 rows)

blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
1 | l | 2 | 2512 | 3116 | 0 | 2 | 0 | 1
2 | l | 2 | 2512 | 3116 | 1 | 4 | 0 | 1
3 | r | 3 | 1677 | 3104 | 0 | 0 | 1 | 2
4 | l | 3 | 2512 | 600 | 2 | 0 | 0 | 1
(4 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (1,1) | 2512 | | 20 27 00 00 62 62 | b
2 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
(2 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (2,1) | 2512 | | 20 27 00 00 63 63 | c
2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
(2 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (1.0) | 8 | | |
2 | (2,1) | 2512 | | 20 27 00 00 62 62 | b
3 | (4.1) | 2512 | | 20 27 00 00 63 63 | c
(3 rows)
```

```
 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
 1 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
 2 | (3.1) | 2512 | (3.1) | 20 27 00 00 64 64 | d
 3 | (4.1) | 2512 | (4.1) | 20 27 00 00 65 65 | e
(3 rows)
```

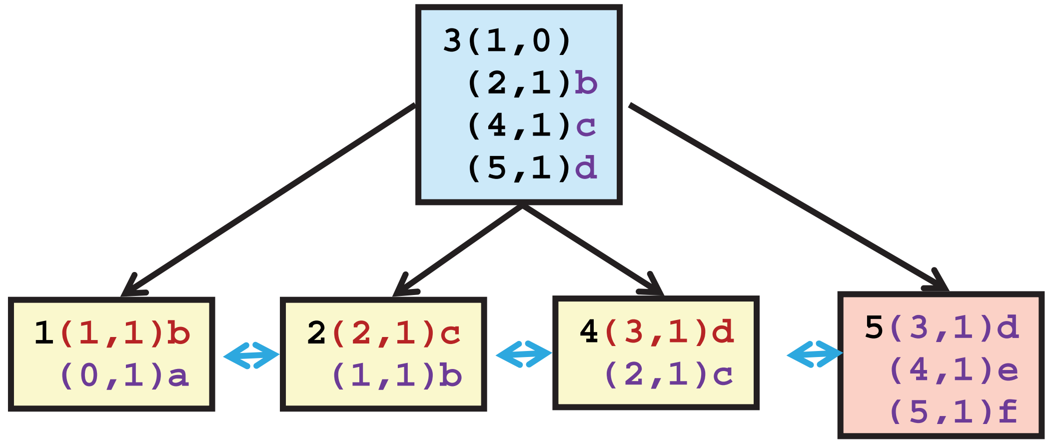


6) Repeat for line 6 :

```
postgres=# insert into t values (repeat('f',2500));
select ctid, left(s, 24) from t;
select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev, btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
INSERT 0 1
 ctid | left
--------+-------------------------
 (0,1) | aaaaaaaaaaaaaaaaaaaaaaaa
 (1,1) | bbbbbbbbbbbbbbbbbbbbbbbb
 (2,1) | cccccccccccccccccccccccc
 (3.1) | dddddddddddddddddddddd
 (4.1) | eeeeeeeeeeeeeeeeeeeeeeee
 (5,1) | ffffffffffffffffffffff
(6 rows)

 blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
 1 | l | 2 | 2512 | 3116 | 0 | 2 | 0 | 1
 2 | l | 2 | 2512 | 3116 | 1 | 4 | 0 | 1
 3 | r | 4 | 1886 | 588 | 0 | 0 | 1 | 2
 4 | l | 2 | 2512 | 3116 | 2 | 5 | 0 | 1
 5 | l | 3 | 2512 | 600 | 4 | 0 | 0 | 1
(5 rows)


 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
 1 | (1,1) | 2512 | | 20 27 00 00 62 62 | b
 2 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
(2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
 1 | (2,1) | 2512 | | 20 27 00 00 63 63 | c
 2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
(2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
 1 | (1.0) | 8 | | |
 2 | (2,1) | 2512 | | 20 27 00 00 62 62 | b
 3 | (4.1) | 2512 | | 20 27 00 00 63 63 | c
 4 | (5,1) | 2512 | | 20 27 00 00 64 64 | d
(4 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

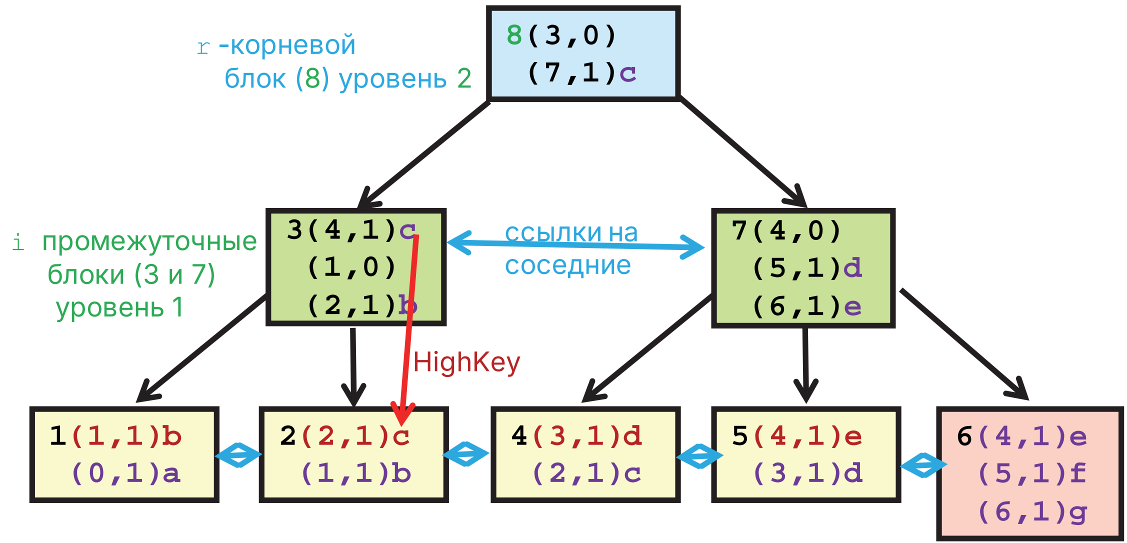
```
1 | (3.1) | 2512 | | 20 27 00 00 64 64 | d
2 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
(2 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (3.1) | 2512 | (3.1) | 20 27 00 00 64 64 | d
2 | (4.1) | 2512 | (4.1) | 20 27 00 00 65 65 | e
3 | (5,1) | 2512 | (5,1) | 20 27 00 00 66 66 | f
(3 rows)
```



7) Repeat for line 7. Three blocks will be added to the index at once , since the number of index levels will increase and the blocks will be divided into two levels:

```
postgres=# insert into t values (repeat('g',2500));
select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev, btpo_next next, btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',3);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',4);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',5);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',6);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',7);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',8);
INSERT 0 1
blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
1 | l | 2 | 2512 | 3116 | 0 | 2 | 0 | 1
2 | l | 2 | 2512 | 3116 | 1 | 4 | 0 | 1
3 | i | 3 | 1677 | 3104 | 0 | 7 | 1 | 0
4 | l | 2 | 2512 | 3116 | 2 | 5 | 0 | 1
5 | l | 2 | 2512 | 3116 | 4 | 6 | 0 | 1
6 | l | 3 | 2512 | 600 | 5 | 0 | 0 | 1
7 | i | 3 | 1677 | 3104 | 3 | 0 | 1 | 0
8 | r | 2 | 1260 | 5620 | 0 | 0 | 2 | 2
(8 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (1,1) | 2512 | | 20 27 00 00 62 62 | b
2 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
(2 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (2,1) | 2512 | | 20 27 00 00 63 63 | c
2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
(2 rows)
```

```
 o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (4.1) | 2512 | | 20 27 00 00 63 63 | c
2 | (1.0) | 8 | | |
3 | (2,1) | 2512 | | 20 27 00 00 62 62 | b
(3 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (3.1) | 2512 | | 20 27 00 00 64 64 | d
2 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
(2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (4.1) | 2512 | | 20 27 00 00 65 65 | e
2 | (3.1) | 2512 | (3.1) | 20 27 00 00 64 64 | d
(2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (4.1) | 2512 | (4.1) | 20 27 00 00 65 65 | e
2 | (5,1) | 2512 | (5,1) | 20 27 00 00 66 66 | f
3 | (6,1) | 2512 | (6,1) | 20 27 00 00 67 67 | g
(3 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (4.0) | 8 | | |
2 | (5,1) | 2512 | | 20 27 00 00 64 64 | d
3 | (6,1) | 2512 | | 20 27 00 00 65 65 | e
(3 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (3.0) | 8 | | |
2 | (7.1) | 2512 | | 20 27 00 00 63 63 | c
(2 rows)
```



Block 8 became the root block instead of 3.

Adding a new level has led to a peculiarity in block 3. It would be "more logical" to make the second entry in block 3 first. However, it comes second and this is not an error. Also, there are two pointers to block 4 from two blocks of the higher level. This is also not an error (searching for values by index is not violated).

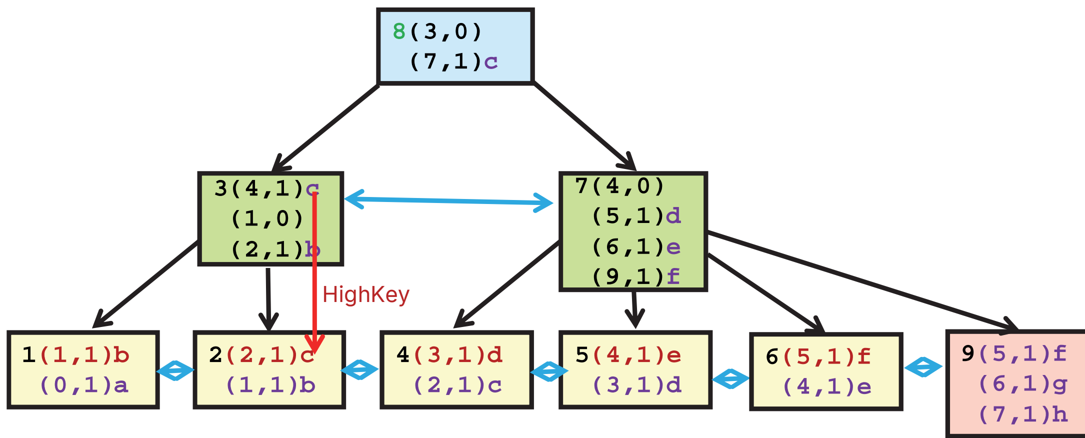
8) Repeat for line 8 :

```
postgres=# insert into t values (repeat('h',2500));
 select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev, btpo_next next, btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',2);
```

```
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',6);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',9);
 INSERT 0 1
 blkno | type | live | size | free | prev | next | level | fl
 -------+------+------+------+------+------+------+-------+----
 1 | l | 2 | 2512 | 3116 | 0 | 2 | 0 | 1
 2 | l | 2 | 2512 | 3116 | 1 | 4 | 0 | 1
 3 | i | 3 | 1677 | 3104 | 0 | 7 | 1 | 0
 4 | l | 2 | 2512 | 3116 | 2 | 5 | 0 | 1
 5 | l | 2 | 2512 | 3116 | 4 | 6 | 0 | 1
 6 | l | 2 | 2512 | 3116 | 5 | 9 | 0 | 1
 7 | i | 4 | 1886 | 588 | 3 | 0 | 1 | 0
 8 | r | 2 | 1260 | 5620 | 0 | 0 | 2 | 2
   9 | l | 3 | 2512 | 600 | 6 | 0 | 0 | 1
 (9 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+-------+---------------------+---
 1 | (1,1) | 2512 | | 20 27 00 00 62 62 | b
 2 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
 (2 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+-------+---------------------+---
 1 | (2,1) | 2512 | | 20 27 00 00 63 63 | c
 2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
 (2 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+------+-----+---
 1 | (4.1) | 2512 | | 20 27 00 00 63 63 | c
 2 | (1.0) | 8 | | |
 3 | (2,1) | 2512 | | 20 27 00 00 62 62 | b
 (3 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+-------+---------------------+---
 1 | (3.1) | 2512 | | 20 27 00 00 64 64 | d
 2 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
 (2 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+-------+---------------------+---
 1 | (4.1) | 2512 | | 20 27 00 00 65 65 | e
 2 | (3.1) | 2512 | (3.1) | 20 27 00 00 64 64 | d
 (2 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+-------+---------------------+---
 1 | (5,1) | 2512 | | 20 27 00 00 66 66 | f
 2 | (4.1) | 2512 | (4.1) | 20 27 00 00 65 65 | e
 (2 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+------+-----+---
 1 | (4.0) | 8 | | |
 2 | (5,1) | 2512 | | 20 27 00 00 64 64 | d
 3 | (6,1) | 2512 | | 20 27 00 00 65 65 | e
 4 | (9,1) | 2512 | | 20 27 00 00 66 66 | f
 (4 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+------+-----+---
 1 | (3.0) | 8 | | |
 2 | (7.1) | 2512 | | 20 27 00 00 63 63 | c
 (2 rows)

 o | ctid | itemlen | htid | data | c
 ---+-------+---------+-------+---------------------+---
 1 | (5,1) | 2512 | (5,1) | 20 27 00 00 66 66 | f
 2 | (6,1) | 2512 | (6,1) | 20 27 00 00 67 67 | g
 3 | (7.1) | 2512 | (7.1) | 20 27 00 00 68 68 | h
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

(3 rows)

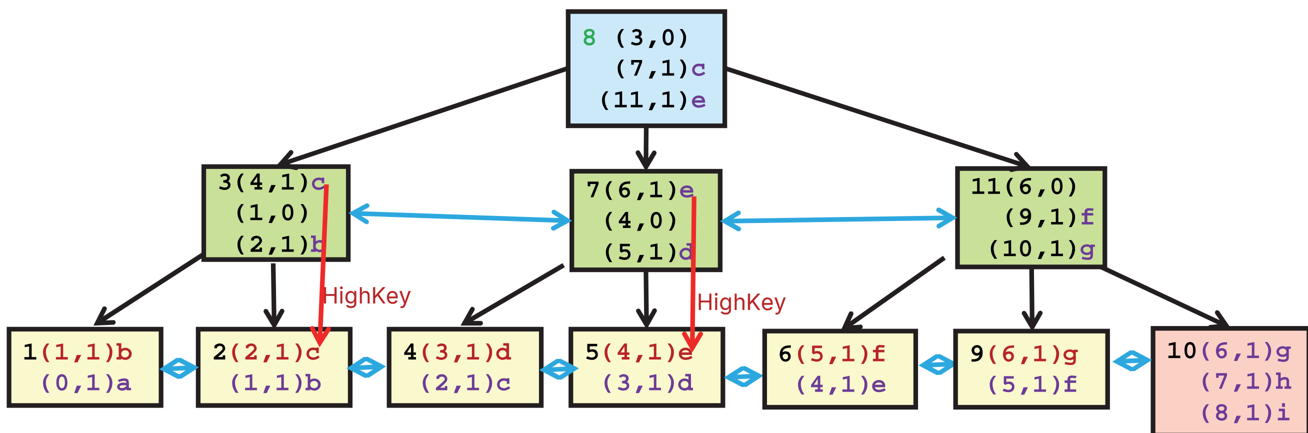


One block will be added to the index . The contents of blocks 6 and 7 have changed .

9) Repeat for line 9.

```
postgres=# insert into t values (repeat('i',2500));
 select blkno, type, live_items live, avg_item_size size, free_size free, btpo_prev prev, btpo_next next,
btpo_level level, btpo_flags fl from bt_multi_page_stats('t_idx',1,-1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',6);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',9);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',10);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',11);
 INSERT 0 1
 blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+-------+----
 1 | l | 2 | 2512 | 3116 | 0 | 2 | 0 | 1
 2 | l | 2 | 2512 | 3116 | 1 | 4 | 0 | 1
 3 | i | 3 | 1677 | 3104 | 0 | 7 | 1 | 0
 4 | l | 2 | 2512 | 3116 | 2 | 5 | 0 | 1
 5 | l | 2 | 2512 | 3116 | 4 | 6 | 0 | 1
 6 | l | 2 | 2512 | 3116 | 5 | 9 | 0 | 1
 7 | i | 3 | 1677 | 3104 | 3 | 11 | 1 | 0
 8 | r | 3 | 1677 | 3104 | 0 | 0 | 2 | 2
 9 | l | 2 | 2512 | 3116 | 6 | 10 | 0 | 1
 10 | l | 3 | 2512 | 600 | 9 | 0 | 0 | 1
  11 | i | 3 | 1677 | 3104 | 7 | 0 | 1 | 0
 (11 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
 1 | (1,1) | 2512 | | 20 27 00 00 62 62 | b
 2 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
 (2 rows)
 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
 1 | (2,1) | 2512 | | 20 27 00 00 63 63 | c
 2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
 (2 rows)
 o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
 1 | (4.1) | 2512 | | 20 27 00 00 63 63 | c
 2 | (1.0) | 8 | | |
 3 | (2,1) | 2512 | | 20 27 00 00 62 62 | b
```

```
(3 rows)
o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (3.1) | 2512 | | 20 27 00 00 64 64 | d
2 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
(2 rows)
o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (4.1) | 2512 | | 20 27 00 00 65 65 | e
2 | (3.1) | 2512 | (3.1) | 20 27 00 00 64 64 | d
(2 rows)
o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (5,1) | 2512 | | 20 27 00 00 66 66 | f
2 | (4.1) | 2512 | (4.1) | 20 27 00 00 65 65 | e
(2 rows)
o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (6,1) | 2512 | | 20 27 00 00 65 65 | e
2 | (4.0) | 8 | | |
3 | (5,1) | 2512 | | 20 27 00 00 64 64 | d
(3 rows)
o | ctid | itemlen | htid | data | c
---+--------+---------+------+-----+---
1 | (3.0) | 8 | | |
2 | (7.1) | 2512 | | 20 27 00 00 63 63 | c
3 | (11.1) | 2512 | | 20 27 00 00 65 65 | e
(3 rows)
o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (6,1) | 2512 | | 20 27 00 00 67 67 | g
2 | (5,1) | 2512 | (5,1) | 20 27 00 00 66 66 | f
(2 rows)
o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (6,1) | 2512 | (6,1) | 20 27 00 00 67 67 | g
2 | (7.1) | 2512 | (7.1) | 20 27 00 00 68 68 | h
3 | (8,1) | 2512 | (8,1) | 20 27 00 00 69 69 | i
(3 rows)
o | ctid | itemlen | htid | data | c
---+--------+---------+------+-----+---
1 | (6.0) | 8 | | |
2 | (9,1) | 2512 | | 20 27 00 00 66 66 | f
3 | (10,1) | 2512 | | 20 27 00 00 67 67 | g
(3 rows)
```



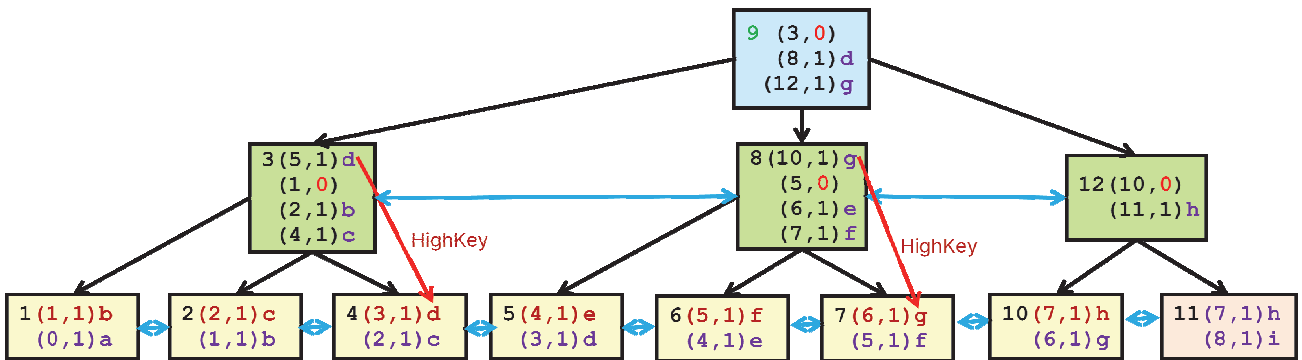
10) See how the index structure changes after rebuilding with the REINDEX command:

```
postgres=# reindex index t_idx;
select blkno blk, type, live_items live, btpo_prev prev, btpo_next next, btpo_level level from
bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',3);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',4);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',5);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13
for 2))::integer,0)) c from bt_page_items('t_idx',6);
```

```
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',9);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',10);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',11);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data, chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items('t_idx',12);
 REINDEX
 blk | type | live | prev | next | level
-----+------+------+------+------+-------
 1 | l | 2 | 0 | 2 | 0
 2 | l | 2 | 1 | 4 | 0
 3 | i | 4 | 0 | 8 | 1
 4 | l | 2 | 2 | 5 | 0
 5 | l | 2 | 4 | 6 | 0
 6 | l | 2 | 5 | 7 | 0
 7 | l | 2 | 6 | 10 | 0
 8 | i | 4 | 3 | 12 | 1
 9 | r | 3 | 0 | 0 | 2
 10 | l | 2 | 7 | 11 | 0
 11 | l | 2 | 10 | 0 | 0
 12 | i | 2 | 8 | 0 | 1
 (12 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+----------------------+---
 1 | (1,1) | 2512 | | 20 27 00 00 62 62 | b
 2 | (0,1) | 2512 | (0,1) | 20 27 00 00 61 61 | a
 (2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+----------------------+---
 1 | (2,1) | 2512 | | 20 27 00 00 63 63 | c
 2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
 (2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
 1 | (5,1) | 2512 | | 20 27 00 00 64 64 | d
 2 | (1.0) | 8 | | |
 3 | (2,1) | 2512 | | 20 27 00 00 62 62 | b
 4 | (4.1) | 2512 | | 20 27 00 00 63 63 | c
 (4 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+----------------------+---
 1 | (3.1) | 2512 | | 20 27 00 00 64 64 | d
 2 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | c
 (2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+----------------------+---
 1 | (4.1) | 2512 | | 20 27 00 00 65 65 | e
 2 | (3.1) | 2512 | (3.1) | 20 27 00 00 64 64 | d
 (2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+----------------------+---
 1 | (5,1) | 2512 | | 20 27 00 00 66 66 | f
 2 | (4.1) | 2512 | (4.1) | 20 27 00 00 65 65 | e
 (2 rows)

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+----------------------+---
 1 | (6,1) | 2512 | | 20 27 00 00 67 67 | g
 2 | (5,1) | 2512 | (5,1) | 20 27 00 00 66 66 | f
 (2 rows)

 o | ctid | itemlen | htid | data | c
---+--------+---------+------+-----+---
 1 | (10,1) | 2512 | | 20 27 00 00 67 67 | g
 2 | (5.0) | 8 | | |
 3 | (6,1) | 2512 | | 20 27 00 00 65 65 | e
 4 | (7.1) | 2512 | | 20 27 00 00 66 66 | f
 (4 rows)

 o | ctid | itemlen | htid | data | c
---+--------+---------+------+-----+---
 1 | (3.0) | 8 | | |
```

tantor

```
2 | (8,1) | 2512 | | 20 27 00 00 64 64 | d
3 | (12.1) | 2512 | | 20 27 00 00 67 67 | g
(3 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (7.1) | 2512 | | 20 27 00 00 68 68 | h
2 | (6,1) | 2512 | (6,1) | 20 27 00 00 67 67 | g
(2 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+---------------------+---
1 | (7.1) | 2512 | (7.1) | 20 27 00 00 68 68 | h
2 | (8,1) | 2512 | (8,1) | 20 27 00 00 69 69 | i
(2 rows)

o | ctid | itemlen | htid | data | c
---+--------+---------+------+-----+---
1 | (10.0) | 8 | | |
2 | (11.1) | 2512 | | 20 27 00 00 68 68 | h
(2 rows)
```



11) See that after deleting all rows in the table, the number of levels in the index does not change, and fasroot points to the leaf block:

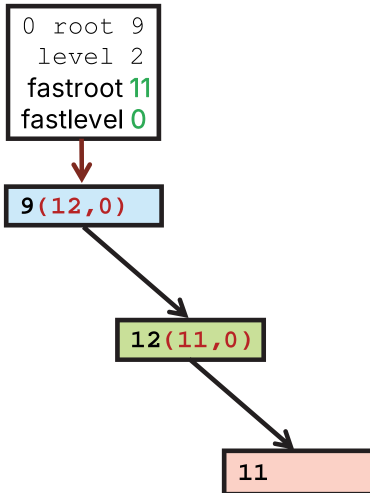
```
postgres=# delete from t;
select root, level, fastroot, fastlevel,
last_cleanup_num_delpages, allequalimage from bt_metap('t_idx');
vacuum t;
select version, root, level, fastroot, fastlevel,
last_cleanup_num_delpages delpages,
last_cleanup_num_tuples tuples, allequalimage from bt_metap('t_idx');
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',9);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx', 10 );
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',11);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',12);
DELETE 9
version | root | level | fastroot | fastlevel | delpages | tuples | allequalimage
---------+------+-------+----------+-----------+----------+--------+---------------
4 | 9 | 2 | 9 | 2 | 0 | -1 | t
(1 row)
VACUUM
version | root | level | fastroot | fastlevel | delpages | tuples | allequalimage
---------+------+-------+----------+-----------+----------+--------+---------------
4 | 9 | 2 | 11 | 0 | 9 | -1 | t
(1 row)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
blk | type | live | prev | next | level
-----+------+-------+------+------+-------
1 | d | 0 | 0 | 2 | 0
2 | d | 0 | 0 | 4 | 0
3 | D | 0 | 0 | 8 | 1
4 | d | 0 | 0 | 5 | 0
5 | d | 0 | 0 | 6 | 0
6 | d | 0 | 0 | 7 | 0
7 | d | 0 | 0 | 10 | 0
8 | D | 0 | 0 | 12 | 1
  9 | r | 1 | 0 | 0 | 2
10 | d | 0 | 0 | 11 | 0
  11 | l | 0 | 0 | 0 | 0
  12 | i | 1 | 0 | 0 | 1
(12 rows)

o | ctid | itemlen | htid | data
---+--------+---------+------+------
1 | (12.0) | 8 | |
(1 row)

NOTICE: page from block 10 is deleted
o | ctid | itemlen | htid | data
---+------+---------+------+------
(0 rows)

o | ctid | itemlen | htid | data
---+------+---------+------+------
(0 rows)

o | ctid | itemlen | htid | data
---+--------+---------+------+------
1 | (11.0) | 8 | |
(1 row)
```

After deleting all the lines, fastroot did not change and the blocks were not excluded from their index structure.

After vacuuming, 9 blocks were excluded from the index structure, 3 left right blocks from each level. `fastroot` points to leaf block 11 . This is the block from which the index search will begin.



Part 10. Cleaning index blocks during its scanning

When deleting a row in a table block, the deleting transaction places its number in the row header in the xmax field . In this case, the commit flag is not set. The next time the row is accessed, another process usually checks the transaction status in the "XLOG Ctl" buffer (obsolete name "CLOG") and sometimes the multi-transaction status buffer " Shared MultiXact State ". Based on the results of the check, hint bits are set in the table row header indicating that the transaction that deleted the row was committed or cancelled. In this case, the row can be referenced by records of indexes created on the table. If the table row was not accessed through an index, the index is not updated in any way, as are all other indexes created on the table.

However, if a process used an index scan to access a row and the process detects that the row (or the chain of rows referenced by the index entry) has been deleted and has gone beyond the base horizon, then the process will set the LP_DEAD (known dead, killed tuple) hint bit in the index entry of the leaf page in lp_flags .
LP_DEAD is set in the entry of the index through which the row was accessed.
LP_DEAD is set regardless of whether the row (or chain of rows referenced by the index entry) was deleted previously or by the same process.
Inserting this hint bit is called in-page cleaning in indexes.
In this part of the practice you will see how much this affects the query execution time.

1) Create a table, insert 1 million rows, delete most of the table rows and collect statistics for the planner:

```
postgres=# drop table if exists t;
create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
insert into t select i,'text number '||i from generate_series(1,1000000) i;
analyze t;
delete from t where id between 501 and 799500;
analyze t;
DROP TABLE
CREATE TABLE
INSERT 0 1000000
ANALYZE
DELETE 799000
ANALYZE
```

The collection of statistics does not play a role, but it illustrates that the planner is not misled.

2) Look at the data from the index blocks:

```
postgres=# select blkno, type, live_items, dead_items , avg_item_size size,
free_size free, btpo_prev prev, btpo_next next, btpo_level level, btpo_flags fl
from bt_multi_page_stats('t_pkey',1,-1) where type='l' limit 5;
select dead_items , count(*) from bt_multi_page_stats('t_pkey',1,-1) group by
dead_items ;
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
blkno | type | live_items | dead_items | size | free | prev | next | level | fl
-------+------+------------+------+------+------+------+------+-------+----
1 | l | 367 | 0 | 16 | 808 | 0 | 2 | 0 | 1
2 | l | 367 | 0 | 16 | 808 | 1 | 4 | 0 | 1
4 | l | 367 | 0 | 16 | 808 | 2 | 5 | 0 | 1
5 | l | 367 | 0 | 16 | 808 | 4 | 6 | 0 | 1
6 | l | 367 | 0 | 16 | 808 | 5 | 7 | 0 | 1
(5 rows)

dead_items | count
------------+-------
0 | 2744
(1 row)

type | count
------+-------
r | 1
l | 2733
i | 10
(3 rows)
```

IN index No LP_DEAD (column `dead_items`) records .
The number of leaf blocks is 2733, blocks at the intermediate level are 10.

2) To mark index records as dead_items you need to access the lines using an index (index access method Index * Scan). The explain analyze command is suitable for this . It is useful to see how many blocks will be read, for this the buffers option is used . To make the command execution time realistic, the timing option is used off . Execute command :

```
postgres=# explain (buffers,timing off) select * from t where id between 1 and
800000;
 QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22)
 Index Cond: ((id >= 1) AND (id <= 800000))
  Planning:
    Buffers: shared hit= 27
 (4 rows)
```

In the Planning section : Buffers line : shows how many blocks were read when constructing the palan.

Do it command :

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
 QUERY PLAN
--------------------------------------------------------------------------------
  Index Scan using t_pkey on t (cost=0.42..220.58 rows=2008 width=22) (actual rows=1000 loops=1)
 Index Cond: ((id >= 1) AND (id <= 800000))
    Buffers: shared hit= 9501
 Planning:
    Buffers: shared hit= 8
 Planning Time: 0.173 ms
 Execution Time: 187.850 ms
 (7 rows )
```

The number of buffers read to build the plan was reduced from 27 to 8 because the data from the system catalog tables was cached in the local memory of the server process in cache of system catalog tables (the memory structure is called CatalogCache).
Hint bits are set and dirty anyway. If you wait a bit before executing the explain command , the result will be as follows (the number of dirtied blocks will be *arbitrary*):

```
                                          QUERY PLAN
-------------------------------------------------------------------------------------------------
  Index Scan using t_pkey on t (cost=0.42..220.58 rows=2008 width=22) (actual rows=1000 loops=1)
 Index Cond: ((id >= 1) AND (id <= 800000))
 Buffers: shared hit=9501 dirtied= 7306
 Planning:
 Buffers: shared hit=20 dirtied=1
 Planning Time: 0.167 ms
  Execution Time: 223.348 ms
 (7 rows)
```

An index scan was used. Table access methods (Seq Scan) and Bitmap Index Scan does not clear index blocks, only clears index scans.
The number of blocks read is 9501. Most of them are related to the table (there are only 2744 blocks in the index, and only a part of the leaf blocks were read) - the query checked the table to see if a row in the table block was deleted. If the index link led to a deleted row in the table, the index record was marked as LP_DEAD.
Query execution time is 187 milliseconds.
After this query was executed, the records in the index blocks were marked with the LP_DEAD (known dead) hint bit . This bit allows this index record to be ignored without having to double-check whether the record should really be ignored by accessing the table block.

3) Check if dead_items have appeared entries:

```
postgres=# select blkno, type, live_items, dead_items , avg_item_size size,
free_size free, btpo_prev prev, btpo_next next, btpo_level level, btpo_flags fl
from bt_multi_page_stats('t_pkey',1,-1) where type='l' limit 5;
 select dead_items , count(*) from bt_multi_page_stats('t_pkey',1,-1) group by
dead_items ;
 select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
 blkno | type | live_items | dead_items | size | free | prev | next | level | fl
-------+------+------------+------------+------+------+------+------+-------+----
 1 | l | 367 | 0 | 16 | 808 | 0 | 2 | 0 | 1
 2 | l | 135 | 232 | 16 | 808 | 1 | 4 | 0 | 65
 4 | l | 1 | 366 | 16 | 808 | 2 | 5 | 0 | 65
 5 | l | 1 | 366 | 16 | 808 | 4 | 6 | 0 | 65
 6 | l | 1 | 366 | 16 | 808 | 5 | 7 | 0 | 65
 (5 rows)

  dead_items | count
------------+-------
 156 | 1
 232 | 1
 366 | 2182
 0 | 560
 (4 rows)

 type | count
------+-------
 r | 1
 l | 2733
 i | 10
 (3 rows)
```

the LP_DEAD (known dead) hint bit .
There were no blocks excluded from the index structure.

4) Perform request more once :

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;

 QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using t_pkey on t (cost=0.42..220.58 rows=2008 width=22) (actual rows=1000 loops=1)
 Index Cond: ((id >= 1) AND (id <= 800000))
 Buffers: shared hit= 2196
 Planning:
 Buffers: shared hit= 8
 Planning Time: 0.114 ms
 Execution Time: 2.522 ms
 (7 rows )
```

Query execution time has been reduced from 187 to 2 milliseconds.
The query took 187 milliseconds to execute because it was making changes to block records and because it was reading table blocks. The LP_DEAD flag is set once on the master.
On a physical replica, index records are not marked with the LP_DEAD flag because the replica cannot change the contents of local database object blocks.
Changes to LP_DEAD hint bits are not logged or propagated to the replica.
However, **after deleting or changing a large number of rows in a table, queries that would normally run quickly once (on the master) may take hundreds of times (in the example, 187/2=93 times) longer to execute.**
On **the replica, queries will be executed more slowly all the time until autovacuum** on the master.
If the autovacuum manages to access the table before the request, then the autovacuum will experience a delay.
A significant delay in query execution may be unexpected for the client application. To prevent this from happening, it is worth vacuuming the table after mass deletion or modification of rows.

When executing the query repeatedly, much fewer blocks are read. If you repeat the query, the number of blocks read will be the same: 2196 .

5) Vacuum the table and its index:

```
postgres =# vacuum analyze t ;
VACUUM
```

6) View statistics on index pages:

```
postgres =# select blkno, type, live_items, dead_items , avg_item_size size,
free_size free, btpo_prev prev, btpo_next next, btpo_level level, btpo_flags fl
from bt_multi_page_stats('t_pkey',1,-1) where type='l' limit 5;
select dead_items , count(*) from bt_multi_page_stats('t_pkey',1,-1) group by
dead_items ;
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
blkno | type | live_items | dead_items | size | free | prev | next | level | fl
-------+------+------------+------+------+------+------+------+-------+----
1 | l | 367 | 0 | 16 | 808 | 0 | 2 | 0 | 1
2 | l | 135 | 0 | 16 | 5448 | 1 | 286 | 0 | 1
286 | l | 1 | 0 | 16 | 8128 | 2 | 2194 | 0 | 1
2194 | l | 211 | 0 | 16 | 3928 | 286 | 2195 | 0 | 1
2195 | l | 367 | 0 | 16 | 808 | 2194 | 2196 | 0 | 1
(5 rows)

dead_items | count
------------+-------
0 | 2744
(1 row)
type | count
------+-------
 D | 6
r | 1
l | 552
i | 4
 d | 2181
(5 rows)
```

marked with the LP_DEAD flag were vacuumed out of the index blocks.
Block types: d - deleted sheet (deleted leaf) block, D - deleted internal (deleted internal) block.
Blocks in which **all records were marked as deleted (during vacuuming or earlier during Index Scan) are excluded from the index structure during vacuuming** . Such blocks are not scanned by queries, but they are read during vacuuming, since indexes do not have a visibility map and all blocks are read.

7) Perform request :

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
QUERY PLAN
------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit= 15
Planning:
Buffers: shared hit= 16
Planning Time: 0.108 ms
Execution Time: 0.284 ms
(7 rows )
```

Vacuuming cleared index blocks. The number of blocks read by the query decreased from 2196 to 15 . The value of " `shared hit = 16` " is the number of buffers read to create the plan and they contain mainly blocks of the system catalog tables. Why in the section " `Planning :` " the number of

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

buffers increased from 8 to 16 ? Because statistics were collected with the **vacuum command analyze** and the planner calculated it to create a plan.

8) Look size index :

```
postgres=# select pg_indexes_size('t');
```

Vacuuming did not reduce the size of the index because there are blocks with records at the end of the index file and the size of the index file did not decrease.

9) Rebuild the index and check if the index size decreases:

```
postgres=# reindex table t;
analyze t;
select pg_indexes_size('t');
pg_indexes_size
-----------------
        22487040
(1 row)


REINDEX
ANALYZE
pg_indexes_size
-----------------
         4538368
(1 row )
```

After rebuilding the index, the size of the index file has decreased.

10) Complete request :

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;
QUERY PLAN
-------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42.. 132.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=11 read=2
Planning:
Buffers: shared hit=17 read=s4
Planning Time: 0.201 ms
Execution Time : 0.346 ms
(7 rows )
```

The index size has decreased by 5 times. The cost of the same plan has decreased from 220.58 to 132.56 . However, the execution time has not improved. Physical reads of two blocks do not affect the query execution time. If you execute the query several times before and after rebuilding the index, you can see that the average query execution time is the same, which means that rebuilding the index did not affect the query execution time.

Why didn't it affect? Index files may include blocks that are excluded from the index structure. Blocks excluded from the index structure are counted in the index size. The last two queries with different costs but the same execution time most likely scanned the same number of leaf blocks of the index, which determines the query execution time.

When vacuuming, all index file blocks are read in order from the beginning of the file to the end of each file. Reducing the size of index files speeds up the operation of autovacuum and reduces the number of empty index blocks that autovacuum is forced to load into the buffer cache.

After deleting a large number of rows, it is worth rebuilding the indexes on the table if you need to speed up the work of autovacuum with this table.

And when deleting a large number of rows at once, it is more optimal to delete indexes before deleting rows, then delete rows, then create indexes, then perform a vacuum and collect statistics. In the absence of indexes, deleting rows will be performed significantly (by orders of magnitude) faster.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Part 11. Slow query execution due to lack of index block cleaning on the replica

1) Create physical replica :

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/backup/1 -m immediate
psql -c "select pg_drop_replication_slot('replica1');"
rm -rf /var/lib/postgresql/backup/1
pg_basebackup -D /var/lib/postgresql/backup/1 -P -R -c fast -C --slot=replica1
echo "port=5433" >> /var/lib/postgresql/backup/1/postgresql.auto.conf
pg_ctl start -D /var/lib/postgresql/backup/1

ERROR: replication slot "replica1" does not exist
...
server started
postgres @ tantor :~$
```

`server` appears `started` , then the replica runs on port 5433.
Diagnostic messages will be displayed in the terminal; to return the terminal, simply press **< ENTER > on the keyboard .**

2) Launch psql and run the commands on the master instance:

```
postgres@tantor:~$ psql
postgres=# drop table if exists t;
create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
insert into t select i,'text number '||i from generate_series(1,1000000) i;
analyze t;
delete from t where id between 501 and 799500;
analyze t;
DROP TABLE
CREATE TABLE
INSERT 0 1000000
ANALYZE
DELETE 799000
ANALYZE
```

3) In a new terminal window, run psql and connect to the replica:

```
postgres @ tantor :~$ psql - p 5433
```

4) Run the query on the replica several times:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
explain (analyze,buffers,timing off) select * from t where id between 1 and
800000;
QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=7317
Planning:
Buffers: shared hit=63 read=16
Planning Time: 4.032 ms
 Execution Time: 341.255 ms
(7 rows)

QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=7317
Planning:
Buffers: shared hit=8
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
Planning Time: 0.104 ms
 Execution Time: 345.192 ms
(7 rows )
```

Each time before the first request on the master, the request on the replica takes a long time: ~ 345 milliseconds.

5) Do it request on masters :

postgres=# **explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;**

```
QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=9501 dirtied= 2636
Planning:
Buffers: shared hit=26 dirtied=1
Planning Time: 0.190 ms
Execution Time : 189.604 ms
(7 rows )
```

The first request on the master was executed in `189.604 ms` , the second and subsequent requests are executed quickly, in ~ 2 ms. Exactly the same as in the absence of a replica, as in the previous part of the practice.

6) After the query is executed on the master for the first time, queries on the replica will be executed in ~ 120 ms. Run on replica two times request :

postgres=# **explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;**
explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;

```
QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=7317
Planning:
Buffers: shared hit=8
Planning Time: 0.086 ms
 Execution Time: 122.248 ms
(7 rows)

QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=7317
Planning:
Buffers: shared hit=8
Planning Time: 0.132 ms
Execution Time: 116.896 ms
(7 rows )
```

The first request on the master is completed in ~ 190 ms, subsequent ones in ~ 2 ms.
After the first query on the master is executed, queries on the replica will be executed a little faster: in ~ 120 ms. The difference is due to the fact that changes are made to the index blocks on the master: the LP_DEAD bits are set. No changes are made to the index blocks on the replica. Most of the query time is spent on accessing the table blocks.
Before vacuuming, queries on replicas will take longer to execute than even on the master.

7) Run a query on the master and vacuum the table t:

postgres=# **explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;**
vacuum analyze t;

```
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;

QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=2196
Planning:
Buffers: shared hit=8
Planning Time: 0.125 ms
Execution Time: 2.234 ms
(7 rows)
VACUUM
type | count
------+-------
D | 6
r | 1
l | 552
i | 4
d | 2181
(5 rows)
QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=15
Planning:
Buffers: shared hit=16 dirtied=1
Planning Time: 0.123 ms
Execution Time: 0.286 ms
(7 rows )
```

The request on the master is needed to ensure that the second request on the master will be completed within 2ms.

After vacuuming, most blocks were excluded from the index structure and marked as having type = `d` or `D` .

After vacuuming, the request will start to be executed an order of magnitude faster: in 0.286 ms .

8) Run the following query twice on the replica:

```
postgres=# explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;
explain (analyze,buffers,timing off) select * from t where id between 1 and 800000;
select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=15
Planning:
Buffers: shared hit=19
Planning Time: 0.164 ms
Execution Time: 0.291 ms
(7 rows)

QUERY PLAN
-----------------------------------------------------------------------------------------------
Index Scan using t_pkey on t (cost=0.42..220.56 rows=2007 width=22) (actual rows=1000 loops=1)
Index Cond: ((id >= 1) AND (id <= 800000))
Buffers: shared hit=15
Planning:
Buffers: shared hit=8
Planning Time: 0.071 ms
 Execution Time: 0.307 ms
(7 rows)

type | count
------+-------
D | 6
r | 1
l | 552
 i | 4
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
 d | 2181
(5 rows )
```

After a vacuum on the master, the query execution time on the replica is the same as on the master: ~ 0.3ms .

9) Delete the physical replica to free up disk space:

```
postgres@tantor:~$ pg_ctl stop -D /var/lib/postgresql/backup/1 -m immediate
psql -c "select pg_drop_replication_slot('replica1');"
rm -rf /var/lib/postgresql/backup/1
```

The execution time in the example has decreased by a thousand times : from 345 ms to 0.3 ms .
Timely execution of autovacuum on the master affects the execution time of queries on replicas. Before the execution of autovacuum or vacuuming of a table in which many rows were deleted, queries on the replica are executed significantly slower than they could be.

Part 12. Determining the number of deleted lines

How to determine the number of deleted rows before vacuuming? Analysis and autoanalysis do not collect such statistics on indexes.

1) Run the commands on the master instance:

```
postgres@tantor:~$ psql
postgres=# drop table if exists t;
create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
insert into t select i,'text number '||i from generate_series(1,1000000) i;
analyze t;
delete from t where id between 501 and 799500;
analyze t;
DROP TABLE
CREATE TABLE
INSERT 0 1000000
ANALYZE
DELETE 799000
ANALYZE
```

These commands recreate table t and create a large number of deleted rows.

2) Function `bt_multi_page_stats ( ' t_pkey ' , 1, -1 )` extensions `pageinspect` and `pgstatindex ( ' t_pkey ')` function `pgstattuple` extensions will show the removed blocks only after vacuuming. Example of executing functions before and after vacuuming :

```
postgres =# create extension pgstattuple ;
CREATE EXTENSION
postgres=# select tree_level levels, index_size, internal_pages i_pages, leaf_pages
leaf_pages, empty_pages e, deleted_pages , avg_leaf_density dense, leaf_fragmentation
frag FROM pgstatindex('t_pkey');

levels | index_size | i_pages | leaf_pages | e | deleted_pages | dense | frag
--------+-----------+---------+------+---+---------------+-------+------
 2 | 22487040 | 11 | 2733 | 0 | 0 | 90.06 | 0
 (1 row)

postgres=# select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
type | count
-----------+-------
r |      1
l |   2733
```

```
i |    10
(3 rows)

postgres=# vacuum t;
VACUUM
postgres=# select tree_level levels, index_size, internal_pages i_pages, leaf_pages
leaf_pages, empty_pages e, deleted_pages deleted , avg_leaf_density dense, leaf
_fragmentation frag FROM pgstatindex('t_pkey');
 levels | index_size | i_pages | leaf_pages | e | deleted_pages | dense | frag
 --------+-----------+---------+------+---+---------------+-------+------
 2 | 22487040 | 5 | 552 | 0 | 2187 | 89.63 | 0
 (1 row)

postgres=# select type, count(*) from bt_multi_page_stats('t_pkey',1,-1) group by type;
type | count
-----+------
D |     6
r |     1
l |   552
i |     4d
|  2181
(5 rows)
```

3) Rows in the index will be marked as deleted only after the first query with the `Index Scan access method` on the master. Such queries may not access all deleted rows, are labor-intensive, and may not be present on the master, since they have most likely been transferred to replicas. An example after the query, if it is executed after the commands in the first paragraph of this part of the practice (these queries do not have to be executed):

```
 postgres=# select sum(live_items), sum(dead_items) from
bt_multi_page_stats('t_pkey',1,-1);
 sum | sum
 ---------+-----
 1005484 | 0
 (1 row)

 postgres=# explain (analyze,buffers,timing off) select * from t where id between
1 and 800000;
 ...
 postgres=# select sum(live_items) live, sum(dead_items) deleted from
bt_multi_page_stats('t_pkey',1,-1);
 live | deleted
 --------+--------
 206484 | 799000
 (1 row)
```

4) How to determine the number of deleted rows before vacuuming and without querying the table on the master? This can be done indirectly using the statistics of the number of deleted rows in the table:

```
 postgres=# drop table if exists t;
 create table t(id int4 primary key, val text not null) with
(autovacuum_enabled=off);
 insert into t select i,'text number '||i from generate_series(1,1000000) i;
 delete from t where id between 501 and 799500;
 DROP TABLE
 CREATE TABLE
 INSERT 0 1000000
 ANALYZE
 DELETE 799000
 postgres=# select relname, n_tup_ins, n_tup_upd, n_tup_del, n_live_tup,
n_dead_tup from pg_stat_all_tables where relname='t';
 relname | n_tup_ins | n_tup_upd | n_tup_del | n_live_tup | n_dead_tup
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
---------+-----------+-----------+-----------+------------+------------
 t | 1000000 | 0 | 799000 | 201000 | 799000
(1 row)
```

Working statistics are collected by server processes and do not require any analysis or vacuum. This part of the practice **is** needed to remember the importance of the **n_dead_tup** column of the operational **statistics** returned **by the pg_stat_all_tables** view **.**
pgstattuple extension functions have promising names and are used to detect table bloat , but for the most part the functions in this extension are useless.

Part 13. Search by btree index structure

In Part 9 of this practice, we looked at the index structure and its graphical display. To remember how to draw the index structure based on the query results, you should draw the structure yourself. Drawing the structure helps you understand (visually evaluate) how labor-intensive the index search is for server processes.

1) If you have time to complete practical tasks, you can try to draw the structure of the following index:

```
drop table if exists t;
create table t(s text storage plain) with (autovacuum_enabled=off,
fillfactor=10);
create index t_idx on t(s) with (deduplicate_items = off);
insert into t values (repeat('a',1800));
insert into t values (repeat('b',1800));
insert into t values (repeat('c',1800));
insert into t values (repeat('d',1800));
insert into t values (repeat('e',1800));
insert into t values (repeat('f',1800));
insert into t values (repeat('g',1800));
insert into t values (repeat('h',1800));
insert into t values (repeat('i',1800));
insert into t values (repeat('j',1800));
insert into t values (repeat('k',1800));
insert into t values (repeat('l',1800));
insert into t values (repeat('m',1800));
insert into t values (repeat('n',1800));
insert into t values (repeat('o',1800));
insert into t values (repeat('p',1800));
insert into t values (repeat('q',1800));

select blkno, type, live_items live, avg_item_size size, free_size free,
btpo_prev prev, btpo_next next, btpo_level level, btpo_flags fl from
bt_multi_page_stats('t_idx',1,-1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',1);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',2);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',3);
select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',4);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',5);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',6);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',7);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',8);
 select itemoffset o, ctid, itemlen, htid, left(data::text,18) data,
chr(nullif(('0x0'||substring(data from 13 for 2))::integer,0)) c from
bt_page_items('t_idx',9);
```

Based on the result of the first query, we can draw a skeleton of the structure (index blocks):

```
 blkno | type | live | size | free | prev | next | level | fl
-------+------+------+------+------+------+------+-------+----
1 | l | 4 | 1816 | 868 | 0 | 2 | 0 | 1
2 | l | 4 | 1816 | 868 | 1 | 4 | 0 | 1
3 | i | 4 | 1364 | 2676 | 0 | 8 | 1 | 0
4 | l | 4 | 1816 | 868 | 2 | 5 | 0 | 1
5 | l | 4 | 1816 | 868 | 4 | 6 | 0 | 1
6 | l | 4 | 1816 | 868 | 5 | 7 | 0 | 1
7 | l | 2 | 1816 | 4508 | 6 | 0 | 0 | 1
8 | i | 3 | 1213 | 4496 | 3 | 0 | 1 | 0
9 | r | 2 | 912 | 6316 | 0 | 0 | 2 | 2
(9 rows)
```

It is worth starting with the root block (type = r), then marking the levels (level) and drawing the block rectangles. Number the blocks (blkno) for each level, taking into account which block is to the left (prev) and which is to the right (next).
The index has three levels, 9 blocks. At the intermediate level there are 2 blocks, at the leaf level there are 6 blocks.

Contents of each block:

```
 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (3.1) | 1816 | | 30 1 c 00 00 64 64 | d
2 | (0,1) | 1816 | (0,1) | 30 1 c 00 00 61 61 | a
3 | (1,1) | 1816 | (1,1) | 30 1 c 00 00 62 62 | b
4 | (2,1) | 1816 | (2,1) | 30 1 c 00 00 63 63 | c
(4 rows )

 o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (6,1) | 1816 | | 30 1c 00 00 67 67 | g
2 | (3.1) | 1816 | (3.1) | 30 1c 00 00 64 64 | d
3 | (4.1) | 1816 | (4.1) | 30 1c 00 00 65 65 | e
4 | (5,1) | 1816 | (5,1) | 30 1c 00 00 66 66 | f
(4 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (5,1) | 1816 | | 30 1c 00 00 6a 6a | j
2 | (1, 0 ) | 8 | | |
3 | (2,1) | 1816 | | 30 1c 00 00 64 64 | d
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

tantor

```
4 | (4.1) | 1816 | | 30 1c 00 00 67 67 | g
(4 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+-------+--------------------+---
1 | (9,1) | 1816 | | 30 1c 00 00 6a 6a | j
2 | (6,1) | 1816 | (6,1) | 30 1c 00 00 67 67 | g
3 | (7.1) | 1816 | (7.1) | 30 1c 00 00 68 68 | h
4 | (8,1) | 1816 | (8,1) | 30 1c 00 00 69 69 | i
(4 rows)

o | ctid | itemlen | htid | data | c
---+--------+---------+--------+-----+---
1 | (12.1) | 1816 | | 30 1c 00 00 6d 6d | m
2 | (9,1) | 1816 | (9,1) | 30 1c 00 00 6a 6a | j
3 | (10,1) | 1816 | (10,1) | 30 1c 00 00 6b 6b | k
4 | (11.1) | 1816 | (11.1) | 30 1c 00 00 6c 6c | l
(4 rows)

o | ctid | itemlen | htid | data | c
---+--------+---------+--------+-----+---
1 | (15.1) | 1816 | | 30 1c 00 00 70 70 | p
2 | (12.1) | 1816 | (12.1) | 30 1c 00 00 6d 6d | m
3 | (13.1) | 1816 | (13.1) | 30 1c 00 00 6e 6e | n
4 | (14.1) | 1816 | (14.1) | 30 1c 00 00 6f 6f | o
(4 rows)

o | ctid | itemlen | htid | data | c
---+--------+---------+--------+-----+---
1 | (15.1) | 1816 | (15.1) | 30 1c 00 00 70 70 | p
2 | (16.1) | 1816 | (16.1) | 30 1c 00 00 71 71 | q
(2 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (5, 0 ) | 8 | | |
2 | (6,1) | 1816 | | 30 1c 00 00 6d 6d | m
3 | (7.1) | 1816 | | 30 1c 00 00 70 70 | p
(3 rows)

o | ctid | itemlen | htid | data | c
---+-------+---------+------+-----+---
1 | (3, 0 ) | 8 | | |
2 | (8,1) | 1816 | | 30 1c 00 00 6a 6a | j
(2 rows)
```

Connections between blocks (designated arrows) are drawn By column ctid. In leaf blocks, ctid points to the row in the table on which the index is created.

In internal blocks and the root block, the first number in ctid points to the index block number, and the second number is one or zero . If one, then the data field stores the minimum value that is present in the child leaf block. If zero `ctid=(N, 0 )` , then the `data field` is empty (**treated as "minus infinity", i.e. the boundary is unknown**) and this link leads to the "leftmost" child block.

`htid` field adds "redundancy": duplicates the `ctid field value in leaf blocks` . Differences may occur during the process of making changes to index records and are used to identify that records are in the process of being changed without introducing locks (arbitration). The btree index structure is optimized for parallel operation of processes with a minimum of locks, while guaranteeing consistency.

If you cannot draw the index structure, you can repeat (this time more meaningfully) part 9 of the practice.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

2) The index in this part of the practice differs from the example in Part 9 of this practice in that the leaf block can hold up to three references to table rows, not one reference. And this index allows us to make sure that the inner blocks store the "leftmost" (smaller) values that are present in the second entry (the first entry is HighKey) of all leaf blocks except the rightmost block. The process that searches for the values `e, f, h, i, n, o, q` will focus on the range ("fork") between the value "to the left" of the one being searched for and HighKey, since these values are not present in the root and inner (intermediate) blocks of the index.

After the index structure is drawn, try to find the values e, f, h, i, n, o, q by the structure. Due to the simplicity of the index structure, you will visually evaluate how processes perform searches by btree indexes and the "labor intensity" of the search , imagining that the operations of comparison, selection of records, and traversal of a link to a neighboring block use the resources of the processor core.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Practice for Chapter 8

Part 1. TOAST

1) TOAST supports variable-length data types, called `varlena` (`pg_type.typlen=-1`). Fixed-length fields cannot be stored outside the table block, because there is no code written for these data types to implement storage outside the table block (in a TOAST table).
Most variable length types default to EXTENDED mode, except for the following types:

```
postgres=# select distinct typname, typalign, typstorage, typcategory, typlen
from pg_type where typtype='b' and typlen<0 and typstorage<>'x' order by typname;
 typname | typalign | typstorage | typecategory | typlen
------------+----------+-----------+-------------+--------
  cidr | i | m | I | -1
 gtsvector | i | p | U | -1
 inet | i | m | I | -1
 int2vector | i | p | A | -1
 numeric | i | m | N | -1
 oidvector | i | p | A | -1
 tsquery | i | p | U | -1
 (7 rows)
```

The EXTENDED mode first tries to compress the field and only then decides whether to TOAST the field or not. The EXTERNAL mode immediately TOASTs the field, leaving 18 bytes in the block.

2) The fields taken out to TOAST are divided into parts - "chunks". The division into chunks occurs after compression, if compression was applied.
The maximum chunk size is stored in the cluster control file and can be viewed using the `pg_controldata utility` :

```
postgres@tantor:~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk: 1996
```

If the field size (after compression, if applicable) exceeds **1996** bytes, the field is divided into two or more chunks, the first chunk being 1996 bytes in size. **1996** bytes is the maximum chunk size in Tantor and PostgreSQL (on 64-bit platforms).

For Astralinux PostgreSQL value is not **1996** bytes, but **1988** bytes:

```
postgres@tantor:~$ /usr/lib/postgresql/15/bin/pg_controldata -D
/var/lib/postgresql/15/main | grep TOAST
Maximum size of a TOAST chunk: 1988
```

PostgreSQL on 32-bit platforms has a maximum chunk size of 2000 bytes.

3) Let's see under what condition the fields start to be displaced in TOAST .
Create a table with two columns, insert rows and see the row size:

```
postgres=# drop table if exists t;
create table t(id int4, c text storage external);
insert into t VALUES (1, repeat('a',2000));
insert into t VALUES (1, repeat('a',2000));
insert into t VALUES (1, repeat('a',2000));
insert into t VALUES (1, repeat('a',2000));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
```

```
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size
----------------
2032
2032
2032
2032
(4 rows)
ERROR: block number 0 is out of range for relation "pg_toast_546660"
```

The fields were not TOASTed because the row size did not exceed TOAST_TUPLE_THRESHOLD =2032 bytes .
TOAST_TUPLE_THRESHOLD value = 2032 bytes is set in the PostgreSQL source code and is not overridden by parameters. If the table row being formed (INSERT , UPDATE , COPY) exceeds 2032 bytes, then the code of the compression algorithm and/or the removal of fields to the TOAST table is executed.

4) Increase the size of the text fields by one and repeat the commands:

```
postgres=# drop table if exists t;
create table t (id int4, c text storage external);
insert into t VALUES (1, repeat('a',200 1 ));
insert into t VALUES (1, repeat('a',200 1 ));
insert into t VALUES (1, repeat('a',200 1 ));
insert into t VALUES (1, repeat('a',200 1 ));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
pg_column_size
----------------
2033
2033
2033
2033
(4 rows)

lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2032 | (0.1) | 24
2 | 6104 | 41 | (0.2) | 24
3 | 4072 | 2032 | (0.3) | 24
4 | 4024 | 41 | (0.4) | 24
5 | 1992 | 2032 | (0.5) | 24
6 | 1944 | 41 | (0.6) | 24
(6 rows)
```

All four text fields four lines were displaced in TOAST.
The maximum size of a row in a TOAST table cannot exceed TOAST_TUPLE_THRESHOLD = 2032 bytes.

The text field in this table is evicted starting with the size of 2001 bytes . The criterion for starting or continuing field eviction is that pg_column_size (t .*) >2032 . In this case, the field values will be compressed (if the field storage mode is EXTENDED) and then cut into chunks (parts) of 1996 bytes. In this example , for a text field of 2001 bytes long, a small chunk of 41 bytes will appear, which will be inserted by the server process into a block with 2032-byte chunks. Because of this, only 3 chunks of 2032 bytes will fit in the TOAST table block, and there will be a lot of free space (1944 - 24 bytes of block header - 6 slots of block header * 4 bytes = 1896 bytes).

5) The length of the field candidate for displacement depends on the length of the string. Fields for displacement are selected according to the algorithm described in the chapter.
Let's look at another example with a table of one column :

```
postgres=# drop table if exists t;
create table t ( c text storage external);
insert into t VALUES (repeat('a', 2005 ));
insert into t VALUES (repeat('a', 2005 ));
insert into t VALUES (repeat('a', 2005 ));
insert into t VALUES (repeat('a', 2005 ));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
pg_column_size
----------------
2033
2033
2033
2033
(4 rows)

lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2032 | (0.1) | 24
2 | 6104 | 45 | (0.2) | 24
3 | 4072 | 2032 | (0.3) | 24
4 | 4024 | 45 | (0.4) | 24
5 | 1992 | 2032 | (0.5) | 24
6 | 1944 | 45 | (0.6) | 24
(6 rows)
```

Text field V this table is being displaced starting With size 2005 bytes . The criterion for the start of field displacement is still that pg _ column _ size (t .*)>2032 .

The preempted field is always cut into chunks of 1996 bytes.
In this example, for each text field of a 2005-byte field, there will be two chunks: the first chunk of 2032 bytes and the second chunk of 45 bytes. The small chunk will be inserted by the server process into the block with the first chunk of 2032 bytes, if possible. Chunks of one field are inserted into one block, if possible, based on the fact that chunks related to one field can be collected or deleted more quickly.
Because the field is divided into 2 chunks, only 3 chunks of 2032 bytes will fit in the TOAST table block, and 1896 bytes will be free (1944 - 24 bytes of block header - 6 slots of block header * 4 bytes = 1896 bytes). The free space can be used for chunks from other fields, if the chunks fit in the free space.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

6) Small chunks do not allow large chunks to fit into blocks. Let's see under what conditions small chunks do not appear.

Recreate the table with two 8-byte columns and a text column:

```
drop table if exists t;
create table t ( id int8, c1 int8 , c text storage external);
insert into t VALUES (1, 1, repeat('a',1989));
insert into t VALUES (1, 1, repeat('a',1989));
insert into t VALUES (1, 1, repeat('a',1989));
insert into t VALUES (1, 1, repeat('a',1989));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
pg_column_size
----------------
2033
2033
2033
2033
(4 rows)

lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2025 | (0.1) | 24
2 | 4120 | 2025 | (0.2) | 24
3 | 2088 | 2025 | (0.3) | 24
4 | 56 | 2025 | (0.4) | 24
```

Each field was pushed into one chunk instead of two chunks.

The block fit 4 chunks and there is almost no free space left in the block: 56-24-4*4=16 bytes.

However, if you insert a field larger than 1989 bytes :

```
insert into t VALUES (1, 1, repeat('a', 1990 ));
```

, then such a field will be divided into two chunks.

7) For better understanding, repeat the table creation. Recreate the table with three four-byte columns :

```
drop table if exists t;
create table t ( id int4, c1 int4, c2 int4 , c text storage external);
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',1993));
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
select pg_column_size(t.*) from t;
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',1));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
pg_column_size
----------------
2033
2033
2033
2033
2678
2678
2678
(7 rows)

lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2029 | (0.1) | 24
2 | 4120 | 2029 | (0.2) | 24
3 | 2088 | 2029 | (0.3) | 24
4 | 56 | 2029 | (0.4) | 24
(4 rows)

lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2032 | (1.1) | 24
2 | 5472 | 678 | (1.2) | 24
3 | 3440 | 2032 | (1.3) | 24
4 | 2760 | 678 | (1.4) | 24
5 | 728 | 2032 | (1.5) | 24
6 | 48 | 678 | (1.6) | 24
(6 rows)
```

Fields first 4 lines have one chunk And fit in V first TOAST table block .
The fields of the next three lines have two chunks each and occupy the second block.

8) The only thing that can be optimized here is that, from the point of view of space usage and performance, it makes no difference whether fields of 1997 bytes to 2640 bytes are stored without compression. Example :

```
truncate table t;
insert into t VALUES (1, 1, 1, repeat('a',1997));
insert into t VALUES (1, 1, 1, repeat('a',1997));
insert into t VALUES (1, 1, 1, repeat('a',1997));
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
TRUNCATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2032 | (0.1) | 24
2 | 6112 | 37 | (0.2) | 24
3 | 4080 | 2032 | (0.3) | 24
4 | 4040 | 37 | (0.4) | 24
5 | 2008 | 2032 | (0.5) | 24
6 | 1968 | 37 | (0.6) | 24
(6 rows )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Three fields starting from 1997 bytes without compression create two chunks for each field. Three fields will occupy one block. The block will have 1968-24-6*4=1920 bytes left, but they will hardly be used because there are more chunks of the maximum size than "cuts".

9) For the previous table with a 2640 byte field:

```
truncate table t;
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
insert into t VALUES (1, 1, 1, repeat('a',2640));
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select
reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
TRUNCATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2032 | (0.1) | 24
2 | 5472 | 680 | (0.2) | 24
3 | 3440 | 2032 | (0.3) | 24
4 | 2760 | 680 | (0.4) | 24
5 | 728 | 2032 | (0.5) | 24
6 | 48 | 680 | (0.6) | 24
(6 rows )
```

Three fields of 2636 bytes also created two chunks per field and also occupied one block of the TOAST table. There is no free space in the block.

10) For a table with two columns of 8 bytes:

```
create table t(id int8, c1 int8, c text);
insert into t VALUES (1, 1, repeat('a',1989));
```

1989 byte fields create one chunk and storage is optimal.

11) For a table with two columns of 4 bytes length:

```
create table t(id int4, c1 int4, c text);
insert into t VALUES (1, 1, repeat('a',1997));
```

The fields will create two chunks, which means storage will be less optimal.

For simplicity, the examples show tables with one column, displaced without compression, and a small number of non-displaced columns. In this case, the size of the first chunk will have the maximum size. If the table has many columns, the first chunks can be small. If the chunks are small, they can fit into the free space of the TOAST table blocks. When analyzing TOAST tables, it is worth paying attention to the free space in the TOAST blocks. If there is "a lot" of free space (from the point of view of someone who is engaged in optimization), then it is worth paying attention to the sizes of the chunks. If there are many chunks of the minimum size (up to ~100 bytes), then there is potential for optimization.
First of all, you can try changing the compression algorithm with the `default_toast_compression parameter` ; perhaps improving compression will reduce the number of small chunks.
Paradoxically, disabling compression can increase performance by eliminating the resource waste on compression and decompression without changing the size of the TOAST table. This will happen if:
a) fields are compressed less than 2 times
b) the field size is less than 1996*2=3992 bytes
c) after compression you get not 1 chunk, but two chunks, and without compression you also get two chunks.
You can try to increase (less likely decrease) the toast_tuple_target parameter . The parameter is set only at the table level :

```
ALTER TABLE t SET ( toast _ tuple _ target = 2032 );
```

Changing this parameter affects the number of rows that fit in a table. If you decrease the value below 2032 (the default value), more fields may be TOASTed and work with such fields will slow down. However, these fields can fill empty spaces in TOAST blocks. If you increase the value above 2032, more fields will remain in the table and the table size will increase. It is impossible to predict whether this will increase or decrease performance. For example, reducing the number of rows that fit in a table block can, in rare cases, reduce contention for access to buffer cache buffers.

Part 2. TOAST Table Structure

1) The TOAST table has two columns: `chunk_id` (OID type , unique for the field included in `TOAST`) , `chunk_seq` (chunk ordinal number), `chunk_data (field data). For quick access to chunks` , a composite unique index is created on the TOAST table `by chunk_id` and `chunk_seq` .

```
postgres=# drop table if exists t;
create table t (c text storage external );
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
insert into t VALUES (repeat('a',2005));
select lp,lp_off,lp_len,t_ctid,t_hoff from heap_page_items(get_raw_page((select reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
lp | lp_off | lp_len | t_ctid | t_hoff
----+--------+--------+--------+--------
1 | 6152 | 2032 | (0.1) | 24
2 | 6104 | 45 | (0.2) | 24
3 | 4072 | 2032 | (0.3) | 24
4 | 4024 | 45 | (0.4) | 24
5 | 1992 | 2032 | (0.5) | 24
6 | 1944 | 45 | (0.6) | 24
(6 rows)
```

Full size lines With long chunkom 2032 bytes (6104-4072).

2) A query that is convenient in that it produces one row per block and is useful for determining free space in a TOAST block and regular tables:

```
postgres=# select lower, upper, special, pagesize from page_header(get_raw_page((select reltoastrelid::regclass::text from pg_class where relname='t'),'main',0));
lower | upper | special | pagesize
-------+-------+---------+----------
48 | 1944 | 8184 | 8192
(1 row)
```

Example of how to calculate block space for 4 lines of 2032 bytes (with 4 chunks):
24 (header) + 4*4 (header) + 2032*4 + 8 (pagesize-special) = 8176. 16 bytes are not used, but they could not be used, since the lines are aligned to 8 bytes, and there are 4 of them.

3) Query to determine the name of the TOAST table to use the name in psql commands:

```
postgres=# select reltoastrelid, reltoastrelid::regclass from pg_class where relname='t';
reltoastrelid | reltoastrelid
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
----------------+-------------------------
546160 | pg_toast.pg_toast_546157
(1 row )
```

psql command to view TOAST table characteristics

```
postgres=# \d+ pg_toast.pg_toast_546157
TOAST table "pg_toast.pg_toast_546157"
Column | Type | Storage
------------+---------+---------
chunk_id | oid | plain
chunk_seq | integer | plain
chunk_data | bytea | plain
Owning table: "public.t"
Indexes:
"pg_toast_546157_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Access method : heap
```

5) Query the TOAST table to view chunk sizes:

```
postgres=# select chunk_id, chunk_seq, length(chunk_data) from
pg_toast.pg_toast_546157;
chunk_id | chunk_seq | length
----------+----------+--------
546162 | 0 | 1996
546162 | 1 | 9
546163 | 0 | 1996
546163 | 1 | 9
   546164 | 0 | 1996
546164 | 1 | 9
   546165 | 0 | 1996
546165 | 1 | 9
(8 rows)
```

length(chunk_data)=1996 corresponds meaning " Maximum size of a TOAST chunk: **1996** " V manager file .

6) The value remaining in the table block after the field is TOASTed :

```
postgres=# select lp_len, t_data from heap_page_items(get_raw_page('t', 0));
lp_len | t_data
--------+-------------------------------------
42 | \x0112d9070000d50700007255080070550800
42 | \x0112d9070000d50700007355080070550800
42 | \x0112d9070000d5070000 74550800 70550800
42 | \x01 12 d9070000 d507 0000 75550800 70550800
(4 rows)
```

\ x 07 d 5 = 2005 which is equal to **1996+ 9**
\ x 07 d 9 = 2009

These values store a pointer to the first chunk **of size 18 bytes (regardless of the field size)** .
These 18 bytes store the varatt_external structure, described in varatt.h:
the first byte has the value 0x01, this is a sign that the field is TOASTed;
the second byte is the length of this record (value 0x12 = 18 bytes);
4 bytes is the length of the field with the field header before compression;
4 bytes is the length of what is put into TOAST;
4 bytes - pointer to the first chunk in TOAST (chunk_id column of TOAST table);
4 bytes - oid toast- tables (pg_class.reltoastrelid)

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The chunk_id column (oid type 4 bytes) can have 4 billion (2 to the power 32) values. This means that only 4 billion **fields (not even rows)** can be TOASTed in one table. This significantly limits the number of rows in the original table.

's check if each field really has a separate chunk_id ?

```
drop table if exists t3;
create table t3 (a text storage external , b text storage external );
truncate t;
insert into t3 VALUES (repeat('a',2005), repeat('b',2005));
select lp, t_ctid, lp_off,lp_len,t_hoff,t_data from
heap_page_items(get_raw_page('t3', 0));
SELECT reltoastrelid, reltoastrelid::text FROM pg_class WHERE relname='t3';
SELECT reltoastrelid, reltoastrelid::regclass FROM pg_class WHERE relname='t3';
DROP TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
TRUNCATE TABLE
INSERT 0 1
 lp | t_ctid | lp_off | lp_len | t_hoff | t_data
----+--------+--------+--------+--------+--------------------------------
 1 | (0,1) | 8112 | 60 | 24 |
\x0112d9070000d50700000885508008355080000112d9070000d50700000895508008355080 0
 (1 row)

 reltoastrelid | reltoastrelid
---------------+---------------
 546169 | 546169
 (1 row)

 reltoastrelid | reltoastrelid
---------------+-------------------------
 546169 | pg_toast.pg_toast_546166
 (1 row)
ostgres=# select chunk_id, chunk_seq, length(chunk_data) from
pg_toast.pg_toast_546166;
 chunk_id | chunk_seq | length
----------+-----------+--------
    546174 | 0 | 1996
 546174 | 1 | 9
 546175 | 0 | 1996
 546175 | 1 | 9
 (4 rows )
```

Each field has a separate chunk_id.

Alignment by the length of the entire line, if 18 bytes remain, is also present.

8) How many lines will fit in a block?

```
drop table if exists t3;
drop table if exists t4;
create table t3 (c1 text storage external );
create table t4 (c1 serial , c2 smallint , c3 text storage external );
truncate t3;
truncate t4;
DO $$ BEGIN
FOR i IN 1 .. 200 LOOP
insert into t3 VALUES (repeat('a',1000000));
insert into t4 VALUES (default, 1, repeat('a',1000000));
```

```
END LOOP; END; $$ LANGUAGE plpgsql;
select count(*) from heap_page_items(get_raw_page('t3','main',0)) where
(t_ctid::text::point)[0]=0
union all
select count(*) from heap_page_items(get_raw_page('t4','main',0)) where
(t_ctid::text::point)[0]=0;
select pg_total_relation_size('t3'), pg_total_relation_size('t4');
DROP TABLE
DROP TABLE
CREATE TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
TRUNCATE TABLE
TRUNCATE TABLE
DO
count
-------
156
156
(2 rows )
```

If you replace serial and smallint with bigserial, the size will be different (156 and 135 rows in each block). bigserial is useless for tables in which many fields are displaced in TOAST , since the tables will be able to store no more than 4 billion (2^32) rows with one long field or 2 billion with two long fields.

```
pg_total_relation_size | pg_total_relation_size
-------------------------+-----------------------
208011264 | 208011264
(1 row )
```

9) Blocks of tables t3 and t4 contain the same data on free space in blocks. Example For first blocks tables :

```
SELECT * FROM page_header(get_raw_page((SELECT reltoastrelid::regclass::text
FROM pg_class WHERE relname='t3'),'main',0));
lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
8/3205CB70 | 0 | 4 | 40 | 56 | 8184 | 8192 | 5 | 0
(1 row)

SELECT * FROM page_header(get_raw_page((SELECT reltoastrelid::regclass::text
FROM pg_class WHERE relname='t4'),'main',0));
  lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+----------
8/25366AA0 | 25187 | 0 | 40 | 168 | 8184 | 8192 | 5 | 0
(1 row)
```

10) Each field contains a million bytes `repeat (' a ',1000000)` .b therefore most chunks have maximum sizes and chunk storage is compact:

```
SELECT lp,lp_off,lp_len,t_ctid,t_hoff FROM heap_page_items(get_raw_page((SELECT
reltoastrelid::regclass::text FROM pg_class WHERE relname='t3'),'main',0));
lp | lp_off | lp_len | t_ctid | t_hoff
----+---------+--------+--------+--------
1 | 6152 | 2032 | (0.1) | 24
2 | 4120 | 2032 | (0.2) | 24
3 | 2088 | 2032 | (0.3) | 24
4 | 56 | 2032 | (0.4) | 24
(4 rows)
```

There is no free space in the block between the block header and the data: 56 - 24 - 4*4 = 16 bytes. The `block` is filled completely and corresponds to the constants TOAST_TUPLE_TARGET = TOAST_TUPLE_THRESHOLD = 2032 .

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Row header 24 bytes. Row data area size 2032-24=2008 bytes.

11) In a row of any TOAST table there are three columns:

```
postgres=# \d pg_toast.pg_toast_546186
TOAST table "pg_toast.pg_toast_546186"
Column | Type
------------+---------
chunk_id | oid
chunk_seq | integer
chunk_data | bytea
Owning table: "public.t3"
Indexes:
"pg_toast_546186_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
```

12) Contents of the first row in the TOAST table:

```
postgres=# select * from
heap_page_items(get_raw_page('pg_toast.pg_toast_546186', 0)) limit 1 \gx
-[ RECORD 1 ]---
lp | 1
lp_off | 6152
lp_flags | 1
lp_len | 2032
t_xmin | 2
t_xmax | 0
t_field3 | 0
t_ctid | (0,1)
t_infomask2 | 3
t_infomask | 2818
t_hoff | 24
t_bits |
t_oid |
t_data | \x9c550800 00000000 401f0000 6161616161616161
```

bytea field is stored in the first 4 bytes of the bytea field: *401 f 0000*

13) Look at what the line remaining in the table looks like:

```
postgres=# select lp_len, t_data from heap_page_items(get_raw_page('t3', 0))
limit 2;
lp_len | t_data
--------+---------------------------------------
42 | \x01 12 44420f00 40420f00 9c550800 8d550800
42 | \x01 12 44420f00 40420f00 9e550800 8d550800
(2 rows)
```

`\x44420f00 = 1000004`

`\ x 40420 f 00 = 1000000` which corresponds to a million characters inserted into the field by the **`repeat (' a ',1000000) function`**

14) Create a table with a TOAST structure:

```
postgres=# drop table if exists t5;
create table t5 (a oid, b integer, c bytea);
insert into t5 values ( 1 , 1 , '');
insert into t5 values ( 2 , 2 , ' a ');
insert into t5 values ( 3 , 3 , ' aa ');
select lp,lp_off,lp_len,t_ctid,t_hoff,t_data from
heap_page_items(get_raw_page('t5', 0));
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
lp | lp_off | lp_len | t_ctid | t_hoff | t_data
----+--------+--------+--------+--------+-----------
1 | 8136 | 33 | (0,1) | 24 | \x 0 1 000000 0 1 000000 03
2 | 8096 | 34 | (0,2) | 24 | \x 0 2 000000 0 2 000000 05 61
3 | 8056 | 35 | (0.3) | 24 | \x 0 3 000000 0 3 000000 07 6161
(3 rows)
```

Length each from three lines : `8136-8096=8096-8056=40` byte .
The 40 bytes include a 24-byte header. Data area: 40-24=16=4 (the first column of the oid type)+4 (the second column of the integer type)+8 **bytes** . Why **8 bytes** ? After all, as a result of the query, the first row contains 1 byte , the second 2 bytes , and the third 3 bytes . Because the entire row is always aligned to 8 bytes. In `the lp_len column` The length of three lines is given without alignment: 33, 34, 35 bytes.

Part 3. UPDATE vs. INSERT Efficiency

If you update the same row multiple times in a single transaction, the space occupied by the generated row versions will not be able to be cleared. This can happen when using AFTER .. FOR triggers EACH ROW . It is worth checking the application code for the presence of FOR phrases EACH ROW , DEFERRABLE INITIALLY DEFERRED . The second thing you can pay attention to is if UPDATE prevails in the logic of working with tables instead of INSERT.

1) Do it commands :

```
postgres=# drop table if exists item;
drop table if exists sum;
drop function if exists add_to_sum();
create table item(id bigserial, value int8);
create table sum(total int8);
insert into sum values (0);
CREATE FUNCTION add_to_sum() RETURNS trigger LANGUAGE plpgsql AS
$$BEGIN
UPDATE sum SET total = total + NEW.value;
RETURN NEW;
END;$$;
CREATE CONSTRAINT TRIGGER add_to_sum
AFTER INSERT ON item
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION add_to_sum();
analyze item, sum;
DROP TABLE
DROP TABLE
DROP FUNCTION
CREATE TABLE
CREATE TABLE
INSERT 0 1
CREATE FUNCTION
CREATE TRIGGER
ANALYZE
```

Two tables item and sum are created . After inserting a row into the item table, the trigger updates the row in the sum table.

2) Enable command execution time measurement:

```
postgres=# \timing on
Timing is on.
```

3) Do it insert 10, 20, 30 thousand . lines :

```
postgres=# insert into item select * from generate_series(1,10000);
insert into item select * from generate_series(1.20000);
insert into item select * from generate_series(1.30000);
INSERT 0 10000
Time: 2998.265 ms (00:02.998)
INSERT 0 20000
Time: 11798.652 ms (00:11.799)
INSERT 0 30000
Time: 26589.966 ms (00:26.590)
```

Each command is executed in a single transaction. Row updates by a trigger are executed in the same transaction.

As the number of rows inserted in a single transaction increases, the command execution time increases nonlinearly.

It is not worth following the ideas of "the less frequent COMMIT, the higher the performance", "batch updates are faster than single ones". In PostgreSQL, increasing the duration of a transaction leads to a deterioration in performance due to the inability to clean up (HOT and autovacuum) blocks from old row versions.

4) Recreate the trigger and trigger function:

```
postgres=# drop trigger add_to_sum on item;
drop function add_to_sum();
CREATE OR REPLACE FUNCTION add_to_sum() RETURNS trigger
LANGUAGE plpgsql AS
$$BEGIN
UPDATE sum SET total = total + (SELECT sum(value) FROM new_rows);
RETURN NULL;
END;$$;
CREATE TRIGGER add_to_sum
AFTER INSERT ON item
REFERENCING NEW TABLE AS new_rows
FOR EACH STATEMENT
EXECUTE FUNCTION add_to_sum();
DROP TRIGGER
Time: 7.679 ms
DROP FUNCTION
Time: 3.456 ms
CREATE FUNCTION
Time: 4.232 ms
CREATE TRIGGER
Time : 5.288 ms
```

5) Repeat the commands to insert 10, 20, 30 thousand rows:

```
postgres=# insert into item select * from generate_series(1,10000);
insert into item select * from generate_series(1.20000);
insert into item select * from generate_series(1.30000);
INSERT 0 10000
Time: 22.831 ms
INSERT 0 20000
Time: 44.134 ms
INSERT 0 30000
Time : 62.785 ms
```

Command execution time has been reduced hundreds of times from `26589.966 ms` up to `62.785 ms` and became linear.

FOR EACH ROW was replaced with **FOR EACH STATEMENT** this added efficiency since the trigger function is called much less frequently.

It is more interesting to compare the efficiency of replacing UPDATE logic with INSERT . When designing application logic, it may seem that updating rows does not increase the number of rows and is more preferable than inserting rows (an application that accumulates data) and then periodically deleting unnecessary rows.

Recreate the trigger and trigger function, replacing the row update with a new row insertion:

```
postgres=# drop trigger add_to_sum on item;
drop function if exists add_to_sum();
CREATE FUNCTION add_to_sum() RETURNS trigger LANGUAGE plpgsql AS
$$BEGIN
insert into sum values (NEW.value);
RETURN NEW;
END;$$;
CREATE CONSTRAINT TRIGGER add_to_sum
AFTER INSERT ON item
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW
EXECUTE FUNCTION add_to_sum();
DROP TRIGGER
Time: 5.033 ms
DROP FUNCTION
Time: 3.456 ms
CREATE FUNCTION
Time: 5.270 ms
CREATE TRIGGER
Time : 4.590 ms
```

7) Repeat the commands to insert 10, 20, 30 thousand rows:

```
postgres=# insert into item select * from generate_series(1,10000);
insert into item select * from generate_series(1,20000);
insert into item select * from generate_series(1.30000);
INSERT 0 10000
Time: 87.187 ms
INSERT 0 20000
Time: 188.129 ms
INSERT 0 30000
Time : 247.266 ms
```

The duration of inserting 30 thousand lines has decreased by a hundred times from **26589.966 ms** up to **247.266 ms** .

In this case, the trigger function was triggered for each line.

The effect of replacing application logic from UPDATE to INSERT can be significant.

FOR EACH ROW trigger execution compared to **FOR triggering once EACH STATEMENT** ~4 times: **247.266 ms** compared to **62.785 ms** .

We also found out the effect of optimization: replacing **FOR EACH ROW** on **FOR EACH STATEMENT ~4** times.

8) Turn off command execution time measurement:

```
postgres=# \timing off
Timing is off.
```

9) Delete created objects :

```
postgres=# drop trigger add_to_sum on item;
drop table if item exists;
drop table if exists sum;
drop function if exists add_to_sum();
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
DROP TRIGGER
DROP TABLE
DROP TABLE
DROP FUNCTION
```

Part 4. HOT cleanup

When executing SELECT queries and other commands, the server process may remove dead tuples (row versions that have passed the database visibility horizon, xmin horizon) and reorganize row versions within a block. This is called in-page cleanup.

HOT cleanup ("pruning") is performed if one of the following conditions is met:
a) the block is more than 90% full or more than FILLFACTOR (default 100%);
b) the previously executed UPDATE did not find a place (that is, it set the PD_PAGE_FULL hint in the block header).

In-page vacuuming works within a single table page, does not vacuum index pages, and does not update the free space map or visibility map.
The pointers (4 bytes) in the block header are not freed, they are updated to point to the current version of the row. The pointers cannot be freed, because they may be referenced from indexes, the server process cannot check this. Only vacuum can free the pointers (make the pointers unused).

In this part of the practice you will see in which case the condition " b " is not met and how this affects the execution of HOT cleanup .

1) Create a function to easily view hint bits:

```
create or replace function heap_page(relname text, pageno integer) returns
table(lp_off text, ctid tid,state text,xmin text,xmax text,hhu text,hot
text,t_ctid tid, multi text)
as $$
select lp_off, (pageno,lp)::text::tid as ctid,
case lp_flags
when 0 then 'unused'
when 1 then 'normal'
when 2 then 'redirect to '|| lp_off
when 3 then 'dead'
end as state,
t_xmin || case
when (t_infomask & 256) > 0 then 'c'
when (t_infomask & 512) > 0 then 'a'
else ''
end as xmin,
t_xmax || case
when (t_infomask & 1024) >0 then 'c'
when (t_infomask & 2048) >0 then 'a'
else ''
end as xmax,
case when (t_infomask2 & 16384) >0 then 't' end as hhu,
case when (t_infomask2 & 32768) >0 then 't' end as hot,
t_ctid,
case when (t_infomask&4096) >0 then 't' else 'f'
end as multi
from heap_page_items (get_raw_page(relname, pageno))
order by lp;
$$ language sql;
CREATE FUNCTION
```

2) Create a table, insert a row and update this row three times:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
drop table if exists t;
create table t(s text storage plain) with (autovacuum_enabled=off);
insert into t values (repeat('a',2004));
update t set s=(repeat('c',2004));
update t set s=(repeat('c',2004));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
DROP TABLE
CREATE TABLE
INSERT 0 1
UPDATE 1
UPDATE 1
UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+---------+-----+-----+--------+-------
(0,1) | 6144 | (0,1) | normal | 424011c | 424012c | t | | (0,2) | f
(0,2) | 4112 | (0,2) | normal | 424012c | 424013c | t | t | (0.3) | f
(0.3) | 2080 | (0.3) | normal | 424013c | 424014 | t | t | (0.4) | f
(0.4) | 48 | (0.4) | normal | 424014 | 0a | | t | (0.4) | f
(4 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 40 | 48 | 8176 | 424012
(1 row )
```

A chain of four row versions is organized in a block. The value in the column x max =0 a means that this row version is not deleted. If there is no " c " in xmin, this means that the status of the transaction that generated the row version is not specified. If there is " c " in the column xmin, this means that the transaction that created the row version was committed.

lp_off = 48 or upper =48 means that the block is more than 90% full.

The prune_xid flag was set by the first UPDATE . The prune_xid flag is a hint (not a command) that there is a chance that space can be cleared in the block.

The first UPDATE set up the HOT chain by setting a hint bit:

hhu = t - a hint to processes that they should follow the ctid chain .

hot = t - a hint to processes that this version of the row is not referenced from indexes.

3) If you continue to update the line, the updates will occur within one block:

```
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2004));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+---------------+--------+---------+-----+-----+--------+-------
(0,1) | 4 | (0,1) | redirect to 4 | | | | | | f
```

```
(0,2) | 4112 | (0,2) | normal | 424015 | 0a | | t | (0,2) | f
(0.3) | 0 | (0.3) | unused | | | | | | f
(0.4) | 6144 | (0.4) | normal | 424014c | 424015 | t | t | (0,2) | f
(4 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
1 | 40 | 4112 | 8176 | 424012
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
(0,1) | 4 | (0,1) | redirect to 4 | | | | | | f
(0,2) | 4112 | (0,2) | normal | 424015c | 424016 | t | t | (0.3) | f
(0.3) | 2080 | (0.3) | normal | 424016 | 0a | | t | (0.3) | f
(0.4) | 6144 | (0.4) | normal | 424014c | 424015c | t | t | (0,2) | f
(4 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
1 | 40 | 2080 | 8176 | 424012
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
(0,1) | 4 | (0,1) | redirect to 4 | | | | | | f
(0,2) | 4112 | (0,2) | normal | 424015c | 424016c | t | t | (0.3) | f
(0.3) | 2080 | (0.3) | normal | 424016c | 424017 | t | t | (0.5) | f
(0.4) | 6144 | (0.4) | normal | 424014c | 424015c | t | t | (0,2) | f
(0.5) | 48 | (0.5) | normal | 424017 | 0a | | t | (0.5) | f
(5 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 44 | 48 | 8176 | 424012
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+--------+--------+-----+-----+--------+-------
(0,1) | 5 | (0,1) | redirect to 5 | | | | | | f
(0,2) | 4112 | (0,2) | normal | 424018 | 0a | | t | (0,2) | f
(0.3) | 0 | (0.3) | unused | | | | | | f
(0.4) | 0 | (0.4) | unused | | | | | | f
(0.5) | 6144 | (0.5) | normal | 424017c | 424018 | t | t | (0,2) | f
(5 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
1 | 44 | 4112 | 8176 | 424012
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
(0,1) | 5 | (0,1) | redirect to 5 | | | | | | f
(0,2) | 4112 | (0,2) | normal | 424018c | 424019 | t | t | (0.3) | f
(0.3) | 2080 | (0.3) | normal | 424019 | 0a | | t | (0.3) | f
(0.4) | 0 | (0.4) | unused | | | | | | f
(0.5) | 6144 | (0.5) | normal | 424017c | 424018c | t | t | (0,2) | f
(5 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
1 | 44 | 2080 | 8176 | 424012
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+----------------+---------+---------+-----+-----+--------+-------
(0,1) | 5 | (0,1) | redirect to 5 | | | | | | f
(0,2) | 4112 | (0,2) | normal | 424018c | 424019c | t | t | (0.3) | f
(0.3) | 2080 | (0.3) | normal | 424019c | 424020 | t | t | (0.4) | f
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
(0.4) | 48 | (0.4) | normal | 424020 | 0a | | t | (0.4) | f
(0.5) | 6144 | (0.5) | normal | 424017c | 424018c | t | t | (0,2) | f
(5 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
1 | 44 | 48 | 8176 | 424012
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------------+--------+--------+-----+-----+--------+-------
(0,1) | 4 | (0,1) | redirect to 4 | | | | | | f
(0,2) | 4112 | (0,2) | normal | 424021 | 0a | | t | (0,2) | f
(0.3) | 0 | (0.3) | unused | | | | | | f
(0.4) | 6144 | (0.4) | normal | 424020c | 424021 | t | t | (0,2) | f
(4 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
1 | 40 | 4112 | 8176 | 424012
(1 row)

ERROR: block number 1 is out of range for relation "t"
CONTEXT: SQL function "heap_page" statement 1
```

There are no rows in block 1, all updates are within block 0.

4) Let's see what happens if we increase the length of the text field from 2004 to 2005 bytes:

```
truncate table t;
insert into t values (repeat('a', 2005 ));
update t set s=(repeat('c',2005));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
select ctid,* from heap_page('t',2);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 2));
update t set s=(repeat('c',2005));
select ctid,* from heap_page('t',0);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 0));
select ctid,* from heap_page('t',1);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 1));
select ctid,* from heap_page('t',2);
select flags, lower, upper, special, prune_xid from page_header(get_raw_page('t', 2));
TRUNCATE TABLE
INSERT 0 1
UPDATE 1
UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
-------+--------+-------+--------+--------+---------+-----+-----+--------+-------
(0,1)  | 6136 |  (0,1)  | normal | 424023c | 424024c | t |  |  (0,2)  | f
(0,2)  | 4096 |  (0,2)  | normal | 424024c | 424025c | t | t |  (0.3)  | f
(0.3)  | 2056 |  (0.3)  | normal | 424025 | 0a |  | t |  (0.3)  | f
(3 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0  | 36 | 2056  | 8176 | 424024
(1 row)
```

flags=0, and not 2 (PD_PAGE_FULL) . The clearing condition " previously executed UPDATE did not find space (i.e. set the PD_PAGE_FULL hint in the block header) " is not met.

upper =2056 means that there is more than 10% free space in the block (free space 2056-36=2020 bytes). The cleaning condition " the block is filled more than 90% or more than FILLFACTOR (default 100%) " is not met.

Since none of the cleanup conditions were met, the process does not perform cleanup.

```
UPDATE 1
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+---------+-----+-----+--------+-------
(0,1)  | 6136 |  (0,1)  | normal | 424023 c | 424024 c | t |  |  (0,2)  | f
(0,2)  | 4096 |  (0,2)  | normal | 424024 c | 424025 c | t | t |  (0.3)  | f
(0.3)  | 2056 |  (0.3)  | normal | 424025 c | 424026 |  | t |  ( 1,1 )  | f
(3 rows )
```

There is not enough free space in the block to insert a new version of the row, since there is 2056-36=2020 bytes of free space. These 2020 bytes must fit: 2005 bytes of data + 4 bytes for a new slot in the block header + 3 bytes for storing the field length + 24-byte row header + aligning the row to 8 bytes. The process sets the third `(0.3)` lines:

a) `xmax=424026` ;

b) the commit flag of the transaction that generated the version **`xmin =`** `424025` **`c`** ;

c) pointer to a new version **`of the`** row in another block **`t_ctid =( 1,1 )`** ;

d) hint PD_PAGE_FULL in the block header that there was no space in the block: **`flags = 2`** .

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     2 | 36 | 2056 | 8176 | 424024
(1 row )
```

`flags=2` means there is no room in the block. **`flags=2`** is set if a process looked for space in a block, was unable to allocate a row version, and set a flag to tell other processes there is no room in the block.

```
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+------+-----+-----+--------+-------
(1,1)  | 6136 |  (1,1)  | normal | 424026 | 0 a |  |  |  (1,1)  | f
(1 row )
```

the line was inserted into the second block.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0  | 28 | 6136 | 8176 | 0
(1 row )
```

The second block header has no flags or hints.

```
UPDATE 1
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
(0,1)  | 0 |  (0,1)  | dead |  |  |  |  |  | f
(1 row )
```

The process updating the string scanned all blocks (using the Seq Scan access method) in search of the string. Seeing the flag **`flags=2 in the first block`** , the process quickly cleared the first

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

block. As a result of clearing, space in the first block was freed. The slots of pointers in the block header can be freed from the end, which is what was done. There was one slot left with `the state=dead status .`

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 28 | 8176 | 8176 | 424024
(1 row )
```

`The` flag was set to 0 after cleaning. `prune_xid did` not change.

```
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+--------+-----+-----+--------+-------
(1,1) | 6136 | (1,1) | normal | 424026 c | 424027 | t | | (1,2) | f
(1,2) | 4096 | (1,2) | normal | 424027 | 0 a | | t | (1,2) | f
(2 rows )
```

The new version of the string was added to the second block, creating a HOT chain in that block.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 32 | 4096 | 8176 | 424027
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
(0,1) | 0 | (0,1) | dead | | | | | | f
(1 row)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 28 | 8176 | 8176 | 424024
(1 row)

ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+---------+-----+-----+--------+-------
(1,1) | 6136 | (1,1) | normal | 424026c | 424027c | t | | (1,2) | f
(1,2) | 4096 | (1,2) | normal | 424027c | 424028 | t | t | (1,3) | f
(1,3) | 2056 | (1,3) | normal | 424028 | 0a | | t | (1,3) | f
(3 rows)
```

The HOT chain grows as long as there is space in the block.

```
 flags | lower | upper | special | prune_xid to
-------+-------+-------+---------+-----------
0 | 36 | 2056 | 8176 | 424027
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
(0,1) | 0 | (0,1) | dead | | | | | | f
(1 row)
```

First block Not changed . fsm file if He yes , HOT no is being updated . fsm is being updated vacuum .

```
flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 28 | 8176 | 8176 | 424024
(1 row)

ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+---------+-----+-----+--------+-------
(1,1) | 6136 | (1,1) | normal | 424026c | 424027c | t | | (1,2) | f
(1,2) | 4096 | (1,2) | normal | 424027c | 424028c | t | t | (1,3) | f
(1,3) | 2056 | (1,3) | normal | 424028c | 424029 | | t | (2,1) | f
(3 rows)

flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
     2 | 36 | 2056 | 8176 | 424027
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
(1 row )
```

the process searched for space in a block, could not accommodate a row version, and set the PD_PAGE_FULL flag in the block header (**flags = 2**)

```
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+------+-----+-----+--------+-------
( 2 ,1) | 6136 | (2,1) | normal | 424029 | 0 a | | | (2,1) | f
(1 row )
```

Since there is either no fsm (free space map) or no fsm , the server process does not use the first block, but appends the block to the file and inserts the line into the new, third block.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 28 | 6136 | 8176 | 0
(1 row)

UPDATE 1
ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
(0,1) | 0 | (0,1) | dead | | | | | | f
(1 row)
```

First block Not changed his mind

```
flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 28 | 8176 | 8176 | 424024
(1 row)

ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+-------+------+------+-----+-----+--------+-------
(1,1) | 0 | (1,1) | dead | | | | | | f
(1 row )
```

the process performed a quick cleanup of the second block because it saw the flag **flags=2 in it**

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 28 | 8176 | 8176 | 424027
(1 row)

ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
-------+--------+-------+--------+--------+--------+-----+-----+--------+-------
(2,1) | 6136 | (2,1) | normal | 424029c | 424030 | t | | (2,2) | f
(2,2) | 4096 | (2,2) | normal | 424030 | 0a | | t | (2,2) | f
(2 rows )
```

The string was inserted into the third block, starting the HOT chain.

```
 flags | lower | upper | special | prune_xid
-------+-------+-------+---------+-----------
0 | 32 | 4096 | 8176 | 424030
(1 row )
```

What is interesting: for a table with one column of the `text type` , a field starting from the size of 2005 bytes will be displaced in TOAST. A field of 2004 bytes and less will not be displaced and for fields of this size the `storage mode plain` specified when creating the table has no effect.

The practical meaning of this part of the practice is that if the block fill percentage is less than 90% or FILLFACTOR, and the new row version or the row being inserted does not fit into the available space, then the rows will be inserted into new blocks. This will not happen if the row length is 816 bytes (10% of 8192 bytes, taking into account alignment) or less. The exact statistics of cases when no space was found in a block during UPDATE and the new row version was inserted into another block are shown by **pg_stat_all_tables.n_tup_newpage_upd** .

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

The example used a single row update, but multiple rows, row delete and row insert commands can be used. H OT cleanup cleans up not only HOT chains , but also any deleted row versions that have crossed the database horizon.

HOT Monitoring cleanup

HOT monitoring uses columns from **the pg_stat_all_tables** and **pg_stat_user_tables views** . Example:

```
postgres=# select relname, n_tup_upd, n_tup_hot_upd, n_tup_newpage_upd,
round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio from pg_stat_all_tables where
n_tup_upd<>0 order by 5;
 relname | n_tup_upd | n_tup_hot_upd | n_tup_newpage_upd | hot_ratio
---------------+-----------+--------------+-------------------+-----------
 pg_rewrite | 14 | 9 | 5 | 64.00
 pg_proc | 33 | 23 | 10 | 69.00
 pg_class | 71645 | 63148 | 8351 | 88.00
  pg_attribute | 270 | 267 | 3 | 98.00
 pg_statistic | 3974 | 3904 | 70 | 98.00
```

Let's compare examples from the previous part of the practice when a new version of a line or an inserted line does not fit into the free space of a block and when it does.

1) Run the commands to create tables and a script for the test:

```
postgres@tantor:~$ psql -c "drop table if exists t1;"
psql -c "create table t1(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t1 values (repeat('a',2004));"
psql -c "vacuum analyze t1;"
echo "update t1 set s=(repeat('c',2004));" > hot1.sql
psql -c "drop table if exists t2;"
psql -c "create table t2(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t2 values (repeat('a',2005));"
psql -c "vacuum analyze t2;"
echo "update t2 set s=(repeat('c',2005));" > hot2.sql
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
```

The tables have autovacuum enabled, unlike the previous part of the practice. In the previous part, we studied step by step what the contents of the blocks look like after changes, in order to understand what happens to the blocks at a low level. Tests should be close to real operating conditions.

2) By default, the autovacuum call interval is 1 minute. In order not to waste time on long tests, set the autovacuum call interval to 1 second. It is recommended to do this on loaded DBMS. We will not pay attention to the autoanalysis settings, although frequent statistics updates are usually useless (the collected statistics are the same as before the recollection) and reduce performance. Why is it often believed that "statistics must be updated", "statistics must be up-to-date"? Because in Oracle Database, if you do not update statistics, it is equivalent to its absence, and this logic is transferred to other DBMS. In PostgreSQL, there are no "stale" statistics, the statistics that are available are used.

Execute commands that reduce the frequency of calling the autovacuum to 1 second:

```
postgres@tantor:~$ psql -c "alter system set autovacuum_naptime = '1s';"
pg_ctl reload
ALTER SYSTEM
server signaled
```

3) Run the test and see the result:

```
postgres@tantor:~$ pgbench -T 30-f hot1.sql 2> /dev/null | grep tps
pgbench -T 30-f hot2.sql 2> /dev/null | grep tps
psql -c "select pg_relation_size('t1') t1, pg_relation_size('t2') t2";
psql -c "select relname name, n_tup_upd upd, n_tup_hot_upd hot_upd,
n_tup_newpage_upd newpage_upd, round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio,
seq_scan sel, n_tup_ins ins, n_tup_del del, n_live_tup live, n_dead_tup dead,
autovacuum_count vacuum from pg_stat_all_tables where n_tup_upd<>0 and relname in
('t1','t2') order by 1;"
tps = 246 .374879 (without initial connection time)
tps = 245 .593952 (without initial connection time)
t1|t2
------+--------
8192 | 688128
(1 row)

name | upd | hot_upd | newpage_upd | hot_ratio | sel | ins | del | live | dead | vacuum
-----+------+---------+------------+-----------+------+-----+-----+------+------+--------
t1 | 7391 | 7391 | 0 | 100.00 | 7391 | 1 | 0 | 1 | 2 | 0
t2 | 7367 | 4915 | 2452 | 66.00 | 7367 | 1 | 0 | 1 | 77 | 29
(2 rows)
```

Column **n_tup_newpage_upd** shows that V test With table t2 interrupted the HOT chain And new versions lines were inserted V new blocks . **Percentage** of rows that were updated HOT less than - 66 **% . Column n_tup_del** shows how many rows were deleted by the DELETE command. Column **n _ tup _ ins** shows how many rows were inserted by the INSERT command.

of row **reads seq_scan** by the Seq Scan method is approximately the same.

TPS has decreased slightly and only because of autovacuum and autoanalysis.

Thanks to the autovacuum, which was triggered every second on the t2 table, the table did not swell much, only to 688128 byte. If the autovacuum had not worked during the test, the table would have grown to **10,444,800** bytes .

4) The test was conducted in one session. How will the results change if the tables are updated in several sessions? Let's perform such a test.

Recreate the tables:

```
postgres @ tantor :~$ psql - c " drop table if exists t 1;"
psql -c "create table t1(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t1 values (repeat('a',2004));"
psql -c "vacuum analyze t1;"
echo "update t1 set s=(repeat('c',2004));" > hot1.sql
psql -c "drop table if exists t2;"
psql -c "create table t2(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t2 values (repeat('a',2005));"
psql -c "vacuum analyze t2;"
echo "update t2 set s=(repeat('c',2005));" > hot2.sql
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
DROP TABLE
CREATE TABLE
INSERT 0 1
VACUUM
```

5) Run tests with 4 sessions (or the number of processor cores):

```
pgbench -T 30 -c 4 -f hot1.sql 2> /dev/null | grep tps
pgbench -T 30 -c 4 -f hot2.sql 2> /dev/null | grep tps
psql -c "select pg_relation_size('t1') t1, pg_relation_size('t2') t2";
psql -c "select relname name, n_tup_upd upd, n_tup_hot_upd hot_upd,
n_tup_newpage_upd newpage_upd, round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio,
seq_scan sel, n_tup_ins ins, n_tup_del del, n_live_tup live, n_dead_tup dead,
```

```
autovacuum_count vacuum, autoanalyze_count analyze from pg_stat_all_tables where
n_tup_upd<>0 and relname in ('t1','t2') order by 1;"
```

```
tps = 251 .225621 (without initial connection time)
tps = 250 .060665 (without initial connection time)
t1 | t2
-------+---------
32768 | 2244608
(1 row)

name | upd | hot_upd | newpage_upd | hot_ratio | sel | ins | del | live | dead | vacuum | analyze
------------+------+---------+-------------+-----------+------+-----+-----+------+------+--------+---------
t1 | 7537 | 5480 | 2057 | 72.00 | 7537 | 1 | 0 | 1 | 0 | 10 | 10
t2 | 7501 | 4873 | 2628 | 64.00 | 7501 | 1 | 0 | 1 | 12 | 9 | 9
(2 rows)
```

TPS is the same , so How number activations autovacuum And autoanalysis Not differ .

TPS has increased, which is expected, since the maximum TPS zone is reached starting from the number of sessions corresponding to the number of processor cores.

HOT percentage were reduced. Multiple processes were making changes to the same block, and in most cases the first process inserted a row (was in the process of inserting a new version of the row and changing the block), and the second process could no longer find free space in the block.

6) Reduce the string size to 900 bytes:

```
psql -c "drop table if exists t1;"
psql -c "drop table if exists t2;"
psql -c "create table t2(s text storage plain) with (autovacuum_enabled=on);"
psql -c "insert into t2 values (repeat('a',900));"
psql -c "vacuum analyze t2;"
echo "update t2 set s=(repeat('c',900));" > hot2.sql
pgbench -T 10 -c 4 -f hot2.sql 2> /dev/null | grep tps
psql -c "select pg_relation_size('t2')";
psql -c "select relname name, n_tup_upd upd, n_tup_hot_upd hot_upd,
n_tup_newpage_upd newpage_upd, round(n_tup_hot_upd*100/n_tup_upd,2) as hot_ratio,
seq_scan sel, n_tup_ins ins, n_tup_del del, n_live_tup live, n_dead_tup dead,
autovacuum_count vacuum, autoanalyze_count analyze from pg_stat_all_tables where
n_tup_upd<>0 and relname='t2' order by 1;"
```

```
tps = 251.986305 (without initial connection time)
pg_relation_size
------------------
524288
(1 row)

name | upd | hot_upd | newpage_upd | hot_ratio | sel | ins | del | live | dead | vacuum | analyze
------------+------+---------+-------------+-----------+------+-----+-----+------+------+--------+---------
 t 2 | 2519 | 2184 | 335 | 86.00 | 2519 | 1 | 0 | 1 | 41 | 4 | 4
(1 row )
```

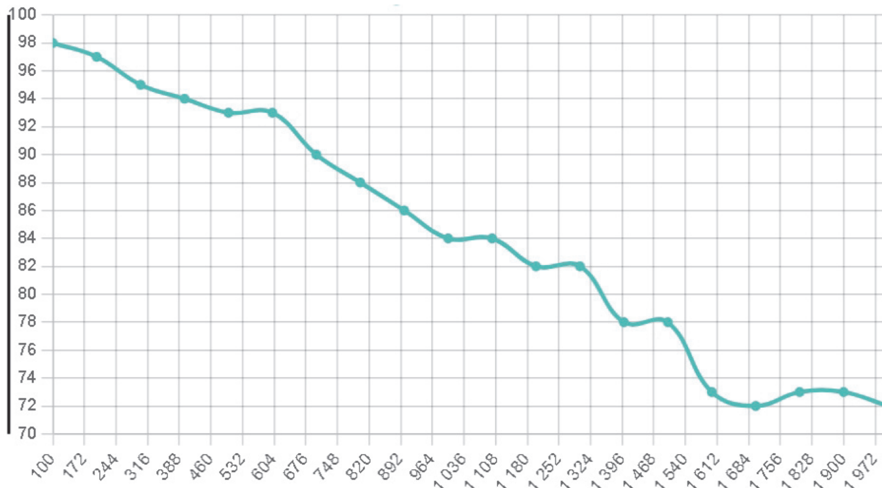
The result has improved. We can continue changing the line length, but we will not waste time on this. It is enough to look at the result depending on the number of characters in the field :

```
length | upd | hot_upd | newpage_upd | hot_ratio
-----+------+---------+-------------+----------
100 : 2519 | 2478 | 41 | 98.00
200 : 2513 | 2439 | 74 | 97.00
300 : 2536 | 2429 | 107 | 95.00
400 : 2523 | 2379 | 144 | 94.00
500 : 2599 | 2421 | 178 | 93.00
600 : 2523 | 2348 | 175 | 93.00
700 : 2515 | 2278 | 237 | 90.00
800 : 2509 | 2218 | 291 | 88.00
900 : 2529 | 2195 | 334 | 86.00
1000 : 2521 | 2131 | 390 | 84.00
1100 : 2513 | 2132 | 381 | 84.00
1200 : 2532 | 2089 | 443 | 82.00
1300 : 2523 | 2076 | 447 | 82.00
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
1400 : 2503 | 1953 | 550 | 78.00
1500 : 2860 | 2252 | 608 | 78.00
1600 : 2891 | 2115 | 776 | 73.00
1700 : 2878 | 2091 | 787 | 72.00
1800 : 2930 | 2168 | 762 | 73.00
1900: 2945 | 2175 | 770 | 73.00
```

Schedule:



The most efficient way to design data storage schemes is to make the row size small. Judging by the graph, a reasonable row size is 600 bytes.

You can test changing the FILLFACTOR value and make sure that it does not affect the probability of HOT cleanup. Setting FILLFACTOR to a value other than the default (FILLFACTOR =100) is useless in most cases and only increases the size of the table files.

The test could be made more realistic by inserting a large number of rows into the table so that index access is used, but this would make it more difficult to identify dependencies.

Part 6. Small Data Types

1) Look at the list of data types:

```
postgres=# select typname, typalign, typstorage, typecategory, typlen from
pg_type where typtype='b' and typcategory <>'A' order by typlen,typalign,typname;
 typname | typalign | typstorage | typecategory | typlen
------------------------------+----------+------------+-------------+--------
 path | d | x | G | -1
 pg_snapshot | d | x | U | -1
 polygon | d | x | G | -1
 txid_snapshot | d | x | U | -1
 bit | i | x | V | -1
 bpchar | i | x | S | -1
 byte | i | x | U | -1
  cidr | i | m | I | -1
 gtsvector | i | p | U | -1
 inet | i | m | I | -1
 json | i | x | U | -1
 jsonb | i | x | U | -1
 jsonpath | i | x | U | -1
  numeric | i | m | N | -1
 pg_brin_bloom_summary | i | x | Z | -1
 pg_brin_minmax_multi_summary | i | x | Z | -1
 pg_dependencies | i | x | Z | -1
 pg_mcv_list | i | x | Z | -1
 pg_ndistinct | i | x | Z | -1
 pg_node_tree | i | x | Z | -1
  refcursor | i | x | U | -1
 text | i | x | S | -1
```

```
tsquery | i | p | U | -1
tsvector | i | x | U | -1
varbit | i | x | V | -1
varchar | i | x | S | -1
xml | i | x | U | -1
 bool | c | p | B | 1
char | c | p | Z | 1
int2 | s | p | N | 2
cid | i | p | U | 4
 date | i | p | D | 4
float4 | i | p | N | 4
int4 | i | p | N | 4
oid | i | p | N | 4
regclass | i | p | N | 4
regcollation | i | p | N | 4
regconfig | i | p | N | 4
regdictionary | i | p | N | 4
regnamespace | i | p | N | 4
regoper | i | p | N | 4
regoperator | i | p | N | 4
regproc | i | p | N | 4
regprocedure | i | p | N | 4
regrole | i | p | N | 4
regtype | i | p | N | 4
macaddr | i | p | U | 6
tid | s | p | U | 6
float8 | d | p | N | 8
int8 | d | p | N | 8
money | d | p | N | 8
pg_lsn | d | p | U | 8
time | d | p | D | 8
timestamp | d | p | D | 8
timestamptz | d | p | D | 8
xid8 | d | p | U | 8
macaddr8 | i | p | U | 8
xid | x | p | U | 8
timetz | d | p | D | 12
uuid | c | p | U | 16
aclitem | d | p | U | 16
interval | d | p | T | 16
point | d | p | G | 16
circle | d | p | G | 24
line | d | p | G | 24
box | d | p | G | 32
lseg | d | p | G | 32
name | c | p | S | 64
(68 rows )
```

typlen =-1 variable width data types
typalign =i 4-byte alignment, d 8-byte alignment
typstorage default storage mode
The bool and char data types have a fixed size of 1 byte.
bpchar(N) is synonym char(N).

2) You can confuse "char" with char (synonymous with character(1) or char(1)). char takes 2 bytes instead of 1, but stores characters in the database encoding, i.e. more characters than in ASCII:

```
drop table if exists t5;
create table t5( c1 "char" default '1');
insert into t5 values(default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;

 lp_off | lp_len | t_hoff | t_data
--------+--------+--------+--------
   8144 | 25 | 24 | \x 31

drop table if exists t5;
create table t5( c1 char default '1');
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
insert into t5 values(default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;

lp_off | lp_len | t_hoff | t_data
--------+--------+--------+--------
   8144 | 26 | 24 | \ x 0531
```

"char" takes 1 byte, and char takes 2 bytes. Why is lp_off (the offset within the block that points to the beginning of the string) the same? Because the length of the entire string is aligned to 8 bytes. "char" is intended for use in system catalog tables, but can be used in regular tables.

3) The boolean type also takes 1 byte, but stores only two values: true, false and empty (NULL).

```
postgres=# drop table if exists t5;
create table t5( c1 boolean default true);
insert into t5 values(default);
insert into t5 values(false);
insert into t5 values(NULL);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
lp_off | lp_len | t_hoff | t_data
--------+--------+--------+--------
8088 | 24 | 24 | \x
8112 | 25 | 24 | \x00
   8144 | 25 | 24 | \x01
(3 rows)
```

4) Look at how table rows with 2 -byte fixed-length columns of type `int 2` (synonym `smallint`) are stored:

```
postgres=# drop table if exists t5;
create table t5( c1 int2 default 1);
insert into t5 values(default);
insert into t5 values(2);
select lp, lp_off, lp_len, t_ctid, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
lp | lp_off | lp_len | t_ctid | t_hoff | t_data
----+--------+--------+--------+--------+--------
2 | 8112 | 26 | (0,2) | 24 | \x0200
1 | 8144 | 26 | (0,1) | 24 | \x0100
(2 rows)
```

String size: 8144-8112= 32 bytes
lp - slot number in block header

```
postgres=# drop table if exists t5;
create table t5(c1 int2, c2 int2, c3 int2, c4 int2);
insert into t5 values( 1 ,2, 3 ,4);
insert into t5 values(5,6,7,8);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
INSERT 0 1
INSERT 0 1
lp_off | lp_len | t_hoff | t_data
--------+--------+--------+-------------------
8112 | 32 | 24 | \ x0500060007000800
   8144 | 32 | 24 | \ x 0100 0200 0300 0400
(2 rows )
```

The string size will remain the same: 8144-8112= 32 bytes.

```
postgres=# drop table if exists t5;
create table t5(c1 int2, c2 int2, c3 int2, c4 int2, c5 int2 );
insert into t5 values( 1 ,2, 3 ,4, 5 );
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
INSERT 0 1
lp_off | lp_len | t_hoff | t_data
--------+--------+--------+-----------------------
   8136 | 34 | 24 | \ x 0100 0200 0300 0400 0500
(1 row )
```

Starting from the fifth column, the row size increased by 8 bytes at once: 8144-8136=8 bytes.
spaces (padding) between lines for storing data.
Conclusion from the example: it is better to create a table with 4 columns `( c 1 int 2, c 2 int 2, c 3 int 2, c 4 int 2)` than with one column `( c 1 int 2)` , since the size of the tables will be the same, and more data can be stored in the first table.
The example is useful to understand how to calculate the full size of the string from the values in `lp_off ;` as in the `t_data field` The values of the data types used are displayed.

Part 7. Storage of variable-length data types

length types have a negative number in the typlen column of the pg_type table : -1 denotes the varlena type .

1) Look at the characteristics of the text type:

```
postgres=# select * from pg_type where typname='text'\gx
-[ RECORD 1 ]--+---------
oid | 25
typename | text
typnamespace | 11
typowner | 10
type | -1 type variable long varlena
typbyval | f   for variable length types always f
type | b   base type, not composite, not interval
type category | S   string type
typispreferred | t   is preferred in the S category of string types
typisdefined | t
typdelim | , the separator of values of this type when parsing an array of values of this type
typrelid | 0
typsubscript | -
typelem | 0
typarray | 1009
typinput | textin
typooutput | textout
typreceive | textrecv name of the function to convert a binary value to this type
typsend | textsend name of the function to convert to binary format
typmodin | -
typmodout | -
typanalyze | -
typalign | i   aligned to 4 bytes
typstorage | x   by default the storage mode is EXTENDED (compression, then moving to TOAST )
typnotnull | f
typebasetype | 0
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
typemod | -1
typndims | 0
typocollation | 100
typdefaultbin |
typedefault |
typacl |
```

Description of columns of **the pg_type view** is in the documentation https://docs.tantorlabs.ru/tdb/ru/16_6/se/catalog-pg-type.html

2) Check if text fields are aligned if their length is up to 126 bytes:

```
postgres=# drop table if exists t;
create table t(c1 text, c2 text);
insert into t values (repeat('a',1), repeat('b',1));
insert into t values (repeat('a',2), repeat('b',2));
insert into t values (repeat('a',3), repeat('b',3));
insert into t values (repeat('a',4), repeat('b',4));
insert into t values (repeat('a',5), repeat('b',5));
insert into t values (repeat('a',6), repeat('b',6));
insert into t values (repeat('a',7), repeat('b',7));
insert into t values (repeat('a',8), repeat('b',8));
select pg_column_size(t.*), pg_column_size(c1) from t;
select lp_off, lp_len, t_data from heap_page_items(get_raw_page('t', 0));
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
pg_column_size | pg_column_size
----------------+----------------
28 | 2
30 | 3
32 | 4
34 | 5
36 | 6
38 | 7
40 | 8
42 | 9
(8 rows)

lp_off | lp_len | t_data
--------+--------+--------------------------------------
8144 | 28 | \x 05 61 05 62
8112 | 30 | \x 07 6161 07 6262
8080 | 32 | \x 09 616161 09 626262
8040 | 34 | \x 0b 61616161 0b 62626262
8000 | 36 | \x 0d 6161616161 0d 6262626262
7960 | 38 | \x 0f 616161616161 0f 626262626262
   7920 | 40 | \x 11 61616161616161 11 62626262626262
7872 | 42 | \x 13 6161616161616161 13 6262626262626262
(8 rows)
```

'61' is the character ' a ' in HEX format.
'62' is the character ' b ' in HEX format .
'00' (padding with zeros) between fields, which means that **there is no alignment for text type fields up to 127 bytes in size** .
The first byte stores the length (30 bits) and 2 bits: if the bits are 00, the length is less than 127 bytes. The value \x03 means an empty string; \x05 a string of 1 byte; \ x 07 a string of 2 bytes.
Storing the length is necessary to support TOAST. In a field of any varlena type, the first byte or first 4 bytes always (even if the field size is small and not TOASTed) contain the total length of the field in bytes (including these 4 bytes). Moreover, these bytes can (but not always) be compressed together

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group; Tantor Labs LLC © 2024 www.tantorlabs.ru

with the data, i.e. stored in a compressed form. One byte is used if the field length does not exceed 126 bytes. Therefore, when storing field data up to 127 bytes in size, three bytes are "saved" on each row version, and there is no alignment, which can save up to 7 bytes. varlena fields with one byte of length are not aligned, and fields with 4 bytes are aligned, which will be checked in the following points.

3) Check how many bytes text type fields are aligned if their length is 127 bytes or more:

```
postgres=# drop table if exists t;
create table t(c1 text, c2 text);
insert into t values (repeat('a',127), repeat('b',127));
insert into t values (repeat('a',128), repeat('b',128));
insert into t values (repeat('a',129), repeat('b',129));
insert into t values (repeat('a',130), repeat('b',130));
insert into t values (repeat('a',131), repeat('b',131));
insert into t values (repeat('a',132), repeat('b',132));
insert into t values (repeat('a',133), repeat('b',133));
insert into t values (repeat('a',134), repeat('b',134));
select pg_column_size(t.*), pg_column_size(c1) from t;
select lp_off, lp_len, substring(t_data from 130 for 18) from
heap_page_items(get_raw_page('t', 0));
DROP TABLE
CREATE TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
pg_column_size
----------------
pg_column_size | pg_column_size
----------------+----------------
287 | 131
288 | 132
293 | 133
294 | 134
295 | 135
296 | 136
301 | 137
302 | 138
(8 rows)

 lp_off | lp_len | substring
--------+--------+--------------------------------------
 7888 | 287 | \x 6161 00 0c020000 62626262626262626262
 7600 | 288 | \ x616161 10020000 62626262626262626262
 7304 | 293 | \x 61616161 000000 14020000 62626262626262
 7008 | 294 | \x 6161616161 0000 18020000 62626262626262
 6712 | 295 | \x 616161616161 00 1c020000 62626262626262
 6416 | 296 | \x 61616161616161 20020000 62626262626262
 6112 | 301 | \x 6161616161616161 000000 24020000 626262
 5808 | 302 | \x 616161616161616161 0000 28020000 626262
 (8 rows)
```

After the values '61' and before '62' there is padding (zeros) and 4 bytes of the header of the second (column c2) field.

The placeholder zeros are highlighted in red to ensure that the first field (column c 1) is aligned.

The maximum number of fillers is three, which means 4-byte alignment instead of 8-byte alignment.

`type` **text of 127 bytes or larger are aligned to 4 bytes** , which corresponds to the `value of pg_type.typalign = i .`

4) Check if the 4-byte alignment changes if you add int 8 as the third column, which is aligned to 8 bytes?

```
postgres=# drop table if exists t;
create table t(c1 text, c2 text, c3 int8);
insert into t values (repeat('a',127), repeat('b',127), 111);
insert into t values (repeat('a',128), repeat('b',128), 111);
insert into t values (repeat('a',129), repeat('b',129), 111);
insert into t values (repeat('a',130), repeat('b',130), 111);
insert into t values (repeat('a',131), repeat('b',131), 111);
insert into t values (repeat('a',132), repeat('b',132), 111);
insert into t values (repeat('a',133), repeat('b',133), 111);
insert into t values (repeat('a',134), repeat('b',134), 111);
select pg_column_size(t.*), pg_column_size(c1) from t;
select lp_off, lp_len, substring(t_data from 130 for 18) from
heap_page_items(get_raw_page('t', 0));
```

```
pg_column_size | pg_column_size
----------------+----------------
296 | 131
296 | 132
304 | 133
304 | 134
304 | 135
304 | 136
312 | 137
312 | 138
(8 rows)
```

```
lp_off | lp_len | substring
--------+--------+--------------------------------------
7880 | 296 | \x6161 00 0c020000 626262626262626262626262
7584 | 296 | \x616161 10020000 626262626262626262626262
7280 | 304 | \x61616161 000000 14020000 6262626262626262
6976 | 304 | \x6161616161 0000 18020000 6262626262626262
6672 | 304 | \x616161616161 00 1c020000 6262626262626262
6368 | 304 | \x61616161616161 20020000 6262626262626262
6056 | 312 | \x6161616161616161 000000 24020000 626262
5744 | 312 | \x616161616161616161 0000 28020000 626262
(8 rows )
```

Adding an int 8 column to the right did not affect the text alignment . Also, adding a 16-byte aligned column instead of an int8 column will not affect either the left, or the right, or both the left and the right:

```
create table t(c1 text, c2 text, c3 uuid default gen_random_uuid());
insert into t values (repeat('a',127), repeat('b',127), default);
create table t(c3 uuid default gen_random_uuid(), c1 text, c2 text);
insert into t values (default, repeat('a',127), repeat('b',127));
create table t(c3 uuid default gen_random_uuid(), c1 text, c2 text, c4 uuid
default gen_random_uuid());
insert into t values (default, repeat('a',127), repeat('b',127), default);
```

column c1 alignment will remain 4 bytes.

5) See how text fields placed in TOAST are stored in the table block and whether they are aligned:

```
postgres=# drop table if exists t;
create table t(c1 text storage external , c3 boolean , c2 text storage external
);
insert into t values (repeat('a',2005), false , '' );
insert into t values (repeat('a',2005), true , repeat('b',2001));
insert into t values (repeat('a',2005), false , repeat('b',2002));
select pg_column_size(t.*), pg_column_size(c1), pg_column_size(c3),
pg_column_size(c2) from t;
select lp_off, lp_len, t_data from heap_page_items(get_raw_page('t', 0));
```

```
DROP TABLE
```

```
CREATE TABLE
ALTER TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
INSERT 0 1
 pg_column_size | pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------+----------------
2035 | 2005 | 1 | 1
4041 | 2005 | 1 | 2001
4042 | 2005 | 1 | 2002
 (3 rows)

lp_off|lp_len|t_data
------+------+----------------------------------------------------------
8128| 44| \x 01 12 d9070000 d5070000 035b0800 015b0800 00 03
8064| 61| \x 01 12 d9070000 d5070000 045b0800 015b0800 01 01 12 d5070000 d1070000 055b0800
015b0800
8000| 61| \x 01 12 d9070000 d5070000 065b0800 015b0800 00 01 12 d6070000 d2070000 075b0800
015b0800
 (3 rows)
```

Fields Not were aligned .
The length of the fields included in TOAST is 18 bytes.
The pg_column_size (c1) function for fields included in TOAST returns the size of the data area of **this field .**
Let's look at what is stored in 18 bytes:
the first byte has the value 0x01 , this is a sign that the field is TOASTed;
the second byte is the length of this record (value 0x12 = 18 bytes);
4 bytes is the length of the field with the field header before compression;
4 bytes is the length of what is put into TOAST;
4 bytes - pointer to the first chunk in TOAST (chunk_id column of TOAST table);
4 bytes - oid of the toast- table (pg_class.reltoastrelid). The TOAST table is the same for all fields of a non-partitioned table.

6) Example of storing text fields in compressed form without taking them out in TOAST :

```
postgres=# drop table if exists t;
create table t (c1 text storage main, c3 boolean , c2 text storage main);
insert into t values (repeat('a',2003), false , '');
insert into t values (repeat('a',1984), true , repeat('b',1984));
select pg_column_size(t.*), pg_column_size(c1), pg_column_size(c3),
pg_column_size(c2) from t;
select lp_off, lp_len, t_data from heap_page_items(get_raw_page('t', 0))\gx
DROP TABLE
CREATE TABLE
ALTER TABLE
ALTER TABLE
INSERT 0 1
INSERT 0 1
pg_column_size | pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------+----------------
65 | 39 | 1 | 1
103 | 39 | 1 | 39
 (2 rows)

-[ RECORD 1]-
lp_off | 8104
lp_len | 65
t_data | \x9e000000d3070000fe610f01ff0f01ff0f01ff0f01ff0f01ff0f01ff0f01ff010f064561616161 00 03
-[ RECORD 2]-
lp_off | 8000
lp_len | 103
t_data | \x 9e000000 c0070000fe610f01ff0f01ff0f01ff0f01ff0f01ff0f01ff0f01ff010f083261616161 01
           9e000000 c0070000fe620f01ff0f01ff0f01ff0f01ff0f01ff0f01ff0f01ff010f083262626262
```

7) Storing Cyrillic. Create a table and insert a row with a Cyrillic character:

```
drop table if exists t5;
```

```
create table t5(c1 text default ' 1 ',c2 text default ' е ', c3 text default ''
);
insert into t5 values(default, default, default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
DROP TABLE
CREATE TABLE
INSERT 0 1
lp_off | lp_len | t_hoff | t_data
--------+--------+--------+----------------
8144 | 30 | 24 | \ x 05 31 07 d 18 d 03
(1 row )
```

Cyrillic character takes 2 bytes : `d 18 d`
An ASCII character takes up 1 byte.
The first byte in each field (`05 07 03`) stores the size of the field.

Part 8. Data type for the primary key column

1) Comparison of using bigint and uuid as primary key:

```
postgres@tantor:~$
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity ( cache 60 )
primary key, data bigint);"
echo "insert into tt1(data) values(1);" >txn.sql
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data bigint);"
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from tt1;"

DROP TABLE
CREATE TABLE
VACUUM
tps = 556 .855615 (without initial connection time)
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
16698 | 589824 | 786432
(1 row)

REINDEX
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
16698 | 393216 | 786432
(1 row)

DROP TABLE
CREATE TABLE
VACUUM
tps = 553 .336293 (without initial connection time)
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
16593 | 622592 | 901120
(1 row)
```

```
REINDEX
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
16593 | 540672 | 901120
(1 row )
```

The insertion speed is comparable. Inserting into the index using uuid is less efficient, the index size is larger.

After rebuilding the indexes, their size decreased. Because of the `cache=60` parameter , insertion into the index was inefficient . Some insertions were not made into the right block because parallel processes (parameter `-c=4`) managed to split the right block, and other processes cached values that were forced to be inserted into more left blocks.

2) Reduce cache to default value cache=1 and repeat test:

```
postgres@tantor:~$
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity ( cache 1
) primary key, data bigint);"
echo "insert into tt1(data) values(1);" >txn.sql
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data
bigint);"
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
psql -c "reindex table tt1;"
psql -c "select count(*), pg_indexes_size('tt1'), pg_table_size('tt1') from
tt1;"
DROP TABLE
CREATE TABLE
VACUUM
tps = 622 .324117 (without initial connection time)
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
 18664 | 434176 | 868352
(1 row)

REINDEX
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
18664 | 434176 | 868352
(1 row)

DROP TABLE
CREATE TABLE
VACUUM
tps = 639 .807107 (without initial connection time)
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
 19185 | 778240 | 1048576
(1 row)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
REINDEX
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
19185 | 622592 | 1048576
(1 row)
```

Index on **bigint column** remained effective, since the size of the index did not decrease after the restructuring.
With a smaller number of rows, the size of the index on a uuid column is significantly larger than the size of the index on a bigint column.
The reason is that the size of the uuid field is 16 bytes (128 bits, a size convenient for use as an encryption key) is twice the size of the bigint field (8 bytes).
Sequences should be used to generate unique values for the primary key. Other formats can be used for business purposes. For example, "booking reference" (booking), which uses ~30 characters (uppercase alphabet ABCDEFGHJKMNPQRSTUXY and numbers).

Part 9. Data types for storing dates and times

The timestamp and timestamptz data types store time and date with microsecond precision and occupy 8 bytes. Both types do not store the time zone, the values are physically stored in the same form.

1) The timestamptz data type displays and performs calculations in the time zone specified by the timezone parameter. See parameter timezone configurations :

```
postgres=# show timezone;
TimeZone
---------------
Europe/Moscow
(1 row)
```

2) Create a table with columns of types timestamp and timestamptz:

```
postgres=# drop table if exists t;
create table t(t TIMESTAMP, ttz TIMESTAMPTZ);
insert into t values (CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
select t, ttz from t;
DROP TABLE
CREATE TABLE
INSERT 0 1
t | tts
--------------------------+-----------------------------
2025-01-02 20:00:24.87259 | 2025-01-02 20:00:24.87259+ 03
(1 row )
```

3) Change the timezone parameter at the session level:

```
postgres=# set timezone='UTC';
select t, ttz from t;
SET
             t | tts
--------------------------+-----------------------------
2025-01-02 20:00:24.87259 | 2025-01-02 17 :00:24.87259+ 00
(1 row )
```

The displayed value in a column of type timestamptz has changed.

4) Look at the field values in the table block:

```
postgres=# select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) order by lp_off;
 lp_off | lp_len | t_hoff | t_data
--------+--------+--------+----------------------------------------------------
 8136 | 40 | 24 | \x8e36b061bdcd0200 8e4af5ddbacd0200
 (1 row)
```

The values are physically different.

5) Make the field values equal by assigning the value from the t column to the value of the ttz column and see how the value in ttz has changed:

```
postgres=# update t set ttz =t;
select lp, lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t','main',0)) order by lp_off;
UPDATE 1
lp | lp_off | lp_len | t_hoff | t_data
---+--------+--------+--------+-------------------------------
2 | 8096 | 40 | 24 | \x8e36b061bdcd0200 8e36b061bdcd0200
1 | 8136 | 40 | 24 | \x8e36b061bdcd0200 8e4af5ddbacd0200
 (2 rows)
```

Values V fields both columns V current versions lines (`lp=2`) became physically are equal .

6) Now the values in the columns are physically equal. Change the timezone from UTC to the original and see the result:

```
postgres=# select t, ttz from t;
t | tts
--------------------------+-----------------------------
2025-01-02 20:00:24.87259 | 2025-01-02 20:00:24.87259+00
 (1 row)

postgres=# RESET timezone;
RESET
postgres=# select t, ttz from t;
             t | tts
--------------------------+-----------------------------
2025-01-02 20:00:24.87259 | 2025-01-02 23 :00:24.87259+ 03
 (1 row)
```

timestamptz physically stores values in UTC .
The timestamp data type does not display a time zone, does not use a time zone, and stores the value as is (without conversion).

Part 10. Data types float and real

1) Numbers can be output with powers of 10. The power is indicated after the signs " e +". In the example, the values are the same:

```
postgres =# select 12345.6:: float 4, '12.3456 e +03':: float 4, '123.456 e
+02':: float 4, '1234.56 e +01':: float 4;
  float 4 | float 4 | float 4 | float 4
---------+---------+---------+---------
  12345.6 | 12345.6 | 12345.6 | 12345.6
 (1 row)
```

2) The precision of float8 is 15 decimal places (significant numbers in the decimal number system).

float4 precision may not be enough: 6 decimal places
Last discharge rounded off :

```
postgres=# select 123456789012345678901234567890.123456789012345678901234567890::float4::numeric;
Numeric
--------------------------------
 123457 000000000000000000000000
(1 row)

postgres=# select 123456789012345678901234567890.123456789012345678901234567890::float8::numeric;
Numeric
--------------------------------
 123456789012346 00000000000000
(1 row )
```

3) You shouldn't rely on digits greater than 15 and 6, although they are stored (including in table rows) and can be taken into account in calculations. An example of displaying additional digits:

```
postgres =# select 123456789012345678901234567890.123456789012345678901234567890:: float 4;
    float 4
---------------
1.23456 79 e +28
(1 row )

postgres=# select 123456789012345678901234567890.123456789012345678901234567890::float8;
float8
------------------------
 1.23456789012345 68 e+28
(1 row )
```

4) Example of rounding when calculating with float8 (synonym float) and float4 (synonym real) data types:

```
postgres=# select 234567890.199999 98 9::float8, 1.19999 99 9123::float4;
float8 | float4
-------------+--------
234567890.2 | 1.2
(1 row)

postgres=# select 1234567890.12345 67 89::float8, 1.12345 67 89::float4;
float8 | float4
---------------------+-----------
 1234567890.12345 67 | 1.12345 68
(1 row)

postgres=# select 234567890.199999989::float8::numeric,
1.19999999123::float4::numeric;
Numeric | Numeric
-------------+---------
234567890.2 | 1.2
(1 row)

postgres=# select 1234567890.123456789::float8::numeric,
1.123456789::float4::numeric;
     Numeric | Numeric
-----------------+---------
 1234567890.12346 | 1.12346
(1 row)
```

5) The disadvantage of float4 and float8 is that adding a small number to a large number is equivalent to adding zero, which can cause difficult to diagnose errors in applications:

```
postgres=# select (12345678901234567890123456789.1234567890123456789::float8 +
123456789::float8 );
 ?column?
------------------------
 1.23456789012345 68 e+28
(1 row)

postgres=# select (12345678901234567890123456789.1234567890123456789::float8 +
123456789::float8)::numeric;
            Numeric
-------------------------------
 12345678901234 6 00000000000000
(1 row)
```

15th digit was rounded when cast to numeric type.
Because of the loss of precision, float4 and float8 are not suitable for applications where precision must be maintained.

6) Look at how float 8, float 4, numeric types are stored :

```
postgres=# drop table if exists t5;
create table t5( c1 double precision , c2 real , c3 numeric );
insert into t5 values
(1,1,1),
(1.0/3, 1.0/3, 1.0/3),
(1111,1111,1111),
(1111.11,1111.11, 1111.11);
select lp, lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp;
select pg_column_size(c1), pg_column_size(c2), pg_column_size(c3) from t5;
select * from t5;
DROP TABLE
CREATE TABLE
INSERT 0 4
lp | lp_off | lp_len | t_hoff | t_data
----+--------+--------+--------+-------------------------------------------------------------
---------
1 | 8128 | 41 | 24 | \x 000000000000f03f 0000803f 0b00800100
2 | 8072 | 49 | 24 | \x 555555555555d53f abaaaa3e 1b7f8a050d050d050d050d
3 | 8024 | 41 | 24 | \x 00000000005c9140 00e08a44 0b00805704
4 | 7976 | 43 | 24 | \x 3d0ad7a3705c9140 85e38a44 0f008157044c04

pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------
              8 | 4 | 5
              8 | 4 | 13
              8 | 4 | 5
             8 | 4 | 7
(4 rows)

c1 | c2 | c3
--------------------+-----------+----------------------
1 | 1 | 1
0. 333333333333333 3 | 0. 333333 34 | 0. 3333333333333333 3333
1111 | 1111 | 1111
1111.11 | 1111.11 | 1111.11
(4 rows)
```

When choosing a data type for storing real numbers, it is worth considering that the numeric type has a variable length and stores data more compactly for small numbers than float 8: in the example, these are all the lines except the second.
float 4 stores data compactly and with a fixed width, but this type has only 6 bits.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

numeric type is huge: 131072 digits before the decimal point and 16383 after. Fields of the numeric type can be displaced in TOA S T.

Why, as a result of dividing `1.0/3,` the digit capacity of the resulting number, although larger than float8, is still not large?

7) Does the division operator work with numeric operands? See the list of division operators:

```
postgres =# \ doS +/
                                               List of operators
Schema | Name | Left arg type | Right arg type | Result type | Function | Description
------------+------+-----------------+-----------------+------------------+--------------+-------------
pg_catalog | / | bigint | bigint | bigint | int8div | divide
pg_catalog | / | bigint | integer | bigint | int84div | divide
pg_catalog | / | bigint | smallint | bigint | int82div | divide
pg_catalog | / | box | point | box | box_div | divide box by point (scale)
pg_catalog | / | circle | point | circle | circle_div_pt | divide
pg_catalog | / | double precision | double precision | double precision | float8div | divide
pg_catalog | / | double precision | real | double precision | float84div | divide
pg_catalog | / | integer | bigint | bigint | int48div | divide
pg_catalog | / | integer | integer | integer | int4div | divide
pg_catalog | / | integer | smallint | integer | int42div | divide
pg_catalog | / | interval | double precision | interval | interval_div | divide
pg_catalog | / | money | bigint | money | cash_div_int8 | divide
pg_catalog | / | money | double precision | money | cash_div_flt8 | divide
pg_catalog | / | money | integer | money | cash_div_int4 | divide
pg_catalog | / | money | money | double precision | cash_div_cash | divide
pg_catalog | / | money | real | money | cash_div_flt4 | divide
pg_catalog | / | money | smallint | money | cash_div_int2 | divide
 pg_catalog | / | numeric | numeric | numeric | numeric_div | divide
pg_catalog | / | path | point | path | path_div_pt | divide (rotate/scale path)
 pg_catalog | / | point | point | point | point_div | divide points (scale/rotate)
 pg_catalog | / | real | double precision | double precision | float48div | divide
pg_catalog | / | real | real | real | float4div | divide
pg_catalog | / | smallint | bigint | bigint | int28div | divide
pg_catalog | / | smallint | integer | integer | int24div | divide
pg_catalog | / | smallint | smallint | smallint | int2div | divide
(25 rows )
```

The " / " operator works with operands of type `numeric` and is implemented by the `numeric_div(..) function` .

8) Look at the body of the `numeric_div(..) function` :

```
postgres=# \sf numeric_div
CREATE OR REPLACE FUNCTION pg_catalog.numeric_div(numeric, numeric)
RETURNS numeric
 LANGUAGE internal
IMMUTABLE PARALLEL SAFE STRICT
AS $function$numeric_div$function$
```

Function implemented V PostgreSQL code , then implementation functions need to look V original PostgreSQL code (functions `numeric_div -> numeric_div_opt_error -> select_div_scale` V file `numeric.c`).

The number of significant digits returned by the `numeric_div(..) function` :

not less than `NUMERIC_MIN_SIG_DIGITS=16` significant digits, i.e. not worse than float8;

not less than the bit depth of any of the input parameters.

9) Check the statement " **not less than the display scale of any of the input parameters** ":

```
postgres =# select 1/1.0000000000000000000000000000000000001;
? column ?
-----------------------------------------
0.99999999999999999999999999999999999999
(1 row )
```

In this example, the result's bit width is greater than 16 and is the same as that of the right operand.

10) When inserting into a table, the same rule applies:

```
postgres=# insert into t5 values (1,1,
1.000000000000000000000000000000000001/3);
select lp, lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp desc limit 1;
select * from t5 order by ctid desc limit 1;
INSERT 0 1
lp | lp_off | lp_len | t_hoff | t_data
----+--------+--------+--------+------------------------------------------------------------------------
5 | 7912 | 57 | 24 | \x 000000000000f03f 0000803f 2b7f92050d050d050d050d050d050d050d050d060d
(1 row)

c1 | c2 | c3
----+----+----------------------------------------
  1 | 1 | 0.33333333333333333333333333333333334
(1 row )
```

The statement is true: the bit depth of the result of division of two numeric types is determined **by the largest bit depth of the operands** , but not less than 16.

Practice for Chapter 9

Part 1. Using PostgreSQL in a docker container

1) Open terminal and switch to root .

```
astra@tantor:~$ su -
Password: root
root @tantor:~#
```

To be able to run commands in the terminal of users postgres and astra without using sudo and without switching to root, add users astra and postgres to a group called docker :

```
root@tantor:~# sudo usermod -aG docker postgres
sudo usermod - aG docker astra
```

The commands in this part of the practice can be executed in the terminal of root or postgres or astra users.

2) Check the list of available images in the docker .io repository :

```
root@tantor:~# docker search --filter stars=10 --no-trunc postgres
NAME DESCRIPTION STARS OFFICIAL
postgres The PostgreSQL object-relational database sy… 14016 [OK]
circleci/postgres The PostgreSQL object-relational database sy… 32
ubuntu/postgres PostgreSQL is an open source object-relation… 40
supabase/postgres Unmodified Postgres with some useful plugins… 44
debezium/postgres PostgreSQL for use with Debezium change data… 28
```

The official image name is postgres .
Image description page: https :// hub . docker . com /_/ postgres

3) Check if there is a postgres image

```
root @ tantor :~# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
postgres latest de869b203456 2 days ago 494MB
```

4) If the list contains a `postgres image` and the Internet access in the virtual machine is limited, then you do not need to download a new image.
If the list is empty, download the image:

```
root@tantor:~# docker pull postgres
Using default tag: latest
latest: Pulling from library/postgres
...
Status: Downloaded newer image for postgres:latest
docker.io/library/postgres : latest
```

Was downloaded most fresh image `postgres` .

The image name has the format:
image [: TAG][@ DIGEST]
where tag is the image version, by default latest .

5) The command format for creating and running a container from an image:
`docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]`

ENTRYPOINT is specified in the image , COMMAND will be added to the ENTRYPOINT parameters.

Run the following command to create a container:

```
root@tantor:~# docker run -d -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=postgres -e POSTGRES_INITDB_ARGS="-k" -e
POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e PGDATA=/var/lib/postgresql/data -
d -v /root/data:/var/lib/postgresql/data --name postgres postgres
```

Description of command parameters:
- d (-- deatach) running container in background
-- name postgres set container name
- e environment variables
- e POSTGRES_USER = postgres database cluster user name
- e POSTGRES_PASSWORD = postgres set password for database cluster user
- e POSTGRES_INITDB_ARGS ="- k " parameters to the initdb utility if a cluster will be created
- e POSTGRES_HOST_AUTH_METHOD = trust default authentication method
- v mount directory / var / lib / postgresql / data inside container to directory / root / data
- p (-- publish) mapping ports from container to host
last parameter: image name

6) To check that the container has started successfully, run the command:

```
root@tantor:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
75db63ec0558 postgres "docker-entrypoint.s…" 3 minutes ago Up 3 minutes 5432/tcp,... postgres
```

In the output you will see a list of all running containers.
The list should contain a container named postgres with a status of Up .
The command is useful because it outputs CONTAINER ID = 75 db 63 ec 0558 , the value of which can be used in docker utility commands instead of the postgres container name .

Command to view container ports:

```
root@tantor:~# docker port postgres
5434/tcp -> 0.0.0.0:5434
5434/ tcp -> [::]:5434
```

the same result can be obtained using CONTAINER ID :

```
root@tantor:~# docker port 75db63ec0558
5434/tcp -> 0.0.0.0:5434
5434/ tcp -> [::]:5434
```

Command to start terminal in container:

```
root@tantor:~# docker exec -it postgres bash
```

Command to stop container:
```
root @ tantor :~# docker stop postgres
```

Command to start container:
```
root @ tantor :~# docker st art postgres
```

7) The instance in the container does not keep diagnostic logs. This is useful to avoid running out of space due to log file growth. By default, the log_destination=stderr configuration

tantor

parameter and the log is sent to the standard error stream, which is saved and cleared by docker. The command For views stderr :

```
root@tantor:~# docker container logs postgres

PostgreSQL Database directory appears to contain a database; Skipping initialization

08:04:07.353 UTC [1] LOG: starting PostgreSQL 17.3 (Debian 17.3-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
08:04:07.353 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
08:04:07.353 UTC [1] LOG: listening on IPv6 address "::", port 5432
08:04:07.368 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
08:04:07.387 UTC [29] LOG: database system was interrupted; last known up at 08:00:29 UTC
08:04:09.351 UTC [29] LOG: database system was not properly shut down; automatic recovery in progress
08:04:09.361 UTC [29] LOG: redo starts at 0/177D060
08:04:09.361 UTC [29] LOG: invalid record length at 0/177D098: expected at least 24, got 0
08:04:09.361 UTC [29] LOG: redo done at 0/177D060 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed:
0.00 s
08:04:09.378 UTC [27] LOG: checkpoint starting: end-of-recovery immediate wait
08:04:09.428 UTC [27] LOG: checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.016 s, sync=0.007 s, total=0.058 s; sync files=2, longest=0.004 s, average=0.004 s; distance=0
kB, estimate=0 kB; lsn=0/177D098, redo lsn=0/177D098
08:04:09.439 UTC [1] LOG: database system is ready to accept connections
```

You can connect to the output and watch the messages appear using the --follow or -f option , and you can use the --tail option to limit the number of lines output :

```
root@tantor:~# docker container logs -f postgres --tail 1
08:04:09.439 UTC [1] LOG: database system is ready to accept connections
^ c
```

To return the prompt, simply type < ctrl + c > on the keyboard .

8) To learn how to change configuration parameters, enable the creation of a cluster diagnostic log and restart the container:

```
root@tantor:~# docker exec -it postgres psql -U postgres -c "alter system set
logging_collector=on"
ALTER SYSTEM
root@tantor:~# docker restart postgres
postgres
```

While the container is not in operation, you can use the simplicity of Docker: quickly restart the container with a simple command .

9) There are no diagnostic commands in the container. For example, there is no ps command. Install the ps utility in the container , which can view lists of processes and their PIDs:

```
root@tantor:~# docker exec -it postgres apt update
root@tantor:~# docker exec -it postgres apt install -y procps
```

10) Look at the PID of the postgres process:

```
root@tantor:~# docker exec -it postgres /usr/bin/ps -ef
UID PID PPID C STIME TTY TIME CMD
postgres 1 0 0 07:11 pts/0 00:00:00 postgres
postgres 27 1 0 21:23 ? 00:00:00 postgres: logger
postgres 28 1 0 21:23 ? 00:00:00 postgres: checkpointer
postgres 29 1 0 21:23 ? 00:00:00 postgres: background writer
postgres 31 1 0 21:23 ? 00:00:00 postgres: walwriter
postgres 32 1 0 21:23 ? 00:00:00 postgres: autovacuum launcher
postgres 33 1 0 21:23 ? 00:00:00 postgres: logical replication launcher
root 34 0 66 21:23 pts/0 00:00:00 ps -ef
```

The main process of the instance has **PID=1** . **This is undesirable for production instances.** It is difficult to understand or believe why this is unacceptable. Let's see that the instance can easily crash and restart.

11) Launch the bash terminal in the container:

```
root@tantor:~# docker exec -it postgres bash
root@1d7f295f6f60:/#
```

12) In the running terminal, run psql and connect to the database:

```
root@1d7f295f6f60:/# psql -U postgres
psql (17.2 (Debian 17.2-1.pgdg120+1))
Type "help" for help.

postgres=#
```

13) Next you need to send psql to the background and close the terminal. Type the key combinations **< ctrl + z > < ctrl + d > < ctrl + d > on the keyboard** :

```
postgres=# <ctrl+z>
[1]+ Stopped psql -U postgres
root@1d7f295f6f60:/# <ctrl+d>
exit
There are stopped jobs.
root@1d7f295f6f60:/# <ctrl+d>
exit

root@tantor:~#
```

The terminal was closed, the psql process was orphaned.

14) Look at the PID of the checkpointer process and other processes in the instance:

```
root@tantor:~# docker exec -it postgres /usr/bin/ps -ef
UID PID PPID C STIME TTY TIME CMD
postgres 1 0 0 07:11 pts/0 00:00:00 postgres
postgres 27 1 0 21:23 ? 00:00:00 postgres: logger
postgres 50 1 0 21:24 ? 00:00:00 postgres: checkpointer
postgres 51 1 0 21:24 ? 00:00:00 postgres: background writer
postgres 52 1 0 21:24 ? 00:00:00 postgres: walwriter
postgres 53 1 0 21:24 ? 00:00:00 postgres: autovacuum launcher
postgres 54 1 0 21:24 ? 00:00:00 postgres: logical replication launcher
root 55 0 60 21:24 pts /0 00:00:00 ps - ef
```

The instance processes have changed numbers, which means they have been restarted. The postgres process restarted the instance .

The actions to start and stop the `psql utility` in a container should not cause the instance to restart. The sequence to stop `psql` that was used, **< ctrl + z > < ctrl + d > < ctrl + d >** is one example. Any program or process inside the container can crash on its own. If the parent process of a process disappears, then its parent becomes process number 1. The appearance of a child process unknown to the postgres process causes a forced restart of the instance.

In the absence of ps utilities (installed separately), not knowing what to look for in the diagnostic log (change in instance process numbers), it is difficult to detect instance restarts in containers.

15) The fact that the instance has restarted can be seen in the diagnostic log:

```
root@tantor:~# tail -20 ~/data/log/postgresql-* .log
21:23:03 .887 UTC [1] LOG: starting PostgreSQL 17.2 (Debian 17.2-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
21:23:03.887 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
 21:23:03.887 UTC [1] LOG: listening on IPv6 address "::", port 5432
 21:23:03.906 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
 21:23:03.927 UTC [30] LOG: database system was shut down at 21:22:59 UTC
 21:23:03.944 UTC [1] LOG: database system is ready to accept connections
 21:24:06.851 UTC [1] LOG: server process (PID 46) was terminated by signal 15: Terminated
 21:24:06.851 UTC [1] LOG: terminating any other active server processes
 21:24:06.862 UTC [1] LOG: all server processes terminated; reinitializing
 21:24:06.888 UTC [49] LOG: database system was interrupted; last known up at 21:23:03 UTC
 21:24:08.858 UTC [49] LOG: database system was not properly shut down; automatic recovery in progress
 21:24:08.869 UTC [49] LOG: redo starts at 0/177D760
 21:24:08.869 UTC [49] LOG: invalid record length at 0/177D798: expected at least 24, got 0
 21:24:08.869 UTC [49] LOG: redo done at 0/177D760 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed:
0.00 s
 21:24:08.892 UTC [50] LOG: checkpoint starting: end-of-recovery immediate wait
 21:24:08.962 UTC [50] LOG: checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.016 s, sync=0.006 s, total=0.081 s; sync files=2, longest=0.004 s, average=0.003 s; distance=0
kB, estimate=0 kB; lsn=0/177D798, redo lsn=0/177D798
 21:24:08.973 UTC [1] LOG: database system is ready to accept connections
```

or by command :

```
root@tantor:~# docker container logs postgres --tail 20
```

Next, we'll look at how to create and run a docker container so that the PID is not equal to 1 .

15) Stop and delete the created container:

```
root@tantor:~# docker rm -f postgres
docker network prune -f
docker volume prune -f
docker network create postgres

postgres
Total reclaimed space: 0B
b4785b7d1814 d39fb2ca92de80f5e248fffe22ce0181095ceedfc1baf20c7d3d
```

Long HEX value - identifier object in docker.
Team **docker network prune -f** remove unused containers networks . No such networks were created.
Team **docker volume prune -f** delete unused containers volumes .
Team **docker network create postgres** creates network . IP address is generated automatically.

```
root @ tantor :~# ifconfig | head -2
br- b4785b7d1814 : flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
inet 172.18.0. 1 netmask 255.255.0.0 broadcast 172.27.255.255
```

16) Create container With option **-- init** :

```
root@tantor:~# docker run -d --init  --network postgres -e DB_HOST=127.0.0.1 -e
POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -e POSTGRES_INITDB_ARGS="--
data-checksums" -e POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e
PGDATA=/var/lib/postgresql/data -d -v /root/data:/var/lib/postgresql/data --name
postgres postgres
```

You can use the **--init** flag to indicate that an init process should be used as the PID 1 in the container. Specifying an init process ensures the usual responsibilities of an init system, such as reaping zombie processes, are performed inside the created container. The default init process used is the first docker-init executable found in the system path of the Docker daemon process. This docker-init binary, included in the default installation, is backed by `tini` .

17) Look at the IP address assigned to the created network that was assigned to the container:

```
root@tantor:~# docker inspect postgres | grep 172
                    " Gateway ": " 172.18.0.1 ",
 " IPAddress ": " 172.18.0.2 ",
```

18) You can connect to an instance in a container from the host using the container network IP address. To do this, you need to specify the container IP address and the port inside the container, which is port 5432, not 5434 :

```
root@tantor:~# psql -h 172.18.0.2 -p 5432 -U postgres -c "show port; select version();"
port
------
5432
(1 row)
version
---------------------------------------------------------------------------------
PostgreSQL 17.2 (Debian 17.2-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled by gcc
(Debian 12.2.0-14) 12.2.0, 64-bit
(1 row )
```

This point illustrates that it is not necessary to use **sudo docker commands. exe with** to connect to an instance in a container. You can connect to an instance in a container by network (ip) address if the container has been assigned a network.

19) Install the ps utility and look at the PID of the postgres process:

```
root@tantor:~# docker exec -it postgres apt update
docker exec -it postgres apt install -y procps
docker exec -it postgres /usr/bin/ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 21:27 ? 00:00:00 /sbin/docker-init -- docker-entrypoint.sh postgres
postgres 7 1 0 21:27 ? 00:00:00 postgres
postgres 28 7 0 21:27 ? 00:00:00 postgres: logger
postgres 29 7 0 21:27 ? 00:00:00 postgres: checkpointer
postgres 30 7 0 21:27 ? 00:00:00 postgres: background writer
postgres 32 7 0 21:27 ? 00:00:00 postgres: walwriter
postgres 33 7 0 21:27 ? 00:00:00 postgres: autovacuum launcher
postgres 34 7 0 21:27 ? 00:00:00 postgres: logical replication launcher
root 253 0 50 21:28 pts/0 00:00:00 /usr/bin/ps -ef
```

The main process of the instance has `PID=7` .
Number process 1 y process `/sbin/docker-init -- docker-entrypoint.sh postgres`

The container has ENTRYPOINT = `docker-entrypoint.sh and the parameter` we passed when creating the container became the `parameter passed to the docker-entrypoint.sh script` .

The script is located inside the container in the **/usr/local/bin directory** :

```
root@tantor:~# docker exec -it postgres ls /usr/local/bin
docker-enforce-initdb.sh docker-ensure-initdb.sh docker-entrypoint.sh gosu
```

20) Repeat the steps that caused the instance to restart:

```
root@tantor:~# docker exec -it postgres bash
root@1d7f295f6f60:/# psql -U postgres
psql (17.2 (Debian 17.2-1.pgdg120+1))
Type "help" for help.

postgres=# <ctrl+z>
[1]+ Stopped psql -U postgres
root@1d7f295f6f60:/# <ctrl+d>
exit
There are stopped jobs.
root@1d7f295f6f60:/# <ctrl+d>
exit
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
root@tantor:~#
```

21) Look at the PID of the checkpointer process and other processes in the instance:

```
root@tantor:~# docker exec -it postgres /usr/bin/ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 21:27 ? 00:00:00 /sbin/docker-init -- docker-entrypoint.sh postgres
postgres 7 1 0 21:27 ? 00:00:00 postgres
postgres 28 7 0 21:27 ? 00:00:00 postgres: logger
postgres 29 7 0 21:27 ? 00:00:00 postgres: checkpointer
postgres 30 7 0 21:27 ? 00:00:00 postgres: background writer
postgres 32 7 0 21:27 ? 00:00:00 postgres: walwriter
postgres 33 7 0 21:27 ? 00:00:00 postgres: autovacuum launcher
postgres 34 7 0 21:27 ? 00:00:00 postgres: logical replication launcher
root 266 0 37 21:29 pts/0 00:00:00 /usr/bin/ps -ef
```

The process numbers did not change, the instance was not restarted.

22) There are no entries about process restarts in the diagnostic log:

```
root@tantor:~# cat ~/data/log/postgresql- <TAB> _ 212736 .log
21:27:36 .139 UTC [7] LOG: starting PostgreSQL 17.2 (Debian 17.2-1.pgdg120+1) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 12.2.0-14) 12.2.0, 64-bit
21:27:36.139 UTC [7] LOG: listening on IPv4 address "0.0.0.0", port 5432
21:27:36.139 UTC [7] LOG: listening on IPv6 address "::", port 5432
21:27:36.161 UTC [7] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
21:27:36.183 UTC [31] LOG: database system was interrupted; last known up at 21:24:08 UTC
21:27:38.136 UTC [31] LOG: database system was not properly shut down; automatic recovery in progress
21:27:38.144 UTC [31] LOG: redo starts at 0/177D810
21:27:38.144 UTC [31] LOG: invalid record length at 0/177D848: expected at least 24, got 0
21:27:38.144 UTC [31] LOG: redo done at 0/177D810 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed:
0.00 s
21:27:38.165 UTC [29] LOG: checkpoint starting: end-of-recovery immediate wait
21:27:38.246 UTC [29] LOG: checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.034 s, sync=0.008 s, total=0.094 s; sync files=2, longest=0.004 s, average=0.004 s; distance=0
kB, estimate=0 kB; lsn=0/177D848, redo lsn=0/177D848
21:27:38.257 UTC [7] LOG: database system is ready to accept connections
```

The instance did not restart.

23) To test the performance of PostgreSQL running in a docker container, run the test from Part 7 of the previous practice:

```
root@tantor:~# sudo docker exec -it postgres bash
root@328a75c4c4c4:/# export PGUSER=postgres
root@328a75c4c4c4:/# psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity (cache 1)
primary key, data bigint);"
echo "insert into tt1(data) values(1);" >txn.sql
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id uuid default gen_random_uuid() primary key, data
bigint);"
psql -c "vacuum analyze tt1;"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
NOTICE: table "tt1" does not exist, skipping
DROP TABLE
CREATE TABLE
VACUUM
tps = 551 .864768 (without initial connection time)
DROP TABLE
CREATE TABLE
VACUUM
tps = 542 .598808 (without initial connection time)
root@328a75c4c4c4:/# exit
exit
```

For primary key type biginteger `tps = 551` , for uuid `tps = 542` .

In test 7 of the previous practice the results were: biginteger `tps = 556` , for uuid `tps = 553` .
The test results are similar.

24) Stop the container to free up memory:

```
root @ tantor :~# docker stop postgres
```

25) The PGDATA directory of the cluster inside the container was mapped to the host directory `/root/data` and is available after stopping the container:

```
root@tantor:~# ls -al /root/data
total 136
drwx------ 19 dnsmasq root 4096 13:17 .
drwxr-x--- 18 root root 4096 12:54 ..
drwx------ 5 dnsmasq systemd-journal 4096 12:31 base
drwx------ 2 dnsmasq systemd-journal 4096 12:44 global
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_commit_ts
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_dynshmem
-rw------- 1 dnsmasq systemd-journal 5839 12:31 pg_hba.conf
-rw------- 1 dnsmasq systemd-journal 2640 12:31 pg_ident.conf
drwx------ 4 dnsmasq systemd-journal 4096 13:22 pg_logical
drwx------ 4 dnsmasq systemd-journal 4096 12:31 pg_multixact
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_notify
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_replslot
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_serial
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_snapshots
drwx------ 2 dnsmasq systemd-journal 4096 13:17 pg_stat
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_stat_tmp
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_subtrans
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_tblspc
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_twophase
-rw------- 1 dnsmasq systemd-journal 3 12:31 PG_VERSION
drwx------ 4 dnsmasq systemd-journal 4096 12:31 pg_wal
drwx------ 2 dnsmasq systemd-journal 4096 12:31 pg_xact
-rw------- 1 dnsmasq systemd-journal 88 12:31 postgresql.auto.conf
-rw------- 1 dnsmasq systemd-journal 30777 12:31 postgresql.conf
-rw------- 1 dnsmasq systemd-journal 36 13:17 postmaster.opts
-rw------- 1 dnsmasq systemd-journal 94 13:17 postmaster.pid
```

On the host, you can edit configuration files and copy the directory for backup purposes.
If you delete the container, the directory contents will not be deleted. When creating a container with the parameters **`PGDATA = / var / lib / postgresql / data - d - v / root / data : / var / lib / postgresql / data`** the initdb utility will not recreate the cluster and the instance will start using the directory contents. This is convenient for copying PGDATA from the host to the container.

26) If the container is not working or is not needed, then you can delete the container and directory:

```
root@tantor:~# docker rm -f postgres
postgres
root@tantor:~# rm -rf /root/data
root@tantor:~# exit
logout
astra @ tantor :~$ exit
```

If the directory is not deleted, it will be used along with its contents when creating a new container.
If desired, the container can be created again. The simplicity of creating containers is the main advantage of docker.

24) Check if there were any warnings or errors related to docker:

```
root@tantor:~# dmesg | tail -20
[457.152664] vethd2310dc (unregistering): left allmulticast mode
[ 457.152669] vethd2310dc (unregistering): left promiscuous mode
[ 457.152693] br-37b8e28c6b2f: port 1(vethd2310dc) entered disabled state
[ 473.723967] overlayfs: fs on
'/var/lib/docker/overlay2/3b717e6fb2d2e00fd7463b8c3a586f92dfffb26dd6e993b402cbddfe4214f68e/merged' does not
support file handles, falling back to xino=off.
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
[475.027287] br-1fc82bacd93a: port 1(veth63f8123) entered blocking state
[ 475.027293] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
[475.027308] veth63f8123: entered allmulticast mode
[475.027360] veth63f8123: entered promiscuous mode
[475.334287] eth0: renamed from veth68a4798
[475.354710] br-1fc82bacd93a: port 1(veth63f8123) entered blocking state
[475.354726] br-1fc82bacd93a: port 1(veth63f8123) entered forwarding state
[ 475.368511] WARNING: chroot access!
[708.878087] veth68a4798: renamed from eth0
[ 708.902616] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
[ 708.948680] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
[ 708.949310] veth63f8123 (unregistering): left allmulticast mode
[ 708.949315] veth63f8123 (unregistering): left promiscuous mode
[ 708.949340] br-1fc82bacd93a: port 1(veth63f8123) entered disabled state
```

Launch parameters related to overlay can be viewed with the command:

```
root@tantor:~# cat /boot/config -6.6.28-1-generic | grep OVERLAY
CONFIG_EFI_CUSTOM_SSDT_OVERLAYS=y
CONFIG_OVERLAY_FS=m
# CONFIG_OVERLAY_FS_REDIRECT_DIR is not set
CONFIG_OVERLAY_FS_REDIRECT_ALWAYS_FOLLOW=y
# CONFIG_OVERLAY_FS_INDEX is not set
CONFIG_OVERLAY_FS_XINO_AUTO=y
# CONFIG_OVERLAY_FS_METACOPY is not set
# CONFIG_OVERLAY_FS_DEBUG is not set
```

The reason for the warnings may be that the container parameters differ from the parameters of the operating system on which the container is running.

27) This point does not need to be completed, just look at the example.

The following command shows a list of available images:

```
root@tantor:~# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
postgres latest de869b203456 2 days ago 494MB
postgres 17 4 bc 6 cc 20 ca 7 a 2 months ago   435 MB
```

There are two images in the example. The postgres image has two different versions of PostgreSQL, 17.1 and the latest.

You can delete unnecessary images so that they don't take up space . You can specify IMAGE ID or REPOSITORY in the command. Example :

```
root@tantor:~# docker image rm postgres
Untagged: postgres:latest
Untagged: postgres@sha256:6e3358e46e34dae6c184f48fd06fe1b3dbf958ad5b83480031907e52b9ec2a7d
Deleted: sha256:de869b20345625dc5430fbe0f15e490dad3921823915b5524ce1436ec8378793
...
Deleted: sha256:7914c8f600f532b7adbd0b003888e3aa921687d62dbe2f1f829d0ab6234a158a
```

List of images after removal:

```
root @ tantor :~# docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
postgres 17 4 bc 6 cc 20 ca 7 a 2 months ago 435 MB
```

One of the images was not deleted. Let's try use `IMAGE ID` :

```
root@tantor:~# docker image rm 4bc6cc20ca7a
Error response from daemon: conflict: unable to delete 4bc6cc20ca7a ( must be
forced ) - image is being used by stopped container 3116d7318306
```

The image can be removed with the -f (forced) option . The image is used by a stopped container, which is not listed:

```
root@tantor:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Let's delete a non-existent container using its CONTAINER ID :

```
root@tantor:~# docker rm -f 3116d7318306
3116 d 7318306
```

Now the image deletion will be successful:

```
root@tantor:~# docker image rm 4bc6cc20ca7a
Untagged: postgres:17
Untagged:
postgres@sha256:888402a8cd6075c5dc83a31f58287f13306c318eaad016661ed12e076f3e6341
...
Deleted: sha256:8b296f48696071aafb5a6286ca60d441a7e559b192fc7f94bb63ee93dae98f17
```

Freeing up space:

```
root@tantor:~# docker image prune -f
```

List images empty :

```
root@tantor:~# docker images -all
REPOSITORY TAG IMAGE ID CREATED SIZE
```

Part 2. Shared Instance Memory

1) Look at the list of shared memory structures of the instance:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a order by 1;
name | off | size | allocated_size | true_size
------------------------------------------+----------+-----------+----------------+-----------
 <anonymous> | | 4984960 | 4984960 |
Archive Data | 147726208 | 8 | 128 | 128
Async Queue Control | 147729280 | 3984 | 4096 | 4096
AutoVacuum Data | 147713792 | 5328 | 5376 | 5376
Backend Activity Buffer | 146921600 | 131072 | 131072 | 131072
Backend Application Name Buffer | 146905216 | 8192 | 8192 | 8192
Backend Client Host Name Buffer | 146913408 | 8192 | 8192 | 8192
Backend GSS Status Buffer | 147094144 | 8576 | 8576 | 8576
Backend SSL Status Buffer | 147052672 | 41472 | 41472 | 41472
Backend Status Array | 146845824 | 59392 | 59392 | 59392
Background Worker Data | 147102848 | 4496 | 4608 | 4608
BTree Vacuum State | 147726976 | 1476 | 1536 | 1536
 Buffer Blocks | 6894592 | 134221824 | 134221824 | 134221824
 Buffer Descriptors | 5846016 | 1048576 | 1048576 | 1048576
Buffer IO Condition Variables | 141116416 | 262144 | 262144 | 262144
Buffer Strategy Status | 142667648 | 28 | 128 | 128
 Checkpoint BufferIds | 141378560 | 327680 | 327680 | 327680
 Checkpointer Data | 147189376 | 524352 | 524416 | 524416
commit_timestamp | 4907520 | 267424 | 267520 | 267520
CommitTs shared | 5175040 | 32 | 128 | 128
Control File | 4377088 | 328 | 384 | 384
 Fast Path Strong Relation Lock Data | 144058112 | 4100 | 4224 | 4224
FinishedSerializableTransactions | 146353792 | 16 | 128 | 128
KnownAssignedXids | 146774400 | 63440 | 63488 | 63488
KnownAssignedXidsValid | 146837888 | 7930 | 7936 | 7936
 LOCK hash | 142667776 | 2896 | 2944 | 695168
Logical Replication Launcher Data | 147726336 | 496 | 512 | 512
 multixact_member | 5576448 | 267424 | 267520 | 267520
 multixact_offset | 5442688 | 133760 | 133760 | 133760
notify | 147733376 | 133760 | 133760 | 133760
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
OldSnapshotControlData | 147726848 | 88 | 128 | 128
PMSignalState | 147176960 | 1016 | 1024 | 1024
PREDICATELOCK hash | 144507648 | 2896 | 2944 | 1260416
PREDICATELOCKTARGET hash | 144062336 | 2896 | 2944 | 445312
PredXactList | 145768064 | 88 | 128 | 234368
Prepared Transaction Table | 147102720 | 16 | 128 | 128
 Proc Array | 146773760 | 544 | 640 | 640
 Proc Header | 146621568 | 136 | 256 | 152192
 PROCLOCK hash | 143362944 | 2896 | 2944 | 695168
ProcSignal | 147177984 | 11272 | 11392 | 11392
ReplicationOriginState | 147722112 | 568 | 640 | 640
ReplicationSlot Ctl | 147719168 | 2880 | 2944 | 2944
RWConflictPool | 146060800 | 24 | 128 | 292992
SerialControlData | 146621440 | 24 | 128 | 128
serializable | 146353920 | 267424 | 267520 | 267520
SERIALIZABLEXID hash | 146002432 | 2896 | 2944 | 58368
 Shared Buffer Lookup Table | 141706240 | 2896 | 2944 | 961408
Shared Memory Stats | 147867136 | 277880 | 277888 | 316800
Shared MultiXact State | 5843968 | 2008 | 2048 | 2048
shmInvalBuffer | 147107456 | 69464 | 69504 | 69504
subtransaction | 5175168 | 267424 | 267520 | 267520
Sync Scan Locations List | 147728512 | 656 | 768 | 768
Tantor Control File | 4377472 | 40 | 128 | 128
transaction | 4377856 | 529568 | 529664 | 529664
Wal Receiver Ctl | 147723904 | 2264 | 2304 | 2304
Wal Sender Ctl | 147722752 | 1144 | 1152 | 1152
 XLOG Ctl | 168832 | 4208232 | 4208256 | 4208256
XLogPrefetchStats | 4377600 | 72 | 128 | 128
XLOG Recovery Ctl | 4377728 | 104 | 128 | 128
     | 148183936 | 1885312 | 1885312 |
(60 rows)
```

Values in bytes.

names of the structures mean nothing and are not included in the documentation. In books and textbooks, shared memory is discussed using several structures as an example: buffer cache, log buffer, transaction status buffer CLOG (**Xact**) . In the `name column` **of the pg_shmem_allocations** view These structures are not easy to find.

Buffer Blocks - buffer cache, the size is specified by the shared_buffers parameter

Buffer Descriptors - headers of buffers that store pinning attributes count , number of buffer accesses usage count and other bits. Size depends on number of buffers.

Shared Buffer Lookup Table - (Buffer Mapping Table) is a hash table for searching blocks in the buffer cache.

`Buffer Strategy Status` - (BufferStrategyControl structure) data on free blocks in the buffer cache (first free buffer firstFreeBuffer , last free buffer lastFreeBuffer , pointer to the block for the clock algorithm sweep and other meanings)

XLOG Ctl - WAL buffer, log buffer, size is specified by wal_buffers parameter .

`transaction` buffer, the size of which is specified by the transaction_buffers parameter.

`subtransaction` buffer, the size of which is specified by the subtransaction_buffers parameter .

`serializable` buffer, the size of which is specified by the serializable_buffers parameter .

`notify` buffer whose size is specified by the notify_buffers parameter .

multixact_member buffer, the size of which is specified by the multixact_member_buffers **parameter .**

multixact_offset buffer, the size of which is specified by the multixact_offset_buffers **parameter .**

`commit_timestamp` a buffer whose size is specified by the commit_timestamp_buffers parameter.

`shmInvalBuffer` - a ring buffer for 4096 messages about changes in the system catalog tables.

The size of `KnownAssignedXids` and `KnownAssignedXidsValid` (8 times smaller than `KnownAssignedXids`) is proportional to the sum of the values of the parameters max _ connections + maximum number of background processes of the instance + max _ prepared _ transactions . The structures are allocated only if the parameter `hot_standby=on` , including on the master. They are used by the replica in recovery mode.

2) C list of shared memory structures for which the allocated memory data differs from the size and allocated_size data:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a where a.true_size<>a.allocated_size
order by 1;
 name | off | size | allocated_size | true_size
-----------------------------+----------+--------+----------------+----------
 LOCK hash | 142667776 | 2896 | 2944 | 695168
 PREDICATELOCK hash | 144507648 | 2896 | 2944 | 1260416
 PREDICATELOCKTARGET hash | 144062336 | 2896 | 2944 | 445312
 PredXactList | 145768064 | 88 | 128 | 234368
 Proc Header | 146621568 | 136 | 256 | 152192
 PROCLOCK hash | 143362944 | 2896 | 2944 | 695168
 RWConflictPool | 146060800 | 24 | 128 | 292992
 SERIALIZABLEXID hash | 146002432 | 2896 | 2944 | 58368
 Shared Buffer Lookup Table | 141706240 | 2896 | 2944 | 961408
 Shared Memory Stats | 147867136 | 277880 | 277888 | 316800
 (10 rows)
```

3) Statistics using SLRU caches :

```
postgres=# select name , blks_hit, blks_read, blks_written, blks_exists,
flushes, truncates from pg_stat_slru;
       name | blks_hit | blks_read | blks_written | blks_exists | flushes | truncates
------------------+----------+-----------+--------------+-------------+---------+----------
 commit_timestamp | 0 | 0 | 0 | 0 | 253 | 0
 multixact_member | 0 | 0 | 0 | 0 | 253 | 0
 multixact_offset | 0 | 6 | 5 | 0 | 253 | 0
 notify | 0 | 0 | 0 | 0 | 0 | 0
 serializable | 0 | 0 | 0 | 0 | 0 | 0
 subtransaction | 0 | 0 | 210 | 0 | 253 | 252
 transaction | 409116 | 7 | 221 | 0 | 253 | 0
 other | 0 | 0 | 0 | 0 | 0 | 0
 (8 rows)
```

Statistics are saved. The view has a column `stats_reset` since the last time statistics were reset. Resetting statistics is performed by the function: select pg _ stat _ reset _ slru (null); cache name as an argument , or NULL if you want to reset statistics for all caches.
The statistics from the view can be used to set configuration parameters that control the sizes of SLRU caches:

```
postgres=# \dconfig *_buffers
 Parameter | Value
--------------------------+-------
 commit_timestamp_buffers | 256kB
 multixact_member_buffers | 256kB
 multixact_offset_buffers | 128kB
 notify_buffers | 128kB
 serializable_buffers | 256kB
 shared_buffers | 128MB
 subtransaction_buffers | 256kB
 temp_buffers | 8MB
 transaction_buffers | 256kB
 wal_buffers | 4MB
```

Next, we will look at how to determine which shared memory structures of an instance are affected by changes in configuration parameters. Some configuration parameters affect a large number of memory structures. The size of the change is also important, since some structures are accessed very frequently by processes, and such structures (or their parts that are accessed frequently) will be located (or evicted and loaded each time) in processor caches, and references to their pages in the TLB. The sizes of processor caches are not very large: from kilobytes to megabytes, and even a small increase in some structures (for example, `Proc Header` and `Proc Array` from 100 KB to 200 KB) can lead to a sharp increase in the number of evictions and loading into caches.
The second point: to access (read, change the contents) to parts of structures in shared memory (or the entire structure if it is small), the process always gets a lock: either a SpinLock or an LWLock. If the

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

number of LWLocks that protect the structure does not change, then when the size of the structure (or its part protected by one lock) increases, the process will read/change the contents of the structure longer, which will lead to an increase in the duration of holding the LWLock. This can lead to the appearance or increase in the number of process collisions when receiving an LWLock . LWLocks are not designed for a long time of holding and for a long wait for receiving a lock.

By increasing the number of LWLocks, you can determine which memory structure this type of lock protects access to and check whether this structure has been increased by some configuration parameter. Perhaps reducing the memory structure will eliminate the contention for LWLocks.

In some forks, the parameters may increase the number of LWLocks that protect parts of the memory structure (the number of "trenches"), but the LWLock itself is also in the memory structure (usually protected by a SpinLock) and the size of such a structure also increases. Access to such a structure may become a bottleneck.

5) Create views for ease of writing queries and a table with the original (with which they will be compared) sizes of shared memory structures:

```
postgres=# drop table if exists shmem_reference cascade;
create table shmem_reference as select coalesce(name, name, '<NULL>') name,
size, allocated_size, coalesce(true_size, true_size, allocated_size) true_size
from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a order by 1;
create or replace view shmem_v as select coalesce(name, name, '<NULL>') name,
size, allocated_size, coalesce(true_size, true_size, allocated_size) true_size
from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a order by 1;
create or replace view shmem_compare as select v.name, v.allocated_size
allocated, r.allocated_size ref_alloc, v.true_size true_size, r.true_size
ref_true, v.size, r.size ref_size from shmem_reference r full outer join shmem_v
v on (r.name=v.name) where r.true_size<>v.true_size order by v.name;
\q
NOTICE: table "shmem_reference" does not exist, skipping
DROP TABLE
SELECT 60
CREATE VIEW
CREATE VIEW
postgres@tantor:~$
```

6) Increase the buffer cache size to 130 MB, restart the instance and see which memory structures have changed in size:

```
postgres@tantor:~$ psql -c "alter system set shared_buffers='130MB'"
sudo restart
psql -c "select * from shmem_compare"
ALTER SYSTEM
name | allocated | ref_alloc | true_size | ref_true | size | ref_size
------------------------------+-----------+-----------+-----------+-----------+-----------+----------
<anonymous> | 4958336 | 4946048 | 4958336 | 4946048 | 4958336 | 4946048
Buffer Blocks | 136318976 | 134221824 | 136318976 | 134221824 | 136318976 | 134221824
Buffer Descriptors | 1064960 | 1048576 | 1064960 | 1048576 | 1064960 | 1048576
Buffer IO Condition Variables | 266240 | 262144 | 266240 | 262144 | 266240 | 262144
Checkpoint BufferIds | 332800 | 327680 | 332800 | 327680 | 332800 | 327680
Checkpointer Data | 532608 | 524416 | 532608 | 524416 | 532544 | 524352
<NULL> | 1918976 | 1924224 | 1918976 | 1924224 | 1918976 | 1924224
Shared Buffer Lookup Table | 2944 | 2944 | 973696 | 961408 | 2896 | 2896
 XLOG Ctl | 4273920 | 4208256 | 4273920 | 4208256 | 4273832 | 4208232
(9 rows)
```

The change affected 9 shared memory structures. Including structures whose memory allocation method does not allow them to be named and they are displayed as < anonymous > and < NULL > (empty). These names account for not one, but several structures. < NULL > - allocated but unused < anonymous > memory.

the XLOG Ctl log buffer has increased to 1/32 **shared _ buffers** : **136318976** / **4273920** = 31.8955. Why is the number not round 32? Because in the Buffer structure Blocks stores not only buffers, the number of which is set by **the shared_buffers parameter** .

7) Increase the buffer cache size to 1024 MB, restart the instance and see which memory structures have changed in size:

```
postgres@tantor:~$ psql -c "alter system set shared_buffers='1024MB'"
sudo restart
psql -c "select * from shmem_compare"
psql -c "alter system reset shared_buffers"
ALTER SYSTEM
name | allocated | ref_alloc | true_size | ref_true | size | ref_size
------------------------------+-----------+-----------+------------+-----------+-----------+----------
<anonymous> | 11368576 | 4946048 | 11368576 | 4946048 | 11368576 | 4946048
Buffer Blocks | 1073745920 | 134221824 | 1073745920 | 134221824 | 1073745920 | 134221824
Buffer Descriptors | 8388608 | 1048576 | 8388608 | 1048576 | 8388608 | 1048576
Buffer IO Condition Variables | 2097152 | 262144 | 2097152 | 262144 | 2097152 | 262144
Checkpoint BufferIds | 2621440 | 327680 | 2621440 | 327680 | 2621440 | 327680
Checkpointer Data | 4194432 | 524416 | 4194432 | 524416 | 4194368 | 524352
 commit_timestamp | 2138624 | 267520 | 2138624 | 267520 | 2138560 | 267424
<NULL> | 1920384 | 1924224 | 1920384 | 1924224 | 1920384 | 1924224
Shared Buffer Lookup Table | 9088 | 2944 | 7390080 | 961408 | 9040 | 2896
 subtransaction | 2138624 | 267520 | 2138624 | 267520 | 2138560 | 267424
 transaction | 4235776 | 529664 | 4235776 | 529664 | 4235712 | 529568
XLOG Ctl | 16803456 | 4208256 | 16803456 | 4208256 | 16803432 | 4208232
(12 rows)
ALTER SYSTEM
```

Three more structures were affected by a larger change . This could be due to alignment if the structure is small or the structure size is calculated using a formula that includes the number of buffers (NBuffers).

The XLOG Ctl log buffer size has increased to 16MB, as by default it is set to either the **16MB** WAL segment size or 1/32 **shared_buffer** . Now XLOG Ctl was set to the size of the WAL segment .

Under what name is the CLOG buffer "hidden" , which is often given as an example of a shared memory structure? The size of this SLRU buffer is calculated by the formula: Min (128, Max (4, **NBuffers/512**)) . For 128 MB it is 262144 bytes, 1 GB - 1028576 bytes. CLOG can be hidden under any of the lines.

Let's try to detect CLOG in another way. Under CLOG, memory is allocated by the CLOGShmemInit () function , which is called in the ipci.c code . The allocation of shared memory occurs in the following sequence:

```
/*
* Set up xlog, clog, and buffers
*/
XLOGShmemInit ();
XLogPrefetchShmem Init();
XLogRecoveryShmem Init();
CLOGShmemInit();
CommitTsShmem Init();
SUBTRANSShmem Init();
MultiXactShmem Init();
InitBufferPool ();
/*
* Set up lock manager
 */
InitLocks ();
```

8) Print the structures in the order in which the operating system allocated memory:

```
postgres=# select * from (select *, lead(off) over(order by off) - off as
true_size from pg_shmem_allocations) as a order by off limit 20;
name | off | size | allocated_size | true_size
------------------------------+-----------+-----------+----------------+------------
 XLOG Ctl | 168832 | 16803432 | 16803456 | 16803456
Control File | 16972288 | 328 | 384 | 384
Tantor Control File | 16972672 | 40 | 128 | 128
 XLogPrefetchStats | 16972800 | 72 | 128 | 128
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
 XLOG Recovery Ctl | 16972928 | 104 | 128 | 128
transaction | 16973056 | 4235712 | 4235776 | 4235776
 commit_timestamp | 21208832 | 2138560 | 2138624 | 2138624
 CommitTs shared | 23347456 | 32 | 128 | 128
 subtransaction | 23347584 | 2138560 | 2138624 | 2138624
 multixact_offset | 25486208 | 133760 | 133760 | 133760
 multixact_member | 25619968 | 267424 | 267520 | 267520
Shared MultiXact State | 25887488 | 2008 | 2048 | 2048
 Buffer Descriptors | 25889536 | 8388608 | 8388608 | 8388608
 Buffer Blocks | 34278144 | 1073745920 | 1073745920 | 1073745920
 Buffer IO Condition Variables | 1108024064 | 2097152 | 2097152 | 2097152
 Checkpoint BufferIds | 1110121216 | 2621440 | 2621440 | 2621440
 Shared Buffer Lookup Table | 1112742656 | 9040 | 9088 | 7390080
 Buffer Strategy Status | 1120132736 | 28 | 128 | 128
LOCK hash | 1120132864 | 2896 | 2944 | 695168
PROCLOCK hash | 1120828032 | 2896 | 2944 | 695168
(20 rows)
```

CLOG buffer V presentation has Name **transaction** .
Size buffers **transaction** is set parameter configurations `transaction_buffers` .
Description of the parameter in the documentation https://docs.tantorlabs.ru/tdb/ru/16_6/se/runtime-config-resource.html :
amount of shared memory used to cache the contents of the `PGDATA/` **`pg_xact directory`** . If this value is specified without units, it is taken in blocks (BLCKSZ=8kB). The default value is 0 and the size will be set from **`shared_buffers/512`** to 1024 blocks, but not less than 16 blocks.

have seen that the **off column of the pg_shmem_allocations** view can **be** used not only to identify "holes " in the virtual address space, but also **to identify memory structures** .

changing the max_connections parameter affects memory structures :

```
postgres@tantor:~$ psql -c "alter system reset shared_buffers"
psql -c "alter system set max_connections =200"
sudo restart
psql -c "select * from shmem_compare"
psql -c "select sum(true_size) - sum(ref_size) delta_true from shmem_compare"
psql -c "alter system reset max_connections"
ALTER SYSTEM
ALTER SYSTEM
name | allocated | ref_alloc | true_size | ref_true | size | ref_size
-------------------------------+-----------+-----------+----------+---------+----------
<anonymous> | 8131200 | 4946048 | 8131200 | 4946048 | 8131200 | 4946048
Async Queue Control | 7296 | 4096 | 7296 | 4096 | 7184 | 3984
Backend Activity Buffer | 233472 | 131072 | 233472 | 131072 | 233472 | 131072
Backend Application Name Buffer | 14592 | 8192 | 14592 | 8192 | 14592 | 8192
Backend Client Host Name Buffer | 14592 | 8192 | 14592 | 8192 | 14592 | 8192
Backend GSS Status Buffer | 15360 | 8576 | 15360 | 8576 | 15276 | 8576
Backend SSL Status Buffer | 73984 | 41472 | 73984 | 41472 | 73872 | 41472
Backend Status Array | 105856 | 59392 | 105856 | 59392 | 105792 | 59392
BTree Vacuum State | 2688 | 1536 | 2688 | 1536 | 2676 | 1476
KnownAssignedXids | 115456 | 63488 | 115456 | 63488 | 115440 | 63440
KnownAssignedXidsValid | 14464 | 7936 | 14464 | 7936 | 14430 | 7930
LOCK hash | 2944 | 2944 | 1264512 | 695168 | 2896 | 2896
<NULL> | 3441152 | 1924224 | 3441152 | 1924224 | 3441152 | 1924224
PMSignalState | 1920 | 1024 | 1920 | 1024 | 1816 | 1016
PREDICATELOCK hash | 2944 | 2944 | 2313088 | 1260416 | 2896 | 2896
PREDICATELOCKTARGET hash | 2944 | 2944 | 818048 | 445312 | 2896 | 2896
PredXactList | 128 | 128 | 426368 | 234368 | 88 | 88
Proc Array | 1024 | 640 | 1024 | 640 | 944 | 544
Proc Header | 256 | 256 | 271872 | 152192 | 136 | 136
PROCLOCK hash | 2944 | 2944 | 1272704 | 695168 | 2896 | 2896
ProcSignal | 20096 | 11392 | 20096 | 11392 | 20072 | 11272
RWConflictPool | 128 | 128 | 532992 | 292992 | 24 | 24
SERIALIZABLEXID hash | 2944 | 2944 | 106752 | 58368 | 2896 | 2896
Shared MultiXact State | 3712 | 2048 | 3712 | 2048 | 3608 | 2008
shmInvalBuffer | 72704 | 69504 | 72704 | 69504 | 72664 | 69464
(25 rows)
delta_true
------------
  11972874
(1 row )
ALTER SYSTEM
```

max_connections parameter from 100 to 200 affected a large number of memory structures. An additional 11972874 bytes were allocated.

10) Look at how changing the max_connections parameter affects memory structures , which , along with max_locks_per_transaction , limits the total number of object locks on the entire instance :

```
postgres@tantor:~$ psql -c "alter system set max_locks_per_transaction =128"
sudo restart
psql -c "select * from shmem_compare"
psql -c "select sum(true_size) - sum(ref_size) delta_true from shmem_compare"
psql -c "alter system reset max_locks_per_transaction"
ALTER SYSTEM
name | allocated | ref_alloc | true_size | ref_true | size | ref_size
---------------+-----------+-----------+-----------+----------+---------+----------
<anonymous> | 6322304 | 4946048 | 6322304 | 4946048 | 6322304 | 4946048
LOCK hash | 2944 | 2944 | 1383296 | 695168 | 2896 | 2896
<NULL> | 3579008 | 1924224 | 3579008 | 1924224 | 3579008 | 1924224
PROCLOCK hash | 2944 | 2944 | 1383296 | 695168 | 2896 | 2896
(4 rows)
delta_true
------------
    5791840
(1 row)
ALTER SYSTEM
```

Number blocking objects And recommendations advisory locks are limited on copy work max_locks_per_transaction * (max_connections + max_prepared_transactions) . If you need to increase the number of locks on an instance, then increase one of these parameters or both parameters at once.

Changing the max_locks_per_transaction parameter value from 64 to 128 affected 3 memory structures and unused (< NULL >) memory area. An additional 5791840 bytes were allocated , which is 2 times less than when changing max_connections .

However, it is impossible to say which parameter will decrease performance more by increasing it, since it is not known which structure's size affects performance more and by how much. If the memory structure is accessed much more often, then increasing the size of this structure will decrease performance more. The heavy lock structure is located in < anonymous > . To access the heavy lock structure, use LOCK hash . Usually, the PROC LOCK hash structure is accessed more often (by ~"number of processes" times) than the heavy lock structure.

The queries given in this part of the practice can be used to obtain how a change in a configuration parameter will affect the size of memory structures and to estimate which locks associated with their memory structures will appear more often.

Part 3. Local memory of the server process

1) Run queries in a new psql session:

```
postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes)
used, sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
with recursive dep as
(select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
union all
select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' || c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select name, total, parent, level l, path from dep where parent<>'CacheMemoryContext';
select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;

total | blocks | used | free | f_chunks
---------+--------+--------+--------+---------
1321128 | 196 | 965664 | 355464 | 130
(1 row)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
name | total | parent | l | path
-------------------------+--------+-----------------+---+------------------------------------------
Type information cache | 24368 | TopMemoryContext | 2 | TopMemoryContext->Type information cache
Operator lookup cache | 24576 | TopMemoryContext | 2 | TopMemoryContext->Operator lookup cache
TopTransactionContext | 8192 | TopMemoryContext | 2 | TopMemoryContext->TopTransactionContext
Record information cache | 8192 | TopMemoryContext | 2 | TopMemoryContext->Record information cache
RowDescriptionContext | 8192 | TopMemoryContext | 2 | TopMemoryContext->RowDescriptionContext
MessageContext | 262144 | TopMemoryContext | 2 | TopMemoryContext->MessageContext
Operator class cache | 8192 | TopMemoryContext | 2 | TopMemoryContext->Operator class cache
smgr relation table | 32768 | TopMemoryContext | 2 | TopMemoryContext->smgr relation table
PgStat Shared Ref Hash | 7216 | TopMemoryContext | 2 | TopMemoryContext->PgStat Shared Ref Hash
PgStat Shared Ref | 4096 | TopMemoryContext | 2 | TopMemoryContext->PgStat Shared Ref
PgStat Pending | 8192 | TopMemoryContext | 2 | TopMemoryContext->PgStat Pending
TransactionAbortContext | 32768 | TopMemoryContext | 2 | TopMemoryContext->TransactionAbortContext
Portal hash | 8192 | TopMemoryContext | 2 | TopMemoryContext->Portal hash
TopPortalContext | 8192 | TopMemoryContext | 2 | TopMemoryContext->TopPortalContext
Relcache by OID | 16384 | TopMemoryContext | 2 | TopMemoryContext->Relcache by OID
CacheMemoryContext | 524288 | TopMemoryContext | 2 | TopMemoryContext->CacheMemoryContext
WAL record construction | 50200 | TopMemoryContext | 2 | TopMemoryContext->WAL record construction
PrivateRefCount | 8192 | TopMemoryContext | 2 | TopMemoryContext->PrivateRefCount
MdSmgr | 8192 | TopMemoryContext | 2 | TopMemoryContext->MdSmgr
LOCALOCK hash | 8192 | TopMemoryContext | 2 | TopMemoryContext->LOCALLOCK hash
GUCMemoryContext | 24576 | TopMemoryContext | 2 | TopMemoryContext->GUCMemoryContext
Timezones | 104112 | TopMemoryContext | 2 | TopMemoryContext->Timezones
ErrorContext | 8192 | TopMemoryContext | 2 | TopMemoryContext->ErrorContext
PortalContext | 1024 | TopPortalContext | 3 | TopMemoryContext->TopPortalContext->PortalContext
GUC hash table | 32768 | GUCMemoryContext | 3 | TopMemoryContext->GUCMemoryContext->GUC hash table
ExecutorState | 147696 | PortalContext | 4 | TopMemoryContext->TopPortalContext->PortalContext-
                                                              >ExecutorState
printup | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->printtup
ExprContext | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->ExprContext
ExprContext | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->ExprContext
ExprContext | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->ExprContext
Table function arguments | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->Table function arguments
ExprContext | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->ExprContext
ExprContext | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->ExprContext
Table function arguments | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->Table function arguments
ExprContext | 8192 | ExecutorState | 5 | TopMemoryContext->TopPortalContext->PortalContext-
>ExecutorState->ExprContext
(35 rows)

total | blocks | used | free | f_chunks
---------+--------+--------+--------+----------
1387480 | 202 | 1033400 | 354080 | 134
(1 row )
```

The first and third queries show the total memory of the server process (total), used (used), allocated but not used (free). The number of blocks and chunks is not informative, since the sizes of blocks and chunks are different and unknown.

To get the memory size of a context, you need to add the memory size of its child contexts to its size.

The second query gives the hierarchy of memory contexts. For compactness, parts of the system cache (`CacheMemoryContext`) in which the ident column (removed from the output for compactness) contains the names of system catalog objects are removed from the output.

The result of the third query differs from the first: total and used increased slightly. This happened because memory was allocated to execute queries. After executing queries, contexts were released, but the memory already allocated to the process is not returned to the operating system.

2) Run the commands:

```
postgres=# set temp_buffers = ' 100MB ';
create temp table temp1 (id integer);
insert into temp1 select * from generate_series(1, 1000000);
explain (analyze, buffers) select * from temp1;
drop table temp1;
select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
```

```
with recursive dep as
(select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
union all
select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' ||
c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select name, total, parent, level l, path from dep where name like '%Local%';
SET
CREATE TABLE
INSERT 0 1000000
QUERY PLAN
--------------------------------------------------------------------------------
Seq Scan on temp1 (cost=0.00..15708.75 rows=1128375 width=4) (actual time=0.026..4071.777 rows=1000000 loops=1)
Buffers: local hit=4425
Planning:
Buffers: shared hit=20
Planning Time: 0.125 ms
Execution Time: 8043.230 ms
(6 rows)

DROP TABLE
total | blocks | used | free | f_chunks
----------+--------+---------+--------+----------
68948568 | 239 | 68399976 | 548592 | 167
(1 row)

name | total | parent | l | path
-------------------------+----------+-----------------+---+------------------------------
LocalBufferContext | 67023280 | TopMemoryContext | 2 | TopMemoryContext->LocalBufferContext
Local Buffer Lookup Table | 524288 | TopMemoryContext | 2 | TopMemoryContext->Local Buffer Lookup Table
(2 rows)
```

The teams set the buffer size for temporary table blocks to 10. They created a temporary table, filled it with local buffer blocks, and deleted the temporary table. The local buffer did not free up memory because temporary tables may still be created in this session, and allocating memory in the operating system and returning it to the operating system is not optimal.

3) It is not possible to free the local buffer memory during a session, only to close the session :

```
postgres=# set temp_buffers = '8MB';
ERROR: invalid value for parameter "temp_buffers": 1024
DETAIL: "temp_buffers" cannot be changed after any temporary tables have been
accessed in the session .
```

The RESET command does not generate an error, but the buffer is not freed :

```
postgres=# reset temp_buffers;
show temp_buffers;
RESET
temp_buffers
--------------
8MB
(1 row)

postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks,
sum(used_bytes) used, sum(free_bytes) free, sum(free_chunks) f_chunks from
pg_backend_memory_contexts;
total | blocks | used | free | f_chunks
----------+--------+---------+--------+----------
68998536 | 244 | 68455416 | 543120 | 207
(1 row)

postgres=# with recursive dep as
(select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
```

tantor

```
 union all
 select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' ||
c.name
 from dep p, pg_backend_memory_contexts c
 where c.parent = p.name)
 select name, total, parent, level l, path from dep where name like '%Local%';
 name | total | parent | l | path
--------------------------+----------+------------------+---+----------------------------------------
 LocalBufferContext | 67023280 | TopMemoryContext | 2 | TopMemoryContext->LocalBufferContext
 Local Buffer Lookup Table | 524288 | TopMemoryContext | 2 | TopMemoryContext->Local Buffer Lookup Table
 (2 rows )
```

4) Moreover, if you repeat the commands for creating and deleting a temporary table, the local buffer will grow in size up to the **set temp_buffer = ' 100MB '** :

```
 postgres=# create temp table temp1 (id integer);
 insert into temp1 select * from generate_series(1, 1000000);
 explain (analyze, buffers) select * from temp1;
 drop table temp1;
 select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
 with recursive dep as
 (select name, total_bytes as total, ident, parent, 1 as level, name as path from
pg_backend_memory_contexts where parent is null
 union all
 select c.name, c.total_bytes, c.ident, c.parent, p.level + 1, p.path || '->' || c.name
 from dep p, pg_backend_memory_contexts c
 where c.parent = p.name)
 select name, total, parent, level l, path from dep where name like '%Local%';
 CREATE TABLE
 INSERT 0 1000000
 QUERY PLAN
 -----------------------------------------------------------------------------------------
 Seq Scan on temp1 (cost=0.00..15708.75 rows=1128375 width=4) (actual time=0.028..4069.873 rows=1000000 loops=1)
 Buffers: local hit=4425
 Planning:
 Buffers: shared hit=4
 Planning Time: 0.050 ms
 Execution Time: 8078.346 ms
 (6 rows)

 DROP TABLE
 total | blocks | used | free | f_chunks
-----------+--------+-----------+---------+----------
 107406776 | 246 | 106343504 | 1063272 | 211
 (1 row)

 name | total | parent | l | path
--------------------------+-----------+------------------+---+----------------------------------------
 LocalBufferContext | 104907232 | TopMemoryContext | 2 | TopMemoryContext->LocalBufferContext
 Local Buffer Lookup Table | 524288 | TopMemoryContext | 2 | TopMemoryContext->Local Buffer Lookup Table
 (2 rows)
```

5) Not only does the local buffer not release memory, but the system directory cache also does not return memory to the operating system. new psql sessions execute commands :

```
 postgres=# drop table if exists parttab;
create table parttab(n numeric,k numeric, v varchar(100)) partition by range (n);
do $$ declare cnt integer; v varchar(200); begin for i in 0..2000 loop v:=
concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')'); execute v; end loop; end; $$ ;

 select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;

with recursive dep as
 (select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
```

```
union all select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident,
c.parent, p.level + 1, p.path || '->' || c.name from dep p,
pg_backend_memory_contexts c where c.parent = p.name) select name, total, used,
free, parent, level l, path from dep where name like '%CacheMemory%';

 drop table if exists parttab;

select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;

with recursive dep as
 (select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
union all select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident,
c.parent, p.level + 1, p.path || '->' || c.name from dep p,
pg_backend_memory_contexts c where c.parent = p.name) select name, total, used,
free, parent, level l, path from dep where name like '%CacheMemory%';
```

```
 NOTICE: table "parttab" does not exist, skipping
DROP TABLE
CREATE TABLE
DO
 total | blocks | used | free | f_chunks
----------+--------+---------+----------+----------
 21854616 | 223 | 2016048 | 19838568 | 52954
 (1 row)

 name | total | used | free | parent | l | path
--------------------+----------+--------+----------+------------------+---+-------------------------
 CacheMemoryContext | 16924912 | 485512 | 16439400 | TopMemoryContext | 2 | TopMemoryContext->CacheMemoryContext
 (1 row)

 DROP TABLE
 NOTICE: table "parttab" does not exist, skipping
 DROP TABLE
 total | blocks | used | free | f_chunks
----------+--------+---------+----------+----------
 24485160 | 403 | 3309016 | 21176144 | 69436
 (1 row)

 name | total | used | free | parent | l | path
--------------------+----------+--------+----------+------------------+---+-------------------------
 CacheMemoryContext | 17023312 | 585528 | 16437784 | TopMemoryContext | 2 | TopMemoryContext->CacheMemoryContext
 (1 row)
```

Unlike the local buffer, the system catalog cache marks the memory as free.
Data for 2000 table sections was loaded into the system catalog cache memory when the execute command **was executed v ;** . The sections were used as a simple example. In real work, the system catalog cache will be filled with data on a large number of objects if the command at the planning stage had requests for information on a large number of database objects. At the execution stage of the command, the number of objects may decrease or remain the same if all objects remained in the created plan.
The result shows the size of only the root `CacheMemoryContext` . The size of its descendants is not shown.

6) Does the system catalog cache hold data until the end of the request or transaction?
It can be assumed that on the one hand, changes in the system catalog should become visible after the transaction ends, on the other hand, the transaction itself should see its changes. To check, in a new psql session, run the commands:

```
postgres=# \q
postgres@tantor:~$ psql

postgres=# drop table if exists parttab;
create table parttab(n numeric,k numeric, v varchar(100)) partition by range
(n);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
begin transaction;
do
$$
declare
cnt integer;
vvarchar(200);
begin
for i in 0..3000 loop
v:= concat('create table parttab_',i,' partition of parttab for values from
(',i*10,') to (',(i+1)*10,')');
execute v;
end loop;
end;
$$
;
select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used,
sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;

with recursive dep as
(select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
union all
select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident, c.parent,
p.level + 1, p.path || '->' || c.name
from dep p, pg_backend_memory_contexts c
where c.parent = p.name)
select name, total, used, free, parent, level l, path from dep where name like
'%CacheMemory%';

commit;
```

```
NOTICE: table "parttab" does not exist, skipping
DROP TABLE
CREATE TABLE
BEGIN
DO
DO
DO
total | blocks | used | free | f_chunks
----------+--------+---------+---------+----------
43441704 | 6396 | 37203376 | 6238328 | 3553
(1 row)

name | total | used | free | parent | l | path
------------------+----------+----------+---------+----------------+---+------------------------------
CacheMemoryContext| 25460976 | 22374816 | 3086160 |TopMemoryContext| 2 |TopMemoryContext->CacheMemoryContext
(1 row)

COMMIT
```

Three commands were executed, adding 1000 sections to the table each. The size of the `CacheMemoryContext context root` became `25460976` bytes, and in the previous example, when adding 2000 sections, it was `16924912` .

7) Check that after the transaction is committed, the contents of the `CacheMemoryContext context` freed , but the memory allocated `by CacheMemoryContext` will not be returned to the operating system:

```
postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks,
sum(used_bytes) used, sum(free_bytes) free, sum(free_chunks) f_chunks from
pg_backend_memory_contexts;

with recursive dep as
```

```
(select name, total_bytes as total, used_bytes as used, free_bytes as free,
ident, parent, 1 as level, name as path from pg_backend_memory_contexts where
parent is null
 union all
 select c.name, c.total_bytes, c.used_bytes, c.free_bytes, c.ident, c.parent,
p.level + 1, p.path || '->' || c.name
 from dep p, pg_backend_memory_contexts c
 where c.parent = p.name)
 select name, total, used, free, parent, level l, path from dep where name like
'%CacheMemory%';
  total | blocks | used | free | f_chunks
----------+--------+---------+----------+----------
31021256 | 269 | 2671688 | 28349568 | 79149
(1 row)

name | total | used | free | parent | l | path
-------------------+----------+--------+---------+---------------+---+-----------------------------------
CacheMemoryContext | 25460976 | 633008 | 24827968 |TopMemoryContext| 2 |TopMemoryContext->CacheMemoryContext
(1 row)
```

At the same time, part of the server process memory was freed: it became 31021256 , and was 43441704 . Most of the freed memory (5818512) belonged to the TopTransactionContext , which is completely freed after the transaction is completed. The other part belonged to the contexts " index info " (3000 pieces of 2048 bytes), heirs of CacheMemoryContext .

8) Delete table parttab :

```
postgres=# drop table parttab;
DROP TABLE
```

Part 4. Logging process memory to the diagnostic log

1) To view the diagnostic log in a convenient place - PGDATA / log directory, enable logging _ collector :

```
postgres=# \q
postgres@tantor:~$ psql -c "alter system set logging_collector=on"
ALTER SYSTEM
postgres@tantor:~$ sudo restart
postgres@tantor:~$ cat $PGDATA/current_logfiles
stderr log/postgresql-YYYY-MM-DD_HHmiss.log
postgres@tantor:~$ ls $PGDATA/log
postgresql-YYYY-MM-DD_HHmiss.log
postgres@tantor:~$ psql
```

The name of the current diagnostic log file is specified in the $PGDATA/current_logfiles file . The file name includes the date and time it was created.

2) Log the memory data of your server process:

```
postgres=# select pg_log_backend_memory_contexts(pg_backend_pid());
pg_log_backend_memory_contexts
--------------------------------
 t
(1 row)
```

3) Look at what appeared at the end of the log file:

```
postgres=# \! tail -112 $PGDATA/log/postgresql-20*
LOG: level: 0; TopMemoryContext: 97664 total in 5 blocks; 13920 free (8 chunks); 83744 used
LOG: level: 1; TopTransactionContext: 8192 total in 1 blocks; 7752 free (1 chunks); 440 used
LOG: level: 1; RowDescriptionContext: 8192 total in 1 blocks; 6896 free (0 chunks); 1296 used
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
LOG: level: 1; MessageContext: 16384 total in 2 blocks; 5960 free (0 chunks); 10424 used
LOG: level: 1; Operator class cache: 8192 total in 1 blocks; 592 free (0 chunks); 7600 used
LOG: level: 1; smgr relation table: 16384 total in 2 blocks; 4640 free (2 chunks); 11744 used
LOG: level: 1; PgStat Shared Ref Hash: 7216 total in 2 blocks; 688 free (0 chunks); 6528 used
LOG: level: 1; PgStat Shared Ref: 4096 total in 3 blocks; 1808 free (2 chunks); 2288 used
LOG: level: 1; PgStat Pending: 8192 total in 4 blocks; 5960 free (21 chunks); 2232 used
LOG: level: 1; TransactionAbortContext: 32768 total in 1 blocks; 32504 free (0 chunks); 264 used
LOG: level: 1; Portal hash: 8192 total in 1 blocks; 592 free (0 chunks); 7600 used
LOG: level: 1; TopPortalContext: 8192 total in 1 blocks; 7664 free (0 chunks); 528 used
LOG: level: 2; PortalContext: 1024 total in 1 blocks; 608 free (0 chunks); 416 used: <unnamed>
LOG: level: 3; ExecutorState: 8192 total in 1 blocks; 3920 free (0 chunks); 4272 used
LOG: level: 4; printtup: 8192 total in 1 blocks; 7928 free (0 chunks); 264 used
LOG: level: 4; ExprContext: 8192 total in 1 blocks; 7928 free (0 chunks); 264 used
LOG: level: 1; Relcache by OID: 16384 total in 2 blocks; 3584 free (2 chunks); 12800 used
LOG: level: 1; CacheMemoryContext: 524288 total in 7 blocks; 117128 free (0 chunks); 407160 used
...
LOG: level: 2; index info: 2048 total in 2 blocks; 912 free (0 chunks); 1136 used: pg_authid_rolname_index
LOG: level: 1; WAL record construction: 50200 total in 2 blocks; 6376 free (0 chunks); 43824 used
LOG: level: 1; PrivateRefCount: 8192 total in 1 blocks; 2648 free (0 chunks); 5544 used
LOG: level: 1; MdSmgr: 8192 total in 1 blocks; 7912 free (0 chunks); 280 used
LOG: level: 1; LOCALOCK hash: 8192 total in 1 blocks; 592 free (0 chunks); 7600 used
LOG: level: 1; GUCMemoryContext: 40960 total in 3 blocks; 28968 free (11 chunks); 11992 used
LOG: level: 2; GUC hash table: 32768 total in 3 blocks; 12704 free (5 chunks); 20064 used
LOG: level: 1; Timezones: 104112 total in 2 blocks; 2648 free (0 chunks); 101464 used
LOG: level: 1; ErrorContext: 8192 total in 1 blocks; 7928 free (4 chunks); 264 used
LOG: Grand total: 1201272 bytes in 183 blocks; 345272 free (139 chunks); 856000 used
```

4) Compare the last line of **Grand total :** with the result **of a query to the pg_backend_memory_contexts view** :

```
postgres=# select sum(total_bytes) total, sum(total_nblocks) blocks, sum(used_bytes) used, sum(free_bytes) free, sum(free_chunks) f_chunks from pg_backend_memory_contexts;
 total | blocks | used | free | f_chunks
---------+---------+---------+---------+---------
1337512 | 197 | 968264 | 369248 | 138
(1 row )
```

The result is similar. The difference is that the query result takes into account the process local memory contexts that were used to execute the query itself.
Logging memory to a log is less convenient than querying a view. Logging with the pg_log_backend_memory_contexts (PID) function **is used to obtain data about the memory contexts of other** processes as they run.

Part 5. Deadlocks during testing

1) Run the commands in the terminal linux :

```
postgres@tantor:~$ psql -c "drop table if exists t cascade;"
psql -c "create table t(pk bigserial, c1 text default 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa');"
psql -c "insert into t select *, 'a' from generate_series(1, 10);"
psql -c "alter table t add constraint pk primary key (pk);"
echo "select * from t for update;" > lock1.sql
echo "update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';" >> lock1.sql
pgbench -T 300 -c 9 -P 5 -f lock1.sql

DROP TABLE
CREATE TABLE
INSERT 0 10
ALTER TABLE
starting vacuum...end.
progress: 5.0 s, 16.4 tps, lat 255.268 ms stddev 571.587, 12 failed
progress: 10.0 s, 15.8 tps, lat 297.406 ms stddev 670.923, 18 failed
progress: 15.0 s, 12.2 tps, lat 464.690 ms stddev 1119.552, 7 failed
progress: 20.0 s, 0.2 tps, lat 5073.576 ms stddev 0.000, 5 failed
progress: 25.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 5 failed
progress: 30.0 s, 6.2 tps, lat 1263.564 ms stddev 2396.412, 9 failed
```

The teams run the test for 5 minutes. The test outputs intermediate results to the terminal every 5 seconds, which is convenient for monitoring the indicators: tps, latency and other indicators.

2) In the second terminal, execute the commands:

```
postgres@tantor:~$ psql -c "alter system set deadlock_timeout = '10s';"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row )
```

The commands increase the value of the deadlock_timeout configuration parameter **to 10 seconds**
.

2) See how the output of the pgbench utility changes:

```
progress: 70.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 75.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 5 failed
progress: 80.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 85.0 s, 3.4 tps, lat 6541.419 ms stddev 7628.907, 1 failed
progress: 90.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
```

Increasing the parameter value had a negative effect on tps. tps decreased to zero.
Try to find out why this happened. After all, increasing the **deadlock_timeout value should** have a beneficial effect on performance.

3) If changing a configuration parameter results in performance degradation, then you need to return the value back or change it in the other direction. Reduce meaning **deadlock_timeout** to 10 milliseconds :

```
postgres@tantor:~$ psql -c "alter system set deadlock_timeout = ' 10ms ';"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row )
```

The changes will not take effect immediately or even within 10 seconds.
Wait until tps increases from zero to ~100 and terminate pgbench by typing `< ctrl + c >` :

```
progress: 205.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 210.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 6 failed
progress: 215.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 220.0 s, 91.8 tps , lat 491.150 ms stddev 5336.318, 63 failed
progress: 225.0 s, 106.4 tps , lat 74.779 ms stddev 35.140, 90 failed
progress: 230.0 s, 88.4 tps , lat 78.530 ms stddev 35.427, 156 failed
progress: 235.0 s, 113.6 tps , lat 74.780 ms stddev 35.355, 40 failed
progress: 240.0 s, 104.0 tps , lat 74.289 ms stddev 34.357, 96 failed
progress: 245.0 s, 106.6 tps , lat 75.671 ms stddev 33.622, 73 failed
^ C
postgres @ tantor :~$
```

4) Reset the deadlock_timeout value **to the** default **.**

```
postgres@tantor:~$ psql -c " alter system reset deadlock_timeout; "
psql -c "select pg_reload_conf();"
ALTER SYSTEM
pg_reload_conf
----------------
t
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
(1 row )
```

5) To find the causes of errors, it is worth looking through the diagnostic log. Look his contents :

```
postgres@tantor:~$ tail -40 $PGDATA/log/postgresql-20*
02:50:31.635 MSK [33653] ERROR: deadlock detected
02:50:31.635 MSK [33653] DETAIL: Process 33653 waits for ShareLock on transaction 1704599; blocked by process 33652.
Process 33652 waits for ShareLock on transaction 1704594; blocked by process 33659.
Process 33659 waits for ExclusiveLock on tuple (18,5) of relation 3013274 of database 5; blocked by process 33653.
Process 33653: select * from t for update;
Process 33652: select * from t for update;
Process 33659: update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';
02:50:31.635 MSK [33653] HINT: See server log for query details.
02:50:31.635 MSK [33653] CONTEXT: while locking tuple (18.5) in relation "t"
02:50:31.635 MSK [33653] STATEMENT: select * from t for update;
02:50:31.646 MSK [33659] ERROR: deadlock detected
02:50:31.646 MSK [33659] DETAIL: Process 33659 waits for ShareLock on transaction 1704599; blocked by process 33652.
Process 33652 waits for ShareLock on transaction 1704594; blocked by process 33659.
Process 33659: update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';
Process 33652: select * from t for update;
02:50:31.646 MSK [33659] HINT: See server log for query details.
02:50:31.646 MSK [33659] CONTEXT: while locking tuple (18.5) in relation "t"
02:50:31.646 MSK [33659] STATEMENT: update t set c1='aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';
02:50:32.622 MSK [33658] LOG: could not send data to client: Broken pipe
02:50:32.622 MSK [33658] FATAL: connection to client lost
02:50:32.662 MSK [33656] LOG: could not send data to client: Broken pipe
02:50:32.662 MSK [33656] FATAL: connection to client lost
02:52:44.350 MSK [32258] LOG: received SIGHUP, reloading configuration files
02:52:44.350 MSK [32258] LOG: parameter "deadlock_timeout" removed from configuration file, reset to default
02:53:26.379 MSK [32260] LOG: checkpoint starting: time
02:53:30.074 MSK [32260] LOG: checkpoint complete: wrote 42 buffers (0.3%); 0 WAL file(s) added, 1 removed, 0 recycled; write=3.615 s, sync=0.030 s, total=3.696 s; sync files=7, longest=0.018 s, average=0.005 s; distance=7573 kB, estimate=85834 kB; lsn=2C/42273240, redo lsn=2C/422731F8
```

The error " `could not send data to client: Broken pipe` " is consequence interruptions works pgbench utilities by pressing keys `^C` .

Errors " `ERROR : deadlock detected` " this is the reason for the decrease in tps when the parameter value increases and vice versa: the increase in tps when the parameter value decreases to 10 milliseconds.

The pgben ch utility reported errors with messages at the end of lines " `failed` ", indicating that the DBMS issued errors. A drop in tps to zero occurs if a set of commands in the test script is executed longer than the interval specified by the parameter " `- P 5` ". In the interval where the script is executed, tps increases and becomes greater than zero.

Part 6. Multitransactions

1) Run the commands:

```
postgres =# drop table if exists t ;
create table t(c int); insert into t values(1);
begin;
select c from t where c=1 for update;
savepoint s1;
update t set c=1 where c=1;
commit;
DROP TABLE
CREATE TABLE
INSERT 0 1
BEGIN
c
---
1
(1 row)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
SAVEPOINT
UPDATE 1
COMMIT
```

The row update command creates a multitransaction.

2) Look at the contents of the block:

```
postgres =# select lp , lp _ off , lp _ len , t _ ctid , t _ xmin , t _ xmax , t
_ ctid , t _ infomask , ( t _ infomask &4096)!=0 as m from heap_page_items (
get_raw_page ( ' t ' , 0 ) ) ;
select * from heap_page('t',0);
lp | lp_off | lp_len | t_ctid | t_xmin | t_xmax | t_ctid | t_infomask | m
----+--------+--------+--------+---------+---------+--------+------------+---
1 | 8144 | 28 | (0,2) | 1695169 | 21016 | (0,2) | 4416 | t
2 | 8112 | 28 | (0,2) | 1695171 | 1695170 | (0,2) | 8336 | f
(2 rows)

lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
8144 | (0,1) | normal | 1695169 c | 21016 | t | | (0,2) | t
8112 | (0,2) | normal | 1695171 | 1695170 | | t | (0,2) | f
(2 rows)
```

The first version of the row `lp=1` has a hint bit set that the row version uses a multitransaction. The row header field `t_xmax` has the number **21016** , which refers to the multitransaction numbering . The HOT chain is also organized, since there are no indexes on the table.
hhu - hint to processes that they should follow the ctid chain.
hot - there are no references to this version of the row from the indexes.
HOT cleanup was not performed because there were no conditions for it to be triggered - there was more than 10% free space in the block.

3) Read the line and look at the contents of the block:

```
postgres=# select * from t;
select * from heap_page('t',0);
c
---
1
(1 row)

lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
8144 | (0,1) | normal | 1695169c | 21016 | t | | (0,2) | t
8112 | (0,2) | normal | 1695171 c | 1695170 | | t | (0,2) | f
(2 rows )
```

When reading the row, the process checked that the transaction that generated the row version `(0,2)` was committed and set the hint bit.

4) Vacuum the table:

```
postgres =# vacuum verbose t ;
INFO: vacuuming "postgres.public.t"
INFO: finished vacuuming "postgres.public.t": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 1 removed , 1 remain, 0 are dead but not yet removable, oldest xmin: 1695172
removable cutoff: 1695172, which was 0 XIDs old when operation ended
new relfrozenxid: 1695172, which is 4 XIDs ahead of previous value
new relminmxid: 21017, which is 1 MXIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 1 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
avg read rate: 0.000 MB/s, avg write rate: 113.499 MB/s
buffer usage: 6 hits, 0 misses, 6 dirtied
WAL usage: 8 records, 6 full page images, 41033 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

The base horizon was not held and the vacuum cleared the block of the old version of the line.

5) Look at the contents of the block after vacuuming:

```
postgres=# select * from heap_page('t',0);
lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+--------------+------+------+-----+-----+--------+-------
2 | (0,1) | redirect to 2 | | | | | | f
8144 | (0,2) | normal | 2c | 0a | | t | (0,2) | f
(2 rows)
```

Signs of c a and xmin =2 mean that the row version was vacuum frozen. The block was cleared of multitransaction numbers.

7) In another terminal with psql, run the commands that will hold the database horizon:

```
postgres=# begin transaction;
select pg_current_xact_id();
BEGIN
pg_current_xact_id
--------------------
            1695208
(1 row )

postgres =*#
```

6) Run the commands that repeat the creation of the row version:

```
postgres=# drop table if exists t;
create table t (c int);
insert into t values(1);
begin;
select c from t where c=1 for update;
savepoint s1;
update t set c=1 where c=1;
commit;
select * from t;
select * from heap_page('t',0);
vacuum verbose t;
select * from heap_page('t',0);
DROP TABLE
CREATE TABLE
INSERT 0 1
BEGIN
c
---
1
(1 row)

SAVEPOINT
UPDATE 1
COMMIT
c
---
1
(1 row)

lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
```

```
--------+-------+--------+----------+---------+-----+-----+--------+-------
8144 | (0,1) | normal | 1695211c | 21025 | t | | (0,2) | t
8112 | (0,2) | normal | 1695213c | 1695212 | | t | (0,2) | f
(2 rows)

INFO: vacuuming "postgres.public.t"
INFO: finished vacuuming "postgres.public.t": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 0 removed, 2 remain , 1 are dead but not yet removable, oldest xmin: 1695208
removable cutoff: 1695208, which was 6 XIDs old when operation ended
new relminmxid: 21026, which is 1 MXIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 2 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 75.120 MB/s
buffer usage: 6 hits, 0 misses, 3 dirtied
WAL usage: 5 records, 3 full page images, 25027 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM

lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
8144 | (0,1) | normal | 1695211c | 1695213 | t | | (0,2) | f
8112 | (0,2) | normal | 1695213 c | 0 a | | t | (0,2) | f
(2 rows )
```

Vacuum failed to clean up any row versions. However, vacuum did clean up the multitransaction data and set the row version freeze flag - the `ca hint bits` .

If a block contains signs of multitransactions, then processes must access buffers whose sizes are specified by the configuration parameters `multixact_member_buffers` and `multixact_offset_buffers to determine the status of transactions included in the multitransaction` . Clearing row versions from multitransactions eliminates the need to access buffers.

7) In the second session, complete the open transaction:

```
postgres= * # commit;
COMMIT
postgres=#
```

8) Complete vacuuming tables :

```
postgres=# vacuum verbose t;
select * from heap_page('t',0);
INFO: vacuuming "postgres.public.t"
INFO: finished vacuuming "postgres.public.t": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 1 removed , 1 remain, 0 are dead but not yet removable, oldest xmin: 1695214
removable cutoff: 1695214, which was 0 XIDs old when operation ended
new relfrozenxid: 1695214, which is 6 XIDs ahead of previous value
frozen: 1 pages from table (100.00% of total) had 1 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 228.659 MB/s
buffer usage: 10 hits, 0 misses, 6 dirtied
WAL usage: 8 records, 6 full page images, 41521 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+---------------+------+------+-----+-----+--------+-------
2 | (0,1) | redirect to 2 | | | | | | f
8144 | (0,2) | normal | 2c | 0a | | t | (0,2) | f
(2 rows)
```

The vacuum cleared the block of old versions that had gone beyond the horizon. `xmin` The frozen version of the row was replaced with transaction number `2` , which is also used as a frozen version indicator.

9) Let's see how freezing changes the row headers . Run commands :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
postgres=# \set ON_ERROR_ROLLBACK interactive \\
truncate table t;
insert into t values(1);
insert into t values(2);
begin;
select * from t for share;
update t set c=1 where c=1;
update t set c=1 where c=1;
commit;
select * from heap_page('t',0);
vacuum freeze t;
select * from heap_page('t',0);
TRUNCATE TABLE
INSERT 0 1
INSERT 0 1
BEGIN
c
---
1
2
(2 rows)

SAVEPOINT
UPDATE 1
UPDATE 1
COMMIT
lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+--------+----------+---------+-----+-----+--------+-------
8144 | (0,1) | normal | 2036130c | 108921 | t | | (0.3) | t
8112 | (0,2) | normal | 2036131c | 2036133 | | | (0,2) | f
8080 | (0.3) | normal | 2036136 | 108922 | t | t | (0.4) | t
8048 | (0.4) | normal | 2036137 | 2036133 | | t | (0.4) | f
(4 rows)

VACUUM
lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
--------+-------+--------------+------+------+-----+-----+--------+-------
4 | (0,1) | redirect to 4 | | | | | | f
8144 | (0,2) | normal | 2 s | 0 a | | | (0,2) | f
0 | (0.3) | unused | | | | | | f
8112 | (0.4) | normal | 2 s | 0 a | | t | (0.4) | f
(4 rows )
```

Two lines are inserted to show that there can be multiple version chains in a block. **for share update** is used to show that other types of locks with savepoints set explicitly and implicitly (**\ set ON_ERROR_ROLLBACK interactive** implicitly sets savepoints after each command in a transaction) generate multitransactions .

Freeze installed hint bits xmin committed and xmax aborted , and xmin=2 . xma x was replaced with 0 to remove multitransaction numbers . Multitransactions have their own number counter.

Part 7. Test example

1) Run the commands in the terminal linux :

```
postgres@tantor:~$
echo " select abalance from pgbench_accounts where bid=1 limit 1; " >bquery.sql
pgbench -n -i --partitions=6 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=7 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=8 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=9 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
pgbench -n -i --partitions=10 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'

latency average = 0.275 ms
tps = 3633.933144 (without initial connection time)
latency average = 0.280 ms
tps = 3570.395697 (without initial connection time)
latency average = 0.291 ms
tps = 3440.218181 (without initial connection time)
latency average = 0.319 ms
tps = 3132.354586 (without initial connection time)
latency average = 0.320 ms
tps = 3124.262538 (without initial connection time)
```

Test from one simple query. I want to test whether the number of relations that will be used when planning a query affects the planning and execution time of a query. To conduct the test, a standardly supplied test table is used, in which the number of sections changes from 5 to 10. The fast path will block 7 or less sections with indexes. For each process, the maximum number of fast blockings is 16.

In the test conducted, the time (`latency`) increases with the increase in the number of sections, and tps decreases proportionally with the increase in the number of sections.

I want to automate the test to eliminate "code duplication". Suddenly I want to make the number of iterations not 5, but more.

2) Run the same commands using a script that automates the execution of tests:

```
postgres@tantor:~$
for (( i=6; i <= 10; i+=1 ))
do
pgbench -n -i --partitions=$1 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
done
latency average = 0.190 ms
tps = 5256.213787 (without initial connection time)
latency average = 0.181 ms
tps = 5539.238994 (without initial connection time)
latency average = 0.178 ms
tps = 5620.314925 (without initial connection time)
latency average = 0.181 ms
tps = 5509.927934 (without initial connection time)
latency average = 0.183 ms
tps = 5471.600964 (without initial connection time)
```

tps have "improved" dramatically and significantly. Moreover, query execution time no longer depends on the number of sections.

I would like to find out what is the reason for the 70% increase in tps and what is the "correct" result.

3) Check that the command in the test runs without errors:

```
postgres@tantor:~$ psql -f bquery.sql
 balance
----------
0
(1 row)
```

The command is normal and runs without errors.

4) Replace the query with another one . In the postgres user terminal , run the following commands:

```
postgres@tantor:~$ echo " select b balance from pgbench_ branches where bid=1
limit 1; " >bquery.sql
for (( i=5; i <= 7; i+=1 ))
do
pgbench -n -i --partitions=$1 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
done
pgbench -n -i --partitions=5 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=6 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
pgbench -n -i --partitions=7 > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
latency average = 0.181 ms
tps = 5539.150039 (without initial connection time)
latency average = 0.181 ms
tps = 5533.754683 (without initial connection time)
latency average = 0.182 ms
tps = 5492.463973 (without initial connection time)
latency average = 0.190 ms
tps = 5253.629658 (without initial connection time)
latency average = 0.181 ms
tps = 5511.707677 (without initial connection time)
latency average = 0.180 ms
tps = 5570.755883 (without initial connection time)
```

When using a different query, the result when running pgbench from within the loop and outside the loop is the same.

5) The reason for the error is in the variable name in the script. Replace the variable with **the correct one** and run the script:

```
postgres@tantor:~$ echo " select abalance from pgbench_accounts where bid=1
limit 1; " >bquery.sql
for (( i=6; i <= 10; i+=1 ))
do
pgbench -n -i --partitions=$ i > /dev/null 2> /dev/null
pgbench -T 10 -f bquery.sql 2> /dev/null | egrep 'latency|tps'
done
latency average = 0.266 ms
tps = 3755.374467 (without initial connection time)
latency average = 0.270 ms
tps = 3704.714460 (without initial connection time)
latency average = 0.291 ms
tps = 3440.145333 (without initial connection time)
latency average = 0.318 ms
tps = 3146.117186 (without initial connection time)
latency average = 0.333 ms
tps = 3006.232748 (without initial connection time)
```

The test started working correctly and the result corresponds to commands without a loop.
Making scripts more complex can lead to typos and errors.

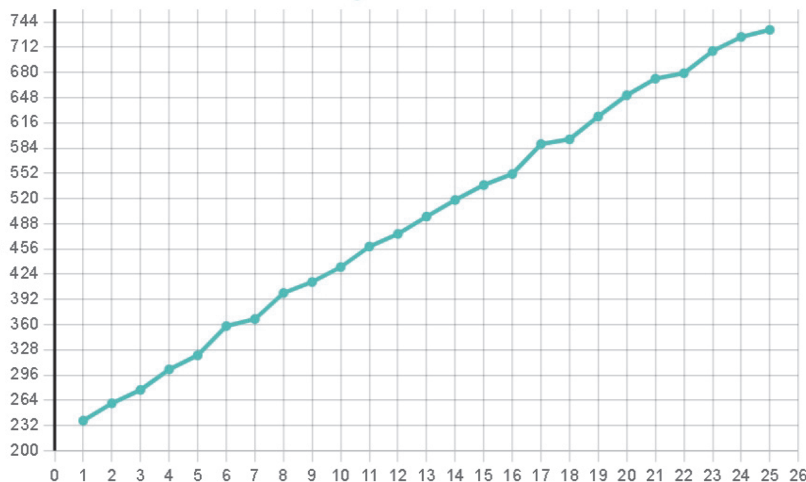
4) For greater accuracy, you can reset the Linux cache and restart the instance between test iterations. You don't have to execute the commands from this point, but look at the example:

```
postgres@tantor:~$ echo "explain (analyze, timing off) select abalance from
pgbench_accounts where bid=1 limit 1; " > bquery.sql
postgres@tantor:~$ su -
Password: root
root@tantor:~#
for (( i=1; i <= 25; i+=1 ))
do
printf "$i"
su - postgres -c "pgbench -n -i --partitions=$i > /dev/null 2> /dev/null"
echo 3 > /proc/sys/vm/drop_caches
systemctl restart tantor-se-server-16
sleep 3
su - postgres -c "pgbench -T 5 -f bquery.sql 2> /dev/null | egrep 'latency|tps'"
done
1 latency average = 0.238 ms
tps = 4200.697407 (without initial connection time)
```

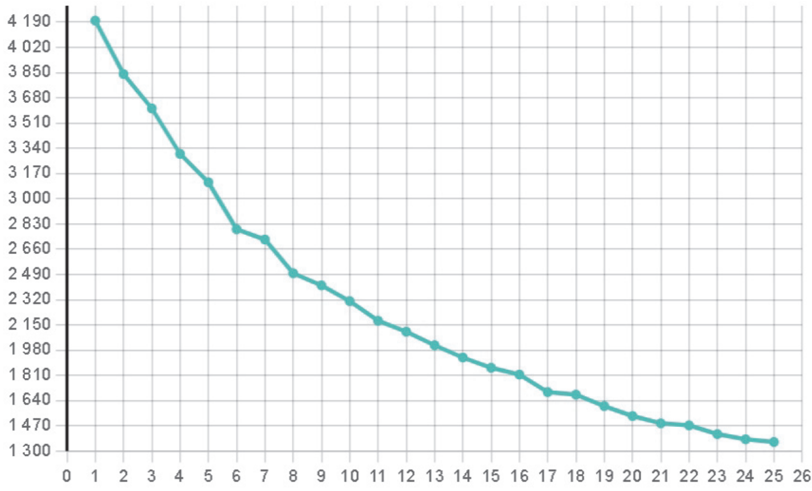
```
2 latency average = 0.260 ms
tps = 3840.529923 (without initial connection time)
3 latency average = 0.277 ms
tps = 3608.223482 (without initial connection time)
4 latency average = 0.303 ms
tps = 3302.631956 (without initial connection time)
5 latency average = 0.321 ms
tps = 3111.183002 (without initial connection time)
6 latency average = 0.358 ms
tps = 2793.766387 (without initial connection time)
7 latency average = 0.367 ms
tps = 2724.507091 (without initial connection time)
8 latency average = 0.400 ms
tps = 2498.189268 (without initial connection time)
9 latency average = 0.414 ms
tps = 2415.747116 (without initial connection time)
10 latency average = 0.433 ms
tps = 2310.439740 (without initial connection time)
11 latency average = 0.459 ms
tps = 2178.293809 (without initial connection time)
12 latency average = 0.475 ms
tps = 2104.057041 (without initial connection time)
13 latency average = 0.497 ms
tps = 2013.104397 (without initial connection time)
14 latency average = 0.518 ms
tps = 1930.415354 (without initial connection time)
15 latency average = 0.537 ms
tps = 1861.375380 (without initial connection time)
16 latency average = 0.551 ms
tps = 1815.541104 (without initial connection time)
17 latency average = 0.589 ms
tps = 1697.304151 (without initial connection time)
18 latency average = 0.595 ms
tps = 1680.943106 (without initial connection time)
19 latency average = 0.624 ms
tps = 1603.523026 (without initial connection time)
20 latency average = 0.651 ms
tps = 1536.011902 (without initial connection time)
21 latency average = 0.672 ms
tps = 1487.413340 (without initial connection time)
22 latency average = 0.679 ms
tps = 1473.400025 (without initial connection time)
23 latency average = 0.707 ms
tps = 1414.174861 (without initial connection time)
24 latency average = 0.725 ms
tps = 1379.219506 (without initial connection time)
25 latency average = 0.734 ms
tps = 1361.741860 (without initial connection time)
root@tantor:~# exit
logout
postgres@tantor:~$
```

When running tests, pay attention to the frequency of checkpoints. At the end of the checkpoint, there will be a drop in performance.

You can plot latency and tps graphs :



"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru



latency graph , which corresponds to the time it takes to create a query plan for a partitioned table depending on the number of partitions, is linear. With a small number of cores and a small load, using fastpath does not affect performance.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Practice for Chapter 10

Part 1. pg_buffercache extension

1) Install pg_buffercache extension :

```
postgres=# create extension pg_buffercache;
CREATE EXTENSION
```

The extension is useful for monitoring the buffer cache and getting general information. The extension is standard and worth using.

2) See what objects are included in the extension:

```
postgres=# \dx+ pg_buffercache
Objects in extension "pg_buffercache"
Object description
-----------------------------------------
function pg_buffercache_pages()
function pg_buffercache_summary()
function pg_buffercache_usage_counts()
view pg_buffercache
(4 rows)
```

pg_buffercache view is wrapper over function `pg_buffercache_pages()` . Since the view selects all columns from the function result, `there is no point in using the pg_buffercache_pages () function .`
The two parameterless functions `pg_buffercache_summary ( ) and` pg_buffercache_usage_counts `( ) do not set locks on the buffer` cache memory structures, unlike the first function and view , `and are fast .`

3) Run the command:

```
postgres = # \ d + pg_buffercache
                           View "public.pg_buffercache"
Column | Type | Collation | Nullable | Default | Storage | Description
------------------+----------+-----------+----------+---------+---------+-------------
 bufferid | integer | | | | plain |
 relfilenode | oid | | | | plain |
 reltablespace | oid | | | | plain |
 reldatabase | oid | | | | plain |
relforknumber | smallint | | | | plain |
relblocknumber | bigint | | | | plain |
isdirty | boolean | | | | plain |
usagecount | smallint | | | | plain |
pinning_backends | integer | | | | plain |
View definition:
SELECT bufferid,
relfilenode,
reltablespace,
reldatabase,
relforknumber,
relblocknumber,
isdirty,
usagecount,
pinning_backends
from pg_buffercache_pages() p(bufferid integer, relfilenode oid, reltablespace oid, reldatabase oid,
relforknumber smallint, relblocknumber bigint, isdirty boolean, usagecount smallint, pinning_backends integer);
```

`bufferid` - buffer ordinal number, starting from 1
The next 5 columns correspond to the BufferTag structure, which specifies the direct (self-sufficient, i.e. storing everything to find the file and the block in it) address of the block on the disk:

reltablespace oid tablespace - name of symbolic link file in PGDATA / pg_tblspc)
reldatabase oid databases - subdirectory name
relfilenode oid name relation file . Names files consist from numbers
relforknumber number fork (0 - main, 1 - fsm, 2 - vm, 4 - init)
relblocknumber block number relative to block zero of the first fork file, maximum 4294967294 (0xFFFFFFFE).

The view reflects the following attributes: pinning, whether the block has changed since it was read from disk (isdirty), the number of processes that have pinned the block.

4) How do I find out how many blocks in the buffer cache are related to tables and indexes? command :

```
postgres=# select relfilinode, count(*) from pg_buffercache group by relfilinode order by 2 desc limit 5;
 relfilenode | count
-------------+-------
 | 15055
1255891 | 2802
1249 | 116
1259 | 60
1255486 | 1698
(5 rows)
```

Unoccupied (free) blocks 15055 . The table name is not **reflected** in **pg_buffercache** , only the file number **.**

5) To get the relation name, a more complex query is needed. Run command :

```
postgres=# SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty) FILTER (WHERE b.isdirty = true) as buffer_dirty
from pg_buffercache b JOIN pg_class c
ON b.relfilenode = pg_relation_filenode(c.oid) AND
b.reldatabase IN ( 0 , (SELECT oid FROM pg_database WHERE datname = current_database()))
JOIN pg_namespace n ON n.oid = c.relnamespace
GROUP BY n.nspname, c.relname
ORDER BY 3 DESC
LIMIT 2;
nspname | relname | buffers | buffer_dirty
------------+------------------+---------+--------------
pg_catalog | pg_class | 2802 | 3
public | pgbench_accounts | 1698 | 0
(2 rows)
```

This query is more convenient. The query does not display rows related to objects of other databases . Database with number 0 - global objects.

6) An even more informative request:

```
postgres=# select relname, buffers, relpages pages, (buffers*8)/1024 as size_mb,
usage_count, dirty, pins,
round(((100*dirty::float8)/nullif(buffers,0)::float8)::numeric,2) as "dirty%",
round(((100*buffers::float8)/nullif(relpages,0)::float8)::numeric,2) as "cached%"
from
(select c.relname, count(*) as buffers, sum(usagecount) as usage_count,
count(*) filter (where isdirty) as dirty, sum(pinning_backends) as pins,
max(c.relpages) as relpages
from pg_buffercache b join pg_class c on b.relfilenode =
pg_relation_filenode(c.oid)
and b.reldatabase IN (0, (select oid
from pg_database where datname = current_database()))
group by c.relname)
order by buffers desc
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
limit 2;

relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
------------------+---------+-------+---------+-------+-------+------+--------+---------
pg_class | 2802 | 2798 | 21 | 14006 | 3 | 1 | 0.11 | 100.14
pgbench_accounts | 1698 | 1695 | 13 | 8490 | 0 | 0 | 0.00 | 100.18
(2 rows)
```

The query determines the size of the relation based on statistics that may be missing. In this case, the cached% column will return an empty value, and pages will return zero.
buffers > pages , since fsm and vm layer blocks are cached, and also in cases of outdated statistics.

7) The request is large. Create a view for convenience:

```
postgres=# create or replace view buffercache as
select relname, buffers, relpages pages, (buffers*8)/1024 as size_mb,
usage_count, dirty, pins,
round(((100*dirty::float8)/nullif(buffers,0)::float8)::numeric,2) as "dirty%",
round(((100*buffers::float8)/nullif(relpages,0)::float8)::numeric,2) as
"cached%"
from
(select c.relname, count(*) as buffers, sum(usagecount) as usage_count,
count(*) filter (where isdirty) as dirty, sum(pinning_backends) as pins,
max(c.relpages) as relpages
from pg_buffercache b join pg_class c on b.relfilenode =
pg_relation_filenode(c.oid)
and b.reldatabase IN (0, (select oid
from pg_database where datname = current_database()))
group by c.relname)
order by buffers desc;
select * from buffercache limit 2;
CREATE VIEW
relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
------------------+---------+-------+---------+-------+-------+------+--------+---------
pg_class | 2802 | 2798 | 21 | 14006 | 3 | 1 | 0.11 | 100.14
pgbench_accounts | 1698 | 1695 | 13 | 8490 | 0 | 0 | 0.00 | 100.18
(2 rows)
```

The view is convenient for viewing the contents of the buffer cache. However, viewing uses access to the buffer headers, which creates a small additional load on a buffer cache of hundreds of gigabytes. The query will work slowly, not because of the number of buffers, but because of competition with other processes for access to the buffer header. By querying the pg_buffercache view **(by dirty buffers) ,** you can estimate whether the checkpoint process will encounter delays when accessing the buffers.
`% cache` The proportion of the cache occupied by object blocks.

8) To get statistics about the buffer cache faster, you can use the following queries:

```
postgres=# select * from pg_buffercache_summary();
buffers_used | buffers_unused | buffers_dirty | buffers_pinned | usagecount_avg
--------------+----------------+---------------+----------------+------------------
1392 | 14992 | 0 | 0 | 4.586206896551724
(1 row)

postgres=# select * buffercache_usage_counts();
usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
           0 | 14992 | 0 | 0
1 | 65 | 0 | 0
2 | 90 | 0 | 0
3 | 11 | 0 | 0
4 | 20 | 0 | 0
5 | 1206 | 0 | 0
(6 rows)
```

is enough data in these views, it is better to use them. Buffers with usage_count = 0 include both free blocks (those in the free list) and those containing a block that has not been used for a long time.

Part 2. Buffer rings

1) Run the commands:

```
postgres=# drop table if exists t;
create table t (id integer, val text storage plain not null) with (autovacuum_enabled =
off);
insert into t select i, 'text number' || i from generate_series(1, 600000) as i;
select * from pg_size_pretty(pg_relation_size('t'));

drop table if exists t1;
create table t1 (id integer, val text storage plain not null) with (autovacuum_enabled =
off);
insert into t1 select i, 'text number' || i from generate_series(1, 700000) as i;
select * from pg_size_pretty(pg_relation_size('t1'));

select * buffercache_usage_counts();
select * from buffercache limit 4;
```

```
DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 600000
pg_size_pretty
----------------
30 MB
(1 row)

DROP TABLE
CREATE TABLE
ALTER TABLE
INSERT 0 700000
pg_size_pretty
----------------
35 MB
(1 row)


usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
0 | 6665 | 0 | 0
1 | 134 | 5 | 0
2 | 57 | 4 | 0
3 | 46 | 1 | 0
4 | 763 | 10 | 0
            5 | 8719 | 8365 | 0
(6 rows)

relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
-----------------+---------+-------+---------+-------+-------+------+--------+---------
t1 | 4492 | 0 | 35 | 22456 | 4490 | 0 | 99.96 |
t | 3850 | 0 | 30 | 19246 | 3848 | 0 | 99.95 |
pg_class | 2802 | 2798 | 21 | 14008 | 2 | 1 | 0.07 | 100.14
pgbench_accounts | 1698 | 1695 | 13 | 8490 | 0 | 0 | 0.00 | 100.18
(4 rows )
```

The commands create two tables. Table t is less than 1/4 of the buffer cache. Table t 1 is greater than 1/4 of the buffer cache.
In the cached % column empty value, and in the pages column zero because the number of rows the view determines from statistics, and they were not collected after the rows were inserted. Statistics are collected by the ANALYZE command.

All table blocks are loaded into the buffer cache and they are all dirty except for a couple of blocks. These pair blocks refers to To fsm fork . Example:

```
postgres=# select bufferid, relforknumber, relblocknumber, usagecount, pinning_backends from pg_buffercache
where relfilnode=3243243 and isdirty=false ;
 bufferid | relforknumber | relblocknumber | usagecount | pinning_backends
----------+---------------+----------------+------------+------------------
 9754 | 1 | 1 | 1 | 0
 9755 | 1 | 0 | 5 | 0
 (2 rows )
```

Why was the buffer ring not used when working with table `t 1` , which is larger than 1/4 of the buffer cache? Because there were no conditions for using the buffer ring.

Conditions for using buffer rings (the algorithm is also called "buffer cache replacement strategy" and "Buffer Access Strategy Type"):

a) BULKREAD. For sequential reading (**using the Seq Scan method**) of **table blocks whose size is greater than or equal to 1/4 of the buffer cache** . The ring size is 256 KB . If a table (not less than 1/4 in size) is already being read in another session, then the sessions wishing to read the table blocks use one buffer ring. We can say that **the sessions are "synchronized"** so as not to read the same blocks again. The other sessions start reading not from the zero block, but from the one that was in the buffer ring of the first session, and then finish reading the blocks from the beginning. Synchronization can be disabled by the `synchronize_seqscans configuration parameter` , but you should not do this.

b) VACUUM. Dirty pages are not removed from the ring, but sent for writing. The ring size is set by the vacuum_buffer_usage_limit configuration parameter. By default, the ring size is 256 KB . Autovacuum also uses a buffer ring (in the aggressive autovacuum mode, the buffer ring is not used).

c) BULKWRITE . **Always used by COPY and CREATE commands. TABLE AS SELECT regardless of data volume and table sizes** . Ring size 16MB .

If a buffer becomes dirty, it is excluded from the buffer ring.

In a buffer cache, a block can only reside in one buffer.

Table buffers have **`usage_count =5`** . The usage_count column `of` **`the`** `buffercache` **`view is`** not informative, it does not show `how` many times the buffer was accessed, it is the sum **`( usagecount ) for`** each block, and **`usagecount`** is limited to 5.

2) Do it commands :

```
postgres=# drop table if exists t2;
create table t2 as select * from t1;
select * from pg_size_pretty(pg_relation_size('t2'));
select * buffercache_usage_counts();
select * from buffercache limit 4;
DROP TABLE
SELECT 700000
pg_size_pretty
----------------
36 MB
(1 row)

usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
0 | 132 | 0 | 0
1 | 2125 | 1998 | 0
2 | 31 | 1 | 0
3 | 5672 | 8 | 0
4 | 8416 | 4777 | 0
5 | 8 | 6 | 0
(6 rows)

relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
---------+---------+-------+---------+--------+-------+------+--------+---------
t1 | 4492 | 0 | 35 | 17965 | 4490 | 0 | 99.96 |
t | 3850 | 0 | 30 | 15396 | 271 | 0 | 7.04 |
pg_class | 2802 | 2798 | 21 | 8410 | 5 | 1 | 0.18 | 100.14
t2 | 2048 | 0 | 16 | 2048 | 1992 | 0 | 97.27 |
(4 rows )
```

When creating the table `t 2,` a buffer ring was used. The ring size is 16MB = 2048 blocks . Of these, **1992 blocks are dirty** . The remaining blocks were sent for writing and evicted from the buffer ring (and, accordingly, the buffer cache). Buffer rings are a set of blocks in the buffer cache, not a separate memory structure.

3) Do it commands :

```
postgres=# VACUUM( ANALYZE ) t1;
VACUUM( ANALYZE , BUFFER_USAGE_LIMIT 0) t2;
select * from buffercache limit 4;
VACUUM
VACUUM
relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
----------+---------+-------+--------+-------+-------+------+--------+---------
t1 | 4492 | 4488 | 35 | 4 | 4406 | 0 | 98.09 | 100.09
t | 3849 | 0 | 30 | 0 | 0 | 0 | 0.00 |
pg_class | 2800 | 2798 | 21 | 8393 | 4 | 1 | 0.14 | 100.07
t2 | 2321 | 4488 | 18 | 2117 | 5 | 0 | 0.22 | 51.72
(4 rows)
```

BUFFER_USAGE_LIMIT parameter sets size buffer rings instead of parameter vacuum_buffer_usage_limit configuration parameter . Unlike the configuration parameter, BUFFER_USAGE_LIMIT can be set to zero and the buffer ring will not be used. The value parameter By default :

```
postgres=# \dconfig *usage*
List of configuration parameters
Parameter | Value
---------------------------+-------
vacuum_buffer_usage_limit | 256kB
(1 row)
```

4) Run the commands:

```
postgres=# copy t1 to '/tmp/t1';
truncate table t2;
select * from buffercache limit 4;
begin;
copy t2 from '/tmp/t1' with FREEZE ;
end;
COPY 700000
TRUNCATE TABLE
relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
------------------+---------+-------+---------+-------+-------+------+--------+---------
t1 | 4492 | 4488 | 35 | 4491 | 0 | 0 | 0.00 | 100.09
t | 3849 | 0 | 30 | 0 | 0 | 0 | 0.00 |
pg_class | 2800 | 2798 | 21 | 13990 | 1 | 1 | 0.04 | 100.07
pgbench_accounts | 1698 | 1695 | 13 | 0 | 0 | 0 | 0.00 | 100.18
(4 rows)

BEGIN
ERROR: cannot perform COPY FREEZE because the table was not created or truncated in the
current subtransaction
ROLLBACK
```

`COPY` command with `the FREEZE option` can only be executed in the transaction in which the table was created or truncated.

From a performance point of view, using the FREEZE option is useful because the rows are frozen and autovacuum will not make an extra pass through all the table blocks after some time . When loading rows into TOAST, they are not frozen (https://github.com/postgres/postgres/commit/8e03eb92).

5) Run the commands:

```
postgres=# begin;
truncate t2;
copy t2 from '/tmp/t1' with FREEZE ;
commit;
select * from buffercache limit 4;
BEGIN
COPY 700000
COMMIT
relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
----------+---------+-------+---------+-------+-------+------+--------+---------
t1 | 4493 | 4488 | 35 | 4499 | 0 | 0 | 0.00 | 100.11
t | 3849 | 0 | 30 | 0 | 0 | 0 | 0.00 |
pg_class | 2799 | 2798 | 21 | 13990 | 1 | 1 | 0.04 | 100.04
t2 | 2049 | 0 | 16 | 2053 | 2009 | 0 | 98.05 |
(4 rows )
```

COPY command with the FREEZE option was executed in the same transaction as the truncate table command . The buffer ring was used during the load . In the buffer cache, the table blocks use 2049 buffers, and the table has 4493 blocks - the same as table t1.

6) Check if the BULKWRITE buffer ring is used if the table size is less than 1/4 of the buffer cache. Save the rows of table t, the size of which is less than 1/4 of the buffer cache and load them into the tables. Execute commands :

```
postgres=# copy t to '/tmp/t1';
truncate table t;
copy t from '/tmp/t1';
begin;
truncate table t2;
copy t2 from '/tmp/t1';
commit;
select * from buffercache limit 4;
COPY 600000
TRUNCATE TABLE
COPY 600000
BEGIN
TRUNCATE TABLE
COPY 600000
COMMIT
relname | buffers | pages | size_mb | usage | dirty | pins | dirty% | cached%
----------+---------+-------+---------+-------+-------+------+--------+---------
t1 | 4493 | 4488 | 35 | 4499 | 0 | 0 | 0.00 | 100.11
pg_class | 2799 | 2798 | 21 | 13990 | 1 | 1 | 0.04 | 100.04
t | 2052 | 0 | 16 | 2068 | 2023 | 0 | 98.59 |
t2 | 2048 | 0 | 16 | 2048 | 2007 | 0 | 98.00 |
(4 rows )
```

BULKWRITE buffer ring is used and does not depend on the size of the data being loaded or the size of the tables. It is always used by the COPY and CREATE commands. TABLE AS SELECT .

Part 3. pg_prewarm extension

1) Install the pg_prewarm extension:

```
postgres=# alter system set shared_preload_libraries='pg_prewarm';
ALTER SYSTEM
postgres=# \q
postgres@tantor:~$ sudo restart
postgres@tantor:~$ psql
postgres=# create extension pg_prewarm;
CREATE EXTENSION
```

2) See what objects are included in the pg_prewarm extension:

```
postgres=# \dx+ pg_prew  arm
Objects in extension "pg_prewarm"
Object description
-------------------------------------------------------
function autoprewarm_dump_now()
function autoprewarm_start_worker()
function pg_prewarm(regclass,text,text,bigint,bigint)
(3 rows)
```

3) The extension has configuration parameters:

```
postgres=# \dconfig *prewarm*
List of configuration parameters
Parameter | Value
---------------------------------+-------
pg_prewarm.autoprewarm | on
pg_prewarm.autoprewarm_interval | 5min
(2 rows)
```

After the instance is restarted, the `autoprewarm background process is started. leader` , which, while the instance is running or when the instance is stopped, saves data in the background to the autoprewarm . blocks file located in the root of the PGDA T A cluster directory . When the instance is restarted , `the leader process` sorts the list of blocks from the file and starts `bgworker` , which sequentially connects to the databases and loads the blocks of database objects into the buffer cache. `pg_prewarm.autoprewarm` parameter can be used to disable the leader process startup. Changing this parameter requires restarting the instance. It is impossible to start or stop the background process while the instance is running, `the autoprewarm_start_worker() function` is useless. The frequency of saving data to the file is set by the `pg_prewarm.autoprewarm_interval parameter` . The default is 5 minutes. If you set it to zero, the file with the list of blocks will not be updated.

`The pg_prewarm` extension function `( regclass , text , text , bigint , bigint ) allows` you to load object blocks into the buffer cache:

```
postgres=# select pg_prewarm('pg_class', 'prefetch');
 pg_prewarm
------------
758
(1 row)
```

, it is enough to specify one parameter with the object name : `pg_prewarm ( ' pg_class ' ) .`

5) The second parameter of the `pg_prewarm function ( regclass , text , text , bigint , bigint ) is` called mode :

```
postgres=# \sf pg_prewarm
CREATE OR REPLACE FUNCTION public.pg_prewarm( reg class , mode text DEFAULT ' buffer
'::text, fork text DEFAULT ' main '::text, first_block bigint DEFAULT NULL::bigint,
last_block bigint DEFAULT NULL::bigint)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
RETURNS bigint
LANGUAGE c
PARALLEL SAFE
AS '$libdir/pg_prewarm', $function$pg_prewarm$function$
```

Parameter `mode text DEFAULT ' buffer '::text` Can specify way downloads blocks :
`buffer` (default value) to buffer cache
`read` only to page cache by synchronous call
`prefetch only to` page cache by calling posix_fadvise (.. , POSIX_FADV_WILLNEDED))

If the buffer cache is empty, loading will happen quickly. If the buffer cache is full, then blocks of other objects will have to be unloaded (eviction) and new blocks will have to be loaded in their place. To avoid this, you can use `mode = prefetch` . The blocks will be loaded into the operating system cache and if the blocks need to be accessed, the blocks will be quickly copied to the buffer cache. Only blocks loaded into the buffer cache are saved in `the autoprewarm . blocks file` . Information about loading blocks into the operating system cache is not saved anywhere.

6) Look at the beginning of the extension file:

```
postgres =# \! head -3 $ PGDATA / autoprewarm . blocks
<<1269>>
0 , 1664 , 1255633 , 0 , 0
0,1664,1255462,0,0
```

The first line of the file contains the number of blocks whose addresses are stored in the file. The block address is in the format of the standard `BufferTag` structure . The structure format allows you to find a block on the disk without accessing any additional data. The structure has 5 fields corresponding to the columns in the extension file:
`oid` tablespace - name of symbolic link file in PGDATA / pg_tblspc)
`oid` databases - subdirectory name
file name relation
number fork (`0 - main, 1 - fsm, 2 - vm, 4 - init`)
number relative to block zero of the first fork file, maximum 4294967294 (0xFFFFFFFE).

7) Delete file :

```
postgres=# \! rm $PGDATA/autoprewarm.blocks
postgres=# \! head -3 $PGDATA/autoprewarm.blocks
head: cannot open '/var/lib/postgresql/tantor-se-16/data/autoprewarm.blocks' for reading: No such file or directory
```

8) Call the function that creates or updates the file:

```
postgres=# select autoprewarm_dump_now();
autoprewarm_dump_now
----------------------
1269
(1 row )
```

The function updates or creates the autoprewarm.blocks file. This can be useful if the leader process is not running, but you want the blocks to be loaded into the buffer cache after the instance is restarted. The function returns the number of blocks present in the autoprewarm.blocks file.

9) The function returned the number of blocks in the file:

```
postgres=# \! head -3 $PGDATA/autoprewarm.blocks
<< 1269 >>
0,1664,1255633,0,0
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
0,1664,1255462,0,0
```

When the instance is restarted, 1269 blocks will be downloaded in the background into the buffer cache.

Part 4. The bgwriter background writing process

The background writer sends the buffer contents for writing (but does not evict them from the buffer cache) if the buffer header contains indications that the buffer:
a) dirty
b) the block is not damaged
c) refcount=0, there are no block pins, i.e. the block is not needed by processes
d) usage_count=0 (falls into the gradation of those not used for a long time).
After sending the buffer contents for writing, the buffer is not included in the free list, the buffer becomes clean (the indicator that the buffer is dirty is removed).

1) See how many blocks the controller can read in advance (read-ahead) from the block device where the cluster directory is located:

```
postgres@tantor:~$ sudo blockdev --getra /dev/sda
256
```

The controller can read 256 blocks. The unit of measurement is 512 bytes. The controller can read up to 128Kb in advance.
Vacuum, analysis, startup process use preliminary reading of blocks (prefetch). The pg_prewarm extension can use prefetch. The maximum number of buffers is set by the configuration parameter maintenance_io_concurrency.

2) Look at the value of the `maintenance_io_concurrency parameter` :

```
postgres=# show maintenance_io_concurrency;
maintenance_io_concurrency
----------------------------
10
(1 row)
```

Meaning By default 10 blocks of 8Kb, which corresponds to 80Kb.
The first and second points of practice are needed for you to pay attention to these parameters, which are described in the theoretical part.

4) Look at the parameters bgwriter configurations :

```
postgres=# select name, setting, context, max_val, min_val from pg_settings
where name ~ ' bgwr ';
name | setting | context | max_val | min_val
--------------------------+----------+---------+-----------+---------
 bgwr iter_delay | 200 | sighup | 10000 | 10
 bgwr iter_flush_after | 64 | sighup | 256 | 0
 bgwr iter_lru_maxpages | 100 | sighup | 1073741823 | 0
 bgwr iter_lru_multiplier | 2 | sighup | 10 | 0
(4 rows )
```

This is a convenient query to view the minimum, maximum, and other details of configuration parameters. `The show` and `\dconfig commands` do not provide these details.

bgwriter_delay - how many milliseconds bgwriter sleeps between iterations.
bgwriter_flush_after - the number of blocks after sending for writing which flush of the linux page cache is initiated . Zero disables flush .
The number of dirty buffers written in an iteration depends on how many blocks were loaded into the buffer cache by server processes (recently ly allocated) in previous cycles. The average value is multiplied by bgwriter_lru_multiplier and specifies how many buffers need to be cleared in the current cycle. The process with the maximum speed tries to reach this value, but not more than bgwriter_lru_maxpages . bgwriter_lru_maxpages is the maximum number of blocks written in one iteration; if the value is zero, bgwriter stops working.

5) Do it request :

```
postgres=# select * from pg_stat_bgwriter\gx
-[ RECORD 1 ]--------+-----------------------------
checkpoints_timed | 707
checkpoints_req | 173
checkpoint_write_time | 11877275
checkpoint_sync_time | 3842836
buffers_checkpoint | 1418992
buffers_clean | 88732
maxwritten_clean | 310
buffers_backend | 8050175
buffers_backend_fsync | 0
buffers_alloc | 3158495
stats_reset | 2035-01-01 01:01:59.220537+03
```

Displays statistics about the performance of bgwriter for the entire instance. The view contains one row.
buffers_clean - how many buffers were sent for writing (" clean ") by bgwriter.
maxwritten_clean - how many times bgwriter paused because it reached bgwriter_lru_maxpages or the estimated number of blocks (taking into account bgwriter_lru_multiplier) if bgwriter_lru_maxpages was already set to the maximum value.
stats_reset is the date and time when the data in this view was reset. You can reset the data in a view using the pg_stat_reset_shared('bgwriter') function . **Calling the** pg_stat_reset() function without parameters does not reset the statistics in either this view or the pg_stat_io view .
buffers_backend - how many buffers were cleared (sent for writing) by server processes and autovacuum.

In version 17, the view has few columns:

```
postgres=# \! sudo docker exec -it postgres psql -U postgres -c "select * from pg_stat_bgwriter"
 buffers_clean | maxwritten_clean | buffers_alloc | stats_reset
---------------+------------------+---------------+----------------------------
 0 | 0 | 176 | 2025-01-03 21:27:38.13624+00
 (1 row)
```

Some columns have been moved to the pg_stat_checkpointer view :

```
postgres=# \! sudo docker exec -it postgres psql -U postgres -c "select * from pg_stat_checkpointer"
 num_timed | num_requested | restartpoints_timed | restartpoints_req | restartpoints_done | write_time | sync_time | buffers_written | stats_reset
----------+--------------+--------------------+------------------+-----+-----------+----------+----------------
 785 | 2 | 0 | 0 | 0 | 35 | 9 | 0 | 2035-01-01 01:01:38.13624+00
 (1 row )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Values similar to those displayed in the `buffers_backend column` in version 17 are suggested to be viewed in `the pg_stat_io view .`

6) It is convenient to view indicators not in absolute values, but as a percentage:

```
postgres=# create view pgstatbgwriter as select to_char(100*checkpoints_timed::numeric /
nullif((checkpoints_timed+checkpoints_req),0),'990D9')
 || ' %' "ckpt by time",
 to_char(100*checkpoints_req::numeric / nullif((checkpoints_timed+checkpoints_req),0),'990D9')
 || ' %' AS "ckpt by size",
 to_char(100*buffers_checkpoint::numeric / nullif((buffers_checkpoint+buffers_clean +
buffers_backend),0),'990D9')
 || ' %' "checkpointer",
 to_char(100*buffers_backend::numeric / nullif((buffers_checkpoint+buffers_clean +
buffers_backend),0),'990D9')
 || ' %' "backend",
 to_char(100*buffers_backend_fsync::numeric / nullif((buffers_checkpoint+buffers_clean +
buffers_backend),0),'990D9')
 || ' %' "backend_fsync",
 to_char(100*buffers_clean::numeric / nullif((buffers_checkpoint+buffers_clean +
buffers_backend),0),'990D9')
 || ' %' "bgwriter",
 pg_size_pretty((buffers_checkpoint+buffers_clean+buffers_backend)*8192/ (extract (epoch from
current_timestamp - stats_reset))::bigint) || '/s' "speed"
 FROM pg_stat_bgwriter;
 select * from pgstatbgwriter;
 CREATE VIEW
 ckpt by time | ckpt by size | checkpointer | backend | backend_fsync | bgwriter | speed
 -------------+--------------+--------------+---------+---------------+----------+------------
 80.5% | 19.5% | 14.8% | 84.2% | 0.0% | 0.9% | 338 kB/s
 (1 row)
```

7) Another convenient query for version 16:

```
SELECT
clock_timestamp()-pg_postmaster_start_time() "Uptime",
clock_timestamp()-stats_reset "Since stats reset",
round(100.0*checkpoints_req/total_checkpoints,1) "Forced checkpoint ratio (%)",
round(np.min_since_reset/total_checkpoints,2) "Minutes between checkpoints",
round(checkpoint_write_time::numeric/(total_checkpoints*1000),2) "Average write time per checkpoint (s)",
round(checkpoint_sync_time::numeric/(total_checkpoints*1000),2) "Average sync time per checkpoint (s)",
round(total_buffers/np.mp,1) "Total MB written",
round(buffers_checkpoint/(np.mp*total_checkpoints),2) "MB per checkpoint",
round(buffers_checkpoint/(np.mp*np.min_since_reset*60),2) "Checkpoint MBps",
round(buffers_clean/(np.mp*np.min_since_reset*60),2) "Bgwriter MBps",
round(buffers_backend/(np.mp*np.min_since_reset*60),2) "Backend MBps",
round(total_buffers/(np.mp*np.min_since_reset*60),2) "Total MBps",
round(1.0*buffers_alloc/total_buffers,3) "New buffer allocation ratio",
round(100.0*buffers_checkpoint/total_buffers,1) "Clean by checkpoints (%)",
round(100.0*buffers_clean/total_buffers,1) "Clean by bgwriter (%)",
round(100.0*buffers_backend/total_buffers,1) "Clean by backends (%)",
round(100.0*maxwritten_clean/(np.min_since_reset*60000/np.bgwr_delay),2) "Bgwriter halt-only length
(buffers)",
coalesce(round(100.0*maxwritten_clean/(nullif(buffers_clean,0)/np.bgwr_maxp),2),0) "Bgwriter halt ratio (%)",
'--------------------------------' "--------------------------------",
bgstats.*
FROM (
SELECT bg.*,
checkpoints_timed + checkpoints_req total_checkpoints,
buffers_checkpoint + buffers_clean + buffers_backend total_buffers,
pg_postmaster_start_time() startup,
current_setting('checkpoint_timeout') checkpoint_timeout,
current_setting('max_wal_size') max_wal_size,
current_setting('checkpoint_completion_target') checkpoint_completion_target,
current_setting('bgwriter_delay') bgwriter_delay,
current_setting('bgwriter_lru_maxpages') bgwriter_lru_maxpages,
current_setting('bgwriter_lru_multiplier') bgwriter_lru_multiplier
FROM pg_stat_bgwriter bg
) bgstats,
(
SELECT
round(extract('epoch' from clock_timestamp() - stats_reset)/60)::numeric min_since_reset,
(1024 * 1024 / block.setting::numeric) mp,
delay.setting::numeric bgwr_delay,
lru.setting::numeric bgwr_maxp
FROM pg_stat_bgwriter bg
JOIN pg_settings lru ON lru.name = 'bgwriter_lru_maxpages'
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
JOIN pg_settings delay ON delay.name = 'bgwriter_delay'
JOIN pg_settings block ON block.name = 'block_size'
) np\gx
-[ RECORD 1 ]-----------------------+-----------------------------
Uptime | 10:05:12.155806
Since stats reset | 2 days 14:55:23.335427
Forced checkpoint ratio (%) | 19.5
Minutes between checkpoints | 4.26
Average write time per checkpoint (s) | 13.39
Average sync time per checkpoint (s) | 4.33
Total MB written | 74671.1
MB per checkpoint | 12.50
Checkpoint MBps | 0.05
Bgwriter MBps | 0.00
Backend MBps | 0.28
Total MBps | 0.33
New buffer allocation ratio | 0.330
Clean by checkpoints (%) | 14.8
Clean by bgwriter (%) | 0.9
Clean by backends (%) | 84.2
Bgwriter halt-only length (buffers) | 0.03
Bgwriter halt ratio (%) | 34.94
------------------------------ | ------------------
checkpoints_timed | 714
checkpoints_req | 173
checkpoint_write_time | 11877275
checkpoint_sync_time | 3842836
buffers_checkpoint | 1418992
buffers_clean | 88732
maxwritten_clean | 310
buffers_backend | 8050175
buffers_backend_fsync | 0
buffers_alloc | 3158508
stats_reset | 2035-01-01 01:07:59.220537+03
total_checkpoints | 887
total_buffers | 9557899
startup | 2035-01-01 01:58:10.400156+03
checkpoint_timeout | 5min
max_wall_size | 1GB
checkpoint_completion_target | 0.9
bgwriter_delay | 200ms
bgwriter_lru_maxpages | 100
bgwriter_lru_multiplier | 2
```

8) Run the query using the `pg_buffercache extension function` :

```
postgres=# select * buffercache_usage_counts();
usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
     0 | 6294 | 0 | 0
1 | 4243 | 0 | 0
2 | 46 | 0 | 0
3 | 37 | 0 | 0
4 | 30 | 0 | 0
5 | 5734 | 0 | 0
(6 rows)
```

Function gives instant picture Togo There is li Job for bgwriter or He will work in vain , only addressing To buffers . bgwriter can clear blocks , u which simultaneously By data performances `pg_buffercache isdirty=true, usagecount=0, pinning_backends=0` **.** According **to** the pg_buffercache_usage_counts() function, **such blocks are counted in** the first row (**usage_count** =0) - the more dirty and the less `pinned` , the more likely it is that the bgwriter process has something to clean. If the buffers column `contains zeros in the upper rows, and most of the blocks are in the last rows (usage_count=5 and` usage_count = 4) , then the probability that bgwriter has (will have in the near future) something to clean is small .

This view is useful because the `buffers column` shows whether the buffers are evenly distributed across the usage_count gradations `given the current` activity and instance settings. The distribution of buffers across the six usage_count "buckets" should not be clearly skewed in one direction or another. If there is a skew, the eviction algorithm loses its effectiveness.

`the buffers` column is skewed towards `usage_count=5,` it means that the buffer cache size is insufficient. If most of the buffers are in the row with `usage_count=1, the buffer cache size`

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

can be reduced. The usage_count=0 row takes into account free buffers. Free buffers are present after an instance restart or deletion, truncation of tables, databases. If the instance has been running for a long time, the buffer cache is completely filled. If there is a large number of buffers in the usage_count=0 row , either the instance has just rebooted and most of the buffers are free, or the buffer cache size can be reduced.

It is better to configure bgwriter parameters after configuring the buffer cache size and configuring the checkpoint process.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Practice for Chapter 11

Part 1. Setting the frequency of checkpoints

1) Look at the configuration parameters of the checkpointer process:

```
postgres=# select name, setting, unit, context, max_val, min_val from
pg_settings where name ~ 'checkpoint' or name='max_wal_size';
 name | setting | unit | context | max_val | min_val
------------------------------+---------+------+---------+------------+---------
checkpoint_completion_target | 0.9 | | sighup | 1 | 0
checkpoint_flush_after | 32 | 8kB | sighup | 256 | 0
 checkpoint_timeout | 300 | s | sighup | 86400 | 30
checkpoint_warning | 30 | s | sighup | 2147483647 | 0
log_checkpoints | on | | sighup | |
 max_wal_size | 1024 | MB | sighup | 2147483647 | 2
(6 rows )
```

By default, `log_checkpoints = on and` checkpoint execution messages are written to the diagnostic log .
checkpoint_timeout defines the intervals **between** checkpoints, but the checkpoint will start earlier if the **WAL size approaches max_wal_size .**

2) Create scripts for the test:

```
postgres @ tantor :~$ mcedit ckpt.sql

\pset tuples_only
\getenv t3 t3
\echo :t3
\o ckpt.tmp
alter system set checkpoint_timeout = :t3;
select pg_reload_conf();
select pg_current_wal_lsn() AS t1 \gset
\! sleep $t3
select pg_current_wal_lsn() AS t2 \gset
\o
select pg_size_pretty((:'t2'::pg_lsn - :'t1'::pg_lsn)/:'t3');
select * buffercache_usage_counts();
```

```
postgres@tantor:~$ mcedit ckpt.sh

for (( i=60; i <= 300; i+=60 ))
do
export t3="$i"
psql -f ckpt.sql
done
```

```
postgres@tantor:~$ mcedit ckpt-horizon.sh

for (( i=60; i <= 300; i+=60 ))
do
export t3="$i"
psql -f ckpt-horizon.sql
done
```

```
postgres@tantor:~$ mcedit ckpt-horizon.sql

\pset tuples_only
```

tantor

```
\getenv t3 t3
\echo :t3
\o ckpt.tmp
alter system set checkpoint_timeout = :t3;
select pg_reload_conf();
select pg_current_wal_lsn() AS t1 \gset
select pg_sleep(:t3);
select pg_current_wal_lsn() AS t2 \gset
\o
select pg_size_pretty((:'t2'::pg_lsn - :'t1'::pg_lsn)/:'t3');
```

```
postgres@tantor:~$ chmod +x ckpt.sh
chmod + x ckpt - horizon .sh
```

3) In the first terminal window, run the standard test:

```
postgres@tantor:~$ pgbench -i
dropping old tables...
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.13 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.45 s (drop tables 0.03 s, create tables 0.02 s, client-side generate
0.19 s, vacuum 0.09 s, primary keys 0.11 s).
postgres@tantor:~$ pgbench -c 4 -j 10 -T 6000 -P 30
starting vacuum...end.
progress: 30.0 s, 220.2 tps, lat 18.154 ms stddev 6.414, 0 failed
progress: 60.0 s, 190.7 tps, lat 20.974 ms stddev 7.433, 0 failed
```

The test time is specified with a reserve of 6000 seconds. When the load is no longer needed, you can interrupt the test with the key combination < ctrl + c >

4) In the second terminal window, run the test without holding the horizon.

```
postgres @ tantor :~$ ./ ckpt . sh
```

While the test is running, look at the script text to understand what the test is doing. The test increases the checkpoint frequency from 30 seconds to 5 minutes with one-minute intervals. The goal of the test is to find the optimal interval. Since the default checkpoint interval is 5 minutes, the recommended is 20 minutes, then the real test would take a long time. In real testing, you can make the interval 5-10 minutes from 5 minutes to 1-2 hours, and set the load time deliberately long, for example, 600000 seconds.

For the purposes of the course, the metrics trend (metric decreases, increases, does not change), testing methodology, and set of metrics are important.

Launching and running a script helps to remember the topic better. If you do not launch scripts, do not type commands on the keyboard, do not get errors and do not think about how to fix them, then the material being studied will not cause emotions (interest, joy from the fact that you managed to run the test, frustration due to typos), thanks to which the material is remembered.

The script changes the checkpoint frequency, rereads the configuration, measures how many WAL records were created , and outputs statistics on the buffer cache.

Increasing the interval between checkpoints reduces the volume of WAL records, which improves performance.

During the test, you can, if you wish, use queries from previous practice.

In the pgbench terminal window tps and latency do not change:

```
progress: 210.0 s, 192.7 tps, lat 20.754 ms stddev 7.556, 0 failed
progress: 240.0 s, 193.7 tps, lat 20.653 ms stddev 7.712, 0 failed
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

tantor

```
progress: 270.0 s, 194.4 tps, lat 20.569 ms stddev 8.251, 0 failed
```

If tps does not change, does it mean that the frequency of checkpoints does not affect performance?
No, it does not. In the load test, short transactions, almost all old rows are cleared by HOT cleanup.

An example of test results that will be output to the terminal:

```
postgres@tantor:~$ ./ckpt.sh &
[1] 137305
postgres@tantor:~$ Tuples only is on.
60
 339 kB

0 | 9883 | 0 | 0
1 | 350 | 0 | 0
2 | 88 | 0 | 0
3 | 36 | 0 | 0
4 | 30 | 1 | 0
5 | 5997 | 1753 | 2

Tuples is only on.
120
 234 kB

0 | 9727 | 0 | 0
1 | 348 | 0 | 0
2 | 88 | 0 | 0
3 | 35 | 0 | 0
4 | 23 | 0 | 0
5 | 6163 | 2014 | 2

Tuples is only on.
180
 192 kB

0 | 9471 | 0 | 0
1 | 348 | 0 | 0
2 | 86 | 0 | 0
3 | 37 | 1 | 0
4 | 22 | 0 | 0
5 | 6420 | 2186 | 1

Tuples is only on.
240
167 kB

0 | 9141 | 0 | 0
1 | 371 | 0 | 0
2 | 86 | 0 | 0
3 | 34 | 0 | 0
4 | 24 | 0 | 0
5 | 6728 | 2254 | 2

Tuples is only on.
300
 158 kB

0 | 8753 | 0 | 0
1 | 371 | 0 | 0
2 | 88 | 2 | 0
3 | 34 | 0 | 0
4 | 22 | 0 | 0
5 | 7116 | 2355 | 1

Tuples is only on.
360
150 kB

0 | 8281 | 0 | 0
1 | 371 | 0 | 0
2 | 87 | 0 | 0
3 | 34 | 0 | 0
4 | 22 | 0 | 0
5 | 7589 | 2462 | 2

Tuples is only on.
420
 154 kB

           0 | 7735 | 0 | 0
1 | 371 | 0 | 0
2 | 87 | 0 | 0
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
3 | 34 | 0 | 0
4 | 22 | 0 | 0
5 | 8135 | 2584 | 1

Tuples is only on.
480
```

Distribution of blocks is not interesting: the test is very simple and blocks accumulate in usage_count=5. The number of free blocks decreases little by little (usage_count = 0). It can be concluded that the size of the buffer cache (128MB) is large for the load (standard pgbench test).
What is interesting is that as the interval between checkpoints increases, the amount of WAL writing decreases .

When you have finished familiarizing yourself with the text of the script, the task, and the test results, stop running the test:

```
postgres@tantor:~$ kill -9 137305
postgres@tantor:~$ ps
PID TTY TIME CMD
137300 pts/1 00:00:00 bash
137593 pts/1 00:00:00 psql
137595 pts/1 00:00:00 sleep
137607 pts/1 00:00:00 ps
[1]+ Killed ./ckpt.sh
postgres@tantor:~$ <ctrl+d>
logout
astra@tantor:~$ su - postgres
Password: postgres
postgres@tantor:~$
```

5) In the second terminal window, run the script with horizon hold.

```
postgres@tantor:~$ ./ckpt-horizon.sh &
[2] 138802
postgres@tantor:~$ Tuples only is on.
60
```

The horizon is held at each iteration of the cycle and the duration of the hold is until the configuration is reread. That is, the duration of the horizon hold is equal to the current interval between checkpoints.

6) In the third terminal window, you can run psql and check that the horizon is being held:

```
postgres=# select age(backend_xmin), extract(epoch from (clock_timestamp()-
xact_start)) secs, pid, datname database, state from pg_stat_activity where
backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
 age | secs | pid | database | state
-------+-----------+--------+----------+--------
 59898 | 338.774153 | 138907 | postgres | active
 8698 | 50.358268 | 138953 | postgres | active
 6 | 0.004147 | 137282 | postgres | active
 6 | 0.007396 | 137284 | postgres | active
 6 | 0.022296 | 137283 | postgres | active
 6 | 0.001173 | 138773 | postgres | active
 (6 rows )
```

In the example, the horizon is held for `338` seconds.

7) The values of this test are not interesting. What is interesting is how the wait is inserted in both scripts. When creating your tests, you need to pay attention to the fact that using SQL commands or running commands in psql, you need to check for side effects in the form of holding locks, creating transactions, holding the horizon or snapshot.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

After about 300 seconds of holding the horizon in the window with the pgbench test, a decrease in tps will become noticeable :

```
progress: 3510.0 s, 173.2 tps, lat 23.092 ms stddev 7.375, 0 failed
progress: 3540.0 s, 172.1 tps, lat 23.240 ms stddev 7.978, 0 failed
progress: 3570.0 s, 171.5 tps, lat 23.311 ms stddev 8.286, 0 failed
progress: 3600.0 s, 172.3 tps, lat 23.211 ms stddev 8.417, 0 failed
progress: 3630.0 s, 174.9 tps, lat 22.871 ms stddev 9.124, 0 failed
progress: 3660.0 s, 170.6 tps, lat 23.442 ms stddev 8.418, 0 failed
```

```
60
380 kB
120
282 kB
180
317 kB
360
227 kB
240
210 kB
300
172 kB
420
165 kB
360
152 kB
480
97 kB
420
161 kB
540
 168 kB
480
137 kB
```

8) Stop the tests in both windows.

9) For reference, the result of a long test with horizon retention and a load of one session. In this test, the minimum WAL volume is at checkpoint _ timeout equal to 20 minutes . From this interval, dirty buffers start to be retained in the buffer cache.

```
300
323 kB
0 | 6664 | 0 | 0
1 | 75 | 6 | 0
2 | 34 | 9 | 0
3 | 55 | 19 | 0
4 | 55 | 38 | 0
5 | 9501 | 7878 | 0
600
299 kB
0 | 4263 | 0 | 0
1 | 67 | 0 | 0
2 | 25 | 0 | 0
3 | 36 | 0 | 0
4 | 15 | 0 | 0
5 | 11978 | 9416 | 0
900
291 kB
0 | 683 | 0 | 0
1 | 67 | 0 | 0
2 | 25 | 0 | 0
3 | 36 | 0 | 0
4 | 14 | 0 | 0
 5 | 15559 | 10646 | 0
1200
271 kB
0 | 4081 | 0 | 0
1 | 279 | 118 | 0
2 | 878 | 868 | 0
3 | 962 | 960 | 0
4 | 1205 | 1204 | 0
5 | 8979 | 7697 | 0
1500
275 kB
0 | 2543 | 0 | 0
1 | 1473 | 1438 | 0
2 | 1448 | 1440 | 0
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
3 | 1558 | 1557 | 0
4 | 384 | 379 | 0
5 | 8978 | 7711 | 0
1800
272 kB
0 | 732 | 0 | 0
1 | 1403 | 1243 | 0
2 | 2161 | 2153 | 0
3 | 2160 | 2158 | 0
4 | 260 | 253 | 0
5 | 9668 | 8389 | 0
2100
282 kB
0 | 901 | 0 | 0
1 | 2892 | 2829 | 0
2 | 2488 | 2478 | 0
3 | 293 | 289 | 0
4 | 662 | 655 | 0
5 | 9148 | 7800 | 0
2400
271 kB
0 | 1064 | 0 | 0
1 | 3872 | 3829 | 0
2 | 44 | 34 | 0
3 | 844 | 841 | 0
4 | 1982 | 1938 | 0
5 | 8578 | 7344 | 0
2700
270 kB

0 | 1070 | 0 | 0
1 | 1537 | 42 | 0
2 | 547 | 537 | 0
3 | 1920 | 1919 | 0
4 | 1502 | 1501 | 0
5 | 9808 | 8597 | 0
3000
293 kB

0 | 672 | 0 | 0
1 | 1705 | 1278 | 0
2 | 2758 | 2743 | 0
3 | 1659 | 1600 | 0
4 | 1236 | 1109 | 0
5 | 8354 | 6985 | 0
3300
303 kB
0 | 1034 | 0 | 0
1 | 563 | 530 | 0
2 | 583 | 571 | 0
3 | 920 | 919 | 0
4 | 2389 | 2388 | 0
5 | 10895 | 9551 | 0
3600
302 kB
0 | 2310 | 0 | 0
1 | 492 | 463 | 0
2 | 918 | 904 | 0
3 | 855 | 851 | 0
4 | 2879 | 2874 | 0
5 | 8930 | 7585 | 0
3900
311 kB
0 | 3285 | 0 | 0
1 | 536 | 485 | 0
2 | 937 | 928 | 0
3 | 856 | 855 | 0
4 | 1555 | 1532 | 0
5 | 9215 | 7969 | 0
4200
269 kB
0 | 2195 | 0 | 0
1 | 1104 | 1062 | 0
2 | 1189 | 1178 | 0
3 | 950 | 950 | 0
4 | 1545 | 1543 | 0
5 | 9401 | 8179 | 0
4500
309 kB
0 | 1634 | 4 | 0
1 | 1517 | 1488 | 0
2 | 991 | 982 | 0
3 | 1038 | 1038 | 0
4 | 532 | 531 | 0
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
5 | 10672 | 9324 | 0
4800
313 kB
0 | 1035 | 0 | 0
1 | 1680 | 1616 | 0
2 | 1520 | 181 | 0
3 | 2453 | 2061 | 0
4 | 1035 | 1034 | 0
5 | 8661 | 7432 | 1
```

Part 2. Delay in instance startup. recovery_init_sync_method parameter

To conduct the test, we will use the script from Part 6 of the practice for Chapter 4.

1) Create a file to drop tables `initdrop .sql` :

```
postgres@tantor:~$ mcedit initdrop.sql

\timing on
DO
$$
begin
for i in 1..10000 by 100 loop
  for j in 0..99 loop
execute concat('drop table if exists test.film_summary',i+j);
  end loop;
commit;
end loop;
execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;
```

1) Create or check content file `init.sql` :

```
postgres@tantor:~$ mcedit init.sql

create schema test;
select format('create table test.film_summary%s (film_id int, title varchar,
release_year smallint) with (autovacuum_enabled=off);', g.id)
from generate_series(1, 10000) as g(id)
\ gexec
```

2) Run the commands:

```
postgres @ tantor :~$
time psql -f initdrop.sql 2> /dev/null
psql -c "checkpoint;"
time psql -f init.sql > /dev/null
pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
Timing is on.
DO
Time: 5920.933 ms (00:05.921)

real 0m5.938s
user 0m0.009s
sys 0m0.000s
CHECKPOINT
psql:init.sql:1: NOTICE: schema "test" does not exist, skipping
```

```
real 2m38.121s
user 0m0.337s
sys 0m0.185s
waiting for server to shut down.... done
server stopped

real 2m36 .460s
user 0m0.016s
sys 0m0.000s
```

The teams performed a checkpoint so that the time in the log did not include the time of the extended checkpoint.

For **2 m 38** .121 s 10000 tables, 10000 TOAST tables, 10000 indexes on TOAST tables were created.

was stopped in **immediate mode** (simulating an instance failure).

For **2 m 36** .460 s the instance was launched.

The list of processes while the instance was running was as follows:

```
postgres@tantor:~$ ps -ef | grep postgres
postgres 62347 7689 0 00:08 pts/0 00:00:00 pg_ctl start
postgres 62349 62347 0 00:08 ? 00:00:00 /opt/tantor/db/16/bin/ postgres
postgres 62350 62349 0 00:08 ? 00:00:00 postgres: logger
postgres 62351 62349 0 00:08 ? 00:00:00 postgres: checkpointer
postgres 62352 62349 0 00:08 ? 00:00:00 postgres: background writer
postgres 62353 62349 4 00:08 ? 00:00:01 postgres: startup
postgres 62361 62319 99 00:08 pts/2 00:00:00 ps -ef
postgres 62362 62319 0 00:08 pts/2 00:00:00 grep postgres
```

3) You will not be able to connect to the instance until the startup process has completed and until a post-recovery checkpoint has been performed:

```
postgres@tantor:~$ psql
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: the database system is not yet accepting connections
DETAIL: Consistent recovery state has not yet been reached.
```

4) After launching, look at the contents of the diagnostic log:

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*
14:13:05.688 MSK [83807] LOG: starting Tantor Special Edition 16.6.1 a74db619 on x86_64-pc-linux-gnu, compiled
by gcc (Astra 12.2.0-14.astra3+b1) 12.2.0, 64-bit
14:13:05.689 MSK [83807] LOG: listening on IPv4 address "127.0.0.1", port 5432
14:13:05.689 MSK [83807] LOG: listening on IPv6 address "::1", port 5432
14:13:05.701 MSK [83807] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
14:13:05 .736 MSK [83812] LOG: database system was interrupted; last known up at 14:10:26 MSK
14:13:16.095 MSK [83812] LOG: syncing data directory (fsync), elapsed time: 10.00 s, current path:
./base/5/1057586
14:13:26.095 MSK [83812] LOG: syncing data directory (fsync), elapsed time: 20.00 s, current path:
./base/5/1074286
14:13:36.096 MSK [83812] LOG: syncing data directory (fsync), elapsed time: 30.00 s, current path:
./base/5/1077650
14:13:46.097 MSK [83812] LOG: syncing data directory (fsync), elapsed time: 40.00 s, current path:
./base/5/1094421
14:13:56.095 MSK [83812] LOG: syncing data directory (fsync), elapsed time: 50.00 s, current path:
./base/5/1065705
14:14:06.096 MSK [83812] LOG: syncing data directory (fsync), elapsed time: 60.00 s, current path:
./base/5/1061386
14:14:16.096 MSK [83812] LOG: syncing data directory (fsync), elapsed time: 70.00 s, current path:
./base/5/1093966
14:14:21 .053 MSK [83812] LOG: database system was not properly shut down; automatic recovery in progress
14:14:21 .078 MSK [83812] LOG: redo starts at 3D/2FB2B608
14:14:22.988 MSK [83812] LOG: invalid record length at 3D/3ABE40D8: expected at least 26, got 0
14:14:22.988 MSK [83812] LOG: redo done at 3D/3ABE3C50 system usage: CPU: user: 1.89 s, system: 0.00 s, elapsed:
1.91 s
14:14:23.094 MSK [83810] LOG: checkpoint starting: end-of-recovery immediate wait
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
14:15:41 .933 MSK [83810] LOG: checkpoint complete : wrote 15858 buffers (96.8%); 0 WAL file(s) added, 11 removed, 0 recycled; write=1.483 s , sync=77.183 s , total=78.851 s ; sync files=30045 , longest=0.049 s, average=0.003 s; distance=180962 kB , estimate=180962 kB; lsn=3D/3ABE40D8, redo lsn=3D/3ABE40D8
14:15:41.960 MSK [83807] LOG: database system is ready to accept connections
```

The status of the startup process is written to the log file at a default interval of once every 10 seconds , which is set by the `log_startup_progress_interval` configuration parameter .

File synchronization time `14:14:21` - `14:13:05` = 76 seconds .
Recovery time (log rewind) `14:15:41` - `14:14:21` = 80 seconds .

5) See what happens if you perform a checkpoint before stopping the instance with **the checkpoint command** :

```
postgres@tantor:~$
time psql -f initdrop.sql 2> /dev/null
psql -c "checkpoint;"
time psql -f init.sql > /dev/null
time psql -c "checkpoint;"
pg_ctl stop -m immediate
time sudo systemctl start tantor-se-server-16
cat $PGDATA/log/postgresql-*
Timing is on.
DO
Time: 5417.500 ms (00:05.418)

real 0m5.454s
user 0m0.014s
sys 0m0.007s
CHECKPOINT
psql:init.sql:1: NOTICE: schema "test" does not exist, skipping

real 2m21 .283s
user 0m0.340s
sys 0m0.173s
CHECKPOINT

real 1m13 .241s
user 0m0.004s
sys 0m0.005s
waiting for server to shut down.... done
server stopped

real 1m15 .575s
user 0m0.012s
sys 0m0.004s
```

When performing **a checkpoint** before stopping the instance, the checkpoint and startup took **1 m 13** .241 s + **1 m 15** .575 s = 2m28.186 sec . The idle time was **1 m 15** .575 s .
Without a checkpoint before stopping the instance, starting the instance took **2 m 36** .460 s . Downtime was **2 m 36** .460 s . Recovery took slightly longer, but was comparable in speed to the checkpoint.

6) Example of the contents of the journal:

```
postgres @ tantor :~$ cat $ PGDATA / log / postgresql -*

14:15:41.960 MSK [83807] LOG: database system is ready to accept connections
18:32:27.928 MSK [83810] LOG: checkpoint starting: immediate force wait
```

```
   18:32:28.741 MSK [83810] LOG: checkpoint complete: wrote 5647 buffers (34.5%); 0 WAL file(s) added, 1
removed, 4 recycled; write=0.189 s, sync=0.066 s, total=0.813 s; sync files=15, longest=0.033 s, average=0.005
s; distance=77253 kB, estimate=170591 kB; lsn=3D/3F755900, redo lsn=3D/3F7558B8
   18:34:50 .042 MSK [83810] LOG: checkpoint starting: immediate force wait
   18:36:03.261 MSK [83810] LOG: checkpoint complete: wrote 11939 buffers (72.9%); 0 WAL file(s) added, 0
removed, 11 recycled; write=0.749 s , sync=72.259 s , total=73.219 s ; sync files=30042 ,
longest=0.034 s, average=0.003 s; distance=179845 kB , estimate=179845 kB; lsn=3D/4B16AEE8, redo
lsn=3D/4A6F6FF8
   18:36:03.268 MSK [83807] LOG: received immediate shutdown request
   18:36:03.301 MSK [83807] LOG: database system is shut down
   18:36:03.531 MSK [89269] LOG: starting Tantor Special Edition 16.6.1 a74db619 on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3+b1) 12.2.0, 64-bit
   18:36:03.532 MSK [89269] LOG: listening on IPv4 address "127.0.0.1", port 5432
   18:36:03.532 MSK [89269] LOG: listening on IPv6 address "::1", port 5432
   18:36:03.540 MSK [89269] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
   18:36:03 .570 MSK [89273] LOG: database system was interrupted; last known up at 18:36:03 MSK
   18:36:13.926 MSK [89273] LOG: syncing data directory (fsync), elapsed time: 10.00 s, current path:
./base/5/1108344
   18:36:23.925 MSK [89273] LOG: syncing data directory (fsync), elapsed time: 20.00 s, current path:
./base/5/1111670
   18:36:33.924 MSK [89273] LOG: syncing data directory (fsync), elapsed time: 30.00 s, current path:
./base/5/1125680
   18:36:43.926 MSK [89273] LOG: syncing data directory (fsync), elapsed time: 40.00 s, current path:
./base/5/1142589
   18:36:53.926 MSK [89273] LOG: syncing data directory (fsync), elapsed time: 50.00 s, current path:
./base/5/1127261
   18:37:03.925 MSK [89273] LOG: syncing data directory (fsync), elapsed time: 60.00 s, current path:
./base/5/1113186
   18:37:13.924 MSK [89273] LOG: syncing data directory (fsync), elapsed time: 70.00 s, current path:
./base/5/1144321
   18:37:18 .586 MSK [89273] LOG: database system was not properly shut down; automatic recovery in progress
   18:37:18 .604 MSK [89273] LOG: redo starts at 3D/4A6F6FF8
   18:37:18.628 MSK [89273] LOG: invalid record length at 3D/4B16AF80: expected at least 26, got 0
   18:37:18.628 MSK [89273] LOG: redo done at 3D/4B16AEE8 system usage: CPU: user: 0.02 s, system: 0.00 s,
elapsed: 0.02 s
   18:37:18.666 MSK [89271] LOG: checkpoint starting: end-of-recovery immediate wait
   18:37:18 .877 MSK [89271] LOG: checkpoint complete: wrote 1321 buffers (8.1%); 0 WAL file(s) added, 1
removed, 0 recycled; write=0.078 s , sync=0.078 s , total=0.219 s ; sync files=13 ,
longest=0.044 s, average=0.00 d 6 s; distance=10703 kB , estimate=10703 kB; lsn=3D/4B16AF80, redo
lsn=3D/4B16AF80
   18:37:18.891 MSK [89269] LOG: database system is ready to accept connections
```

Volume of log data: **distance = 179845 kB + 10703 kB** = 190548. Before: **distance = 180962 kB** . A small increase in time corresponds to a small increase in the volume of log records.

7) Perform request :

```
postgres=# select * from pg_stat_recovery_prefetch\gx
-[ RECORD 1 ]--+-----------------------------
stats_reset | 2035-01-01 00:08:04.65834+03
prefetch | 10556
hit | 435863
skip_init | 3986
skip_new | 2
skip_fpw | 15327
skip_rep | 432413
wal_distance | 0
block_distance | 0
io_depth | 0
```

The view is populated by the startup process. If there was no recovery when the instance was started, the values are zero.

The view only shows log blocks. Before the instance was stopped, 10,000 tables were created and many log records were generated.

Descriptions of the view columns are in the theoretical part of Chapter 10.

8) On the replica, the statistics are constantly increasing and on the replicas, the values are reset by calling the function `select pg_stat_reset_shared('recovery_prefetch')` . Execute reset statistics :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres=# select pg_stat_reset_shared('recovery_prefetch');
select * from pg_stat_recovery_prefetch\gx
pg_stat_reset_shared
----------------------

(1 row)

-[ RECORD 1 ]--+------------------------------
stats_reset | 2035-01-11 00:09:02.679457+03
prefetch | 0
hit | 0
skip_init | 0
skip_new | 0
skip_fpw | 0
skip_rep | 0
wal_distance | 0
block_distance | 0
io_depth | 0
```

Statistics have been reset.

9) You can force stop the instance again and start it again:

```
postgres@tantor:~$ pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
cat $PGDATA/log/postgresql-*
waiting for server to shut down.... done
server stopped

real 1m13 .828s
user 0m0.015s
sys 0m0.000s
03:20:19.094 MSK [14260] LOG: starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
03:20:19.094 MSK [14260] LOG: listening on IPv4 address "127.0.0.1", port 5432
03:20:19.094 MSK [14260] LOG: listening on IPv6 address "::1", port 5432
03:20:19.105 MSK [14260] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
03:20:19 .136 MSK [14265] LOG: database system was interrupted; last known up at 03:17:41 MSK
03:20:29.582 MSK [14265] LOG: syncing data directory (fsync), elapsed time: 10.00 s, current path:
./base/5/4066825
03:20:39.584 MSK [14265] LOG: syncing data directory (fsync), elapsed time: 20.00 s, current path:
./base/5/1903193
03:20:49.583 MSK [14265] LOG: syncing data directory (fsync), elapsed time: 30.00 s, current path:
./base/5/4075920
03:20:59.584 MSK [14265] LOG: syncing data directory (fsync), elapsed time: 40.00 s, current path:
./base/5/4059545
03:21:09.583 MSK [14265] LOG: syncing data directory (fsync), elapsed time: 50.00 s, current path:
./base/5/4048445
03:21:19.582 MSK [14265] LOG: syncing data directory (fsync), elapsed time: 60.00 s, current path:
./base/5/4079304
03:21:29.583 MSK [14265] LOG: syncing data directory (fsync), elapsed time: 70.00 s, current path:
./base/5/4081550
03:21:32 .276 MSK [14265] LOG: database system was not properly shut down; automatic recovery in progress
03:21:32 .298 MSK [14265] LOG: redo starts at 37/CC2D07A8
03:21:32.298 MSK [14265] LOG: invalid record length at 37/CC2D08D0: expected at least 26, got 0
03:21:32.298 MSK [14265] LOG: redo done at 37/CC2D0888 system usage: CPU: user: 0.00 s, system: 0.00 s,
elapsed: 0.00 s
03:21:32.347 MSK [14263] LOG: checkpoint starting: end-of-recovery immediate wait
03:21:32 .418 MSK [14263] LOG: checkpoint complete: wrote 4 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0
recycled; write=0.024 s, sync=0.011 s, total=0.079 s; sync files=3, longest=0.005 s, average=0.004 s; distance=0
kB, estimate=0 kB; lsn=37/CC2D08D0, redo lsn=37/CC2D08D0
03:21:32.431 MSK [14260] LOG: database system is ready to accept connections
```

Due to the presence of 10,000 tables in one of the cluster databases, after stopping an instance in immediate mode, due to lack of memory or power loss, the instance will take 73 seconds to start up, plus a roll forward of log files and a checkpoint after the roll forward.

Since the previous instance opening had a recovery (the WAL log was rolled forward), a post-recovery checkpoint was performed, and there were no changes after that, then when the instance was started, the recovery and checkpoint were performed in a split second 03 :21: 32.418 - 03:21: 32.298 .

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group; Tantor Labs LLC © 2025 www.tantorlabs.ru

The duration of the instance startup synchronization phase depends on the number of files in the cluster's PGDATA .

10) Execute the request:

```
postgres=# select * from pg_stat_recovery_prefetch\gx
-[ RECORD 1 ]--+-----------------------------
stats_reset | 2035-01-11 00:16:37.654068+03
prefetch | 0
hit | 0
skip_init | 0
skip_new | 0
skip_fpw | 720
skip_rep | 0
wal_distance | 0
block_distance | 0
io_depth | 0
```

During recovery, the startup process did not roll over a single log block.

How to speed up instance startup?

11) Run the command:

```
postgres=# alter system set recovery_init_sync_method = syncfs;
ALTER SYSTEM
postgres=# \q
```

12) Repeat restart instance :

```
postgres@tantor:~$ pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
cat $PGDATA/log/postgresql-*
waiting for server to shut down.... done
server stopped

real 0m0.511s
user 0m0.015s
sys 0m0.000s
03:27:10.679 MSK [14302] LOG: starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
03:27:10.680 MSK [14302] LOG: listening on IPv4 address "127.0.0.1", port 5432
03:27:10.680 MSK [14302] LOG: listening on IPv6 address "::1", port 5432
03:27:10.688 MSK [14302] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
03:27:10.717 MSK [14306] LOG: database system was interrupted; last known up at 03:21:32 MSK
03:27:10.787 MSK [14306] LOG: database system was not properly shut down; automatic recovery in progress
03:27:10.808 MSK [14306] LOG: redo starts at 37/CC2D0968
03:27:10.808 MSK [14306] LOG: invalid record length at 37/CC2D09B0: expected at least 26, got 0
03:27:10.808 MSK [14306] LOG: redo done at 37/CC2D0968 system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
03:27:10.851 MSK [14304] LOG: checkpoint starting: end-of-recovery immediate wait
03:27:10.921 MSK [14304] LOG: checkpoint complete: wrote 4 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.023 s, sync=0.012 s, total=0.079 s; sync files=3, longest=0.005 s, average=0.004 s; distance=0 kB, estimate=0 kB; lsn=37/CC2D09B0, redo lsn=37/CC2D09B0
03:27:10.932 MSK [14302] LOG: database system is ready to accept connections
```

The instance launched in a split second.
Using **recovery_init_sync_method = syncfs does not impair fault** tolerance **.**

By default, the duration of keeping pages in the Linux page cache is determined by the Linux parameter : **vm.dirty_expire_centisecs** - how long the buffer can be dirty before it is marked for writing, by default 3000 (30 seconds). Under heavy disk load, more than 30 seconds can pass from marking a block to writing. The Linux page cache settings were discussed in Chapter 2 (2-3) on the slide "Duration of keeping dirty pages in the cache". In the example, there was no load on the virtual

machine disk system, and file-by-file synchronization took 73 seconds. In any case, and even if the disk is loaded with other applications in addition to the instance, full synchronization of the file system will be performed faster, so it is worth using **recovery _ init _ sync _ method = syncfs** . This value is not used by default, probably due to concerns about linux bugs , as stated in the PostgreSQL documentation for the recovery_init_sync_method configuration parameter **: " Linux versions prior to 5.8 may not always report I / O errors to the instance when writing data to disk, and** messages may be visible only in the kernel log." The kernel log can be viewed using the dmesg operating system command with root privileges.

You can estimate the synchronization time of file systems using the operating system command:

```
postgres@tantor:~$ time sync -f

real 0m0.010s
user 0m0.002s
sys 0m0.000s
```

From Linux documentation :

```
syncfs() is like sync() , but synchronizes just the filesystem containing file referred to by the
open file descriptor fd. sync() is always successful.
syncfs() can fail for at least the following reasons:
EBADF fd is not a valid file descriptor.
EIO An error occurred during synchronization. This error may relate to data written to any file
on the filesystem, or on meta - data related to the filesystem itself.
ENOSPC Disk space was exhausted while synchronizing.
```

13) The backup utility has a parameter -- no - sync . Check the backup time without the parameter and with the parameter:

```
postgres@tantor:~$ rm -rf /var/lib/postgresql/backup/1
time pg_basebackup -c fast -D $HOME/backup/1 -P
rm -rf /var/lib/postgresql/backup/1
time pg_basebackup --no-sync -c fast -D $HOME/backup/1 -P
timesync -f
3783151/3783151 kB (100%), 1/1 tablespace

real 2m50.010s
user 0m52.035s
sys 0m3.515s

real 0m40.807s
user 0m36.967s
sys 0m2.612s

real 0m0.007s
user 0m0.001s
sys 0m0.000s
```

The backup time without the parameter is significantly longer than with the parameter and the `sync -f command` . In version 17, the --sync-method=syncfs parameter appeared.

Parameter **- c fast** allows you not to wait for the checkpoint to be performed, it is valid when connecting to the master. The parameter is used to start the backup faster. If the start time of copying is not critical, it is better to wait for the checkpoint to be performed by time.

14) Try deleting the 10,000 tables created:

```
postgres =# \ timing on \\
drop schema test cascade;
ERROR: out of shared memory
HINT: You might need to increase max_locks_per_transaction.
Time : 108.804 ms
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Either you will have to write a script to delete tables one by one, or increase the number of locks.

15) Create a file to drop tables if it has not already been created initdrop .sql :

```
postgres@tantor:~$ cat initdrop.sql

\timing on
DO
$$
begin
for i in 1..10000 by 100 loop
  for j in 0..99 loop
execute concat('drop table if exists test.film_summary',i+j);
  end loop;
commit;
end loop;
execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;
```

16) Run the script:

```
postgres@tantor:~$ time psql -f initdrop.sql 2> /dev/null

real 0m5.678s
user 0m0.009s
sys 0 m 0.000 s
```

The tables have been removed for 5,678 seconds .

Part 3. Checkpoint duration

`init1.sql` file :

```
postgres@tantor:~$ mcedit init1.sql

create schema test;
select format('create table test.film_summary%s (film_id int, title varchar,
release_year smallint) with (autovacuum_enabled=off);', g.id) from
generate_series(1, 10000) as g(id)
\gexec
select format('insert into test.film_summary%1$s select i, %2$s || i, i from
generate_series(0, 300) as i;', g.id, E'\'text number \'') from
generate_series(1, 10000) as g(id)
\gexec
select format('select * from test.film_summary%s where film_id = 0;', g.id)
from generate_series(1, 10000) as g(id)
 \ gexec
```

The script will create 10,000 tables in the test scheme with 300 rows in each table in two blocks and will reread the buffers so that they do not get dirty when reading and this does not create errors in the test results.

2) Install parameters configurations :

```
postgres=# alter system set checkpoint_timeout = 1200;
alter system set recovery_init_sync_method = syncfs;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
alter system set shared_buffers = '512MB';
alter system set max_connections = 1000;
alter system set log_min_duration_statement = '15s';
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
postgres=# \q
```

The checkpoint interval is longer so that the checkpoints have less influence on the results and the interval is closer to the values used in practice. 5 minutes are not used in practice.

If you edit the numbers in the test, you need to make sure that the tests do not generate a log volume larger than max_wal_size (1GB by default).

recovery_init_sync_method = syncfs to avoid waiting an extra ~ 80 seconds if the instance is stopped in immediate mode. These ~80 seconds are not related to checkpoints and log rollbacks and depend only on the number of files in **PGDATA .**

The buffer pool size has been increased because dirty buffers are expected to be greater than 128MB. In real use, there are almost no free buffers in the buffer cache, all buffers are occupied by object blocks.

The number of connections has been increased to avoid hitting the lock limit and to bring it closer to the values used in practice.

Log long commands in case such commands are executed so that attention can be drawn to them.

3) Check that the contents of the `postgresql.auto.conf parameter file` are as follows:

```
postgres@tantor:~$ cat $PGDATA/postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
logging_collector = 'on'
checkpoint_timeout = '1200'
recovery_init_sync_method = 'syncfs'
shared_buffers = '1GB'
max_connections = '1000'
log_min_duration_statement = '15s'
```

4) Delete the log to avoid searching through the set of logs for the latest one, restart the instance to apply the configuration parameter values and clear the buffer cache, run the test script:

```
postgres@tantor:~$ rm -f $PGDATA/log/*
psql -c "vacuum full;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null

VACUUM
real 3m34 .352s
user 0m1.554s
sys 0 m 0.498 s
```

The test script is convenient because it creates many dirty buffers in 10,000 tables. In real work, many blocks are also updated in a large number of tables and indexes.

Using a single table in a test, even a large one, would be far from real work.

5) Do it request :

```
postgres@tantor:~$ psql -c "select * buffercache_usage_counts();"
usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
0 | 4759 | 0 | 0
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
1 | 20126 | 10002 | 0
2 | 10027 | 1 | 0
3 | 12 | 2 | 0
4 | 20 | 2 | 0
5 | 30592 | 28466 | 0
(6 rows)
```

Big Part buffers from located V buffer cache (~50 thousand) dirty (38 thousand).
4762 buffers free.

5) Perform a checkpoint:

```
postgres@tantor:~$ psql -c "alter system set log_statement = ddl;"
psql -c "select pg_reload_conf();"
time psql -c "checkpoint;"
psql -c "alter system reset log_statement"
psql -c "select pg_reload_conf();"
cat $PGDATA/log/postgresql-*
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row)

CHECKPOINT

real 1m23 .284 s
user 0m0.010s
sys 0m0.000s
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row)

   03:19:20.674 MSK [99626] LOG: starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu,
compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
   03:19:20.674 MSK [99626] LOG: listening on IPv4 address "127.0.0.1", port 5432
   03:19:20.674 MSK [99626] LOG: listening on IPv6 address "::1", port 5432
   03:19:20.683 MSK [99626] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
   03:19:20.705 MSK [99631] LOG: database system was shut down at 03:19:19 MSK
   03:19:20.719 MSK [99626] LOG: database system is ready to accept connections
   03:25:00.845 MSK [99626] LOG: received SIGHUP, reloading configuration files
   03:25:00.846 MSK [99626] LOG: parameter "log_statement" changed to "ddl"
   03:25:00.866 MSK [99629] LOG: checkpoint starting: immediate force wait
   03:26:24.124 MSK [99629] LOG: checkpoint complete: wrote 38498 buffers (58.7%); 0 WAL file(s) added,
0 removed, 27 recycled; write=2.811 s , sync=80.146 s , total=83.259 s ; sync files=40047 ,
longest=0.041 s, average=0.002 s; distance=441826 kB , estimate=441826 kB; lsn=3A/E67EE0D0, redo
lsn=3A/E67EE088
   03:26:24.135 MSK [99661] LOG: duration: 83,269.646 ms statement: checkpoint ;
   03:26:24.149 MSK [99672] LOG: statement: alter system reset log_statement
   03:26:24.184 MSK [99626] LOG: received SIGHUP, reloading configuration files
   03:26:24.185 MSK [99626] LOG: parameter "log_statement" removed from configuration file, reset to default
```

The checkpoint took 83 seconds to complete .

At `03:25:00` the checkpoint began on command (`immediate force wait`).

log _ statement = ddl the only thing it did was display the line `duration : 83269.646 ms statement : checkpoint ;` at the end of the checkpoint. The moment of issuing the command this configuration parameter did not output. `duration : 83269.646 ms` is the duration of the maximum speed checkpoint from the checkpoint command until the checkpoint completes. If an extended checkpoint is running, then `total = 83.259 s` gives the duration of the extended checkpoint and is not the same as `duration` . In this example, there was no extended checkpoint and the `duration value` is similar to `total` .

`distance =441826 kB` = 55228 blocks were created .

`28466+10007=38453` dirty blocks . This number of blocks was written in full page writes mode. Since the table blocks were almost completely filled, they took up the same amount of space in the WAL.

6) Check that there are no dirty buffers :

```
postgres@tantor:~$ psql -c "select * buffercache_usage_counts();"
usage_count | buffers | dirty | pinned
-------------+---------+-------+---------
           0 | 4748 | 0 | 0
1 | 20128 | 0 | 0
2 | 10027 | 0 | 0
3 | 11 | 0 | 0
4 | 20 | 0 | 0
5 | 30602 | 0 | 0
(6 rows )
```

All dirty buffers were written to disk via checkpoint.

7) In table blocks there are 2 blocks, in these blocks there are 40 and 612 bytes free:

```
postgres@tantor:~$ psql -c "select upper-lower free from
page_header(get_raw_page('test.film_summary1','main',0));"
psql -c "select upper-lower free from
page_header(get_raw_page('test.film_summary1','main',1));"
free
------
40
(1 row )

 free
------
612
(1 row )
```

When issuing the CHECKPOINT command after the instance has worked checkpoint_timeout time, in order to read the log you need to know the following.

If a timed checkpoint were to be started, the CHECKPOINT command accelerates the timed checkpoint, which starts running at maximum speed. Then another checkpoint is started (by the CHECKPOINT command), and it runs quickly.

That is, the statistics of the execution of a checkpoint using the checkpoint command should be viewed in the message related to the previous checkpoint - by time.

8) Restarting the instance will be instantaneous, since there are no dirty buffers:

```
postgres@tantor:~$ time sudo systemctl restart tantor-se-server-16

real 0m0.515s
user 0m0.015s
sys 0 m 0.000 s
```

Therefore, if you need to stop an instance, it is recommended to perform a checkpoint with the CHECKPOINT command . The benefit is that while the command is running, the instance is available and servicing clients. If you give the instance a stop command, the instance is no longer available to clients and service downtime begins.

9) Drop tables in one transaction:

```
postgres @ tantor :~$ psql

postgres=# \timing on \\
DO
$$
```

```
begin
for i in 1..10000 loop
execute concat('drop table if exists test.film_summary',i);
end loop;
execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 6399.334 ms (00:06.399)
```

Tables were deleted 6399.334 ms .

Part 4. Duration of the final checkpoint when stopping the instance

Let's check that the duration of the final checkpoint when setting an instance is exactly the same as for the CHECKPOINT command.

1) Repeat the creation of tables:

```
postgres @ tantor :~$ rm - f $ PGDATA / log /*
psql -c "vacuum full;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
time sudo systemctl stop tantor-se-server-16
cat $PGDATA/log/postgresql-*

VACUUM
real 3m32 .800s
user 0m1.545s
sys 0m0.553s

real 1m47.895s
user 0m0.011s
sys 0m0.004s

04:55:01.531 MSK [100981] LOG: starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
04:55:01.531 MSK [100981] LOG: listening on IPv4 address "127.0.0.1", port 5432
04:55:01.531 MSK [100981] LOG: listening on IPv6 address "::1", port 5432
04:55:01.540 MSK [100981] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
04:55:01.561 MSK [100985] LOG: database system was shut down at 04:55:00 MSK
04:55:01.575 MSK [100981] LOG: database system is ready to accept connections
05:01:37.224 MSK [100981] LOG: received fast shutdown request
05:01:37.237 MSK [100981] LOG: aborting any active transactions
05:01:37.241 MSK [100981] LOG: background worker "logical replication launcher" (PID 100988) exited with exit code 1
05:01:37.245 MSK [100983] LOG: shutting down
05:01:37 .252 MSK [100983] LOG: checkpoint starting: shutdown immediate
05:03:23 .824 MSK [100983] LOG: checkpoint complete: wrote 38495 buffers (58.7%); 0 WAL file(s) added, 0 removed, 27 recycled; write=3.671 s , sync=102.522 s , total=106.580 s ; sync files=40047 , longest=0.145 s, average=0.003 s; distance=441785 kB , estimate=441785 kB; lsn=3B/28562C40, redo lsn=3B/28562C40
05:03:23.912 MSK [100981] LOG: database system is shut down
```

Checkpoint duration: 05:03:23 - 05:01:37 = 106 seconds .
Stop with pg_ctl command stop or systemctl stop does not affect the duration of the stop, since systemctl stop calls pg_ctl stop .
distance =441785 kB and number of files = 40047 are the same as when executing the CHECKPOINT command.

`write =3.671 s` a little more than it was: `write =2.811 s .`

Test result: The final checkpoint is not faster than the manual checkpoint. In the example, the final checkpoint took 23 seconds longer to complete: 106 seconds instead of 83 seconds.

3) Launch instance :

```
postgres@tantor:~$ sudo systemctl start tantor-se-server-16
```

4) Delete tables 100 at a time in a transaction:

```
postgres=# \timing on \\
DO
$$
begin
for i in 1..10000 by 100 loop
for j in 0..99 loop
execute concat('drop table if exists test.film_summary',i+j);
end loop;
commit;
end loop;
execute 'drop schema if exists test cascade';
end;
$$
LANGUAGE plpgsql;

DO
Time: 6979.210 ms (00:06.979)
```

The deletion took `6979 ms` .

Last time we deleted in one transaction and it took `6399.334 ms` . Now when deleting 100 tables in 100 transactions, the deletion took `6979 ms` . Deleting in one transaction is more efficient, but requires more locks.

Part 5. Duration of checkpoint after instance crash

In this part of the practice, we will perform the same sequence of actions as in the previous part, only without a checkpoint, with the instance stopped in immediate mode (without a checkpoint).
Objective: To determine how long it will take for the startup process to restore WAL blocks when the instance is started and whether the checkpoint duration will change.

1) Repeat the creation of tables:

```
postgres @ tantor :~$ rm - f $ PGDATA / log /*
psql -c "vacuum full;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
cat $PGDATA/log/postgresql-* | egrep "checkpoint|ready"

VACUUM
real 3m36 .456s
user 0m1.498s
sys 0m0.562s
03:59:21.293 MSK [99814] LOG: database system is ready to accept connections
```

The tables took the same amount of time to create as last time.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

2) Stop the instance without checkpoint and start the instance:

```
postgres@tantor:~$ pg_ctl stop -m immediate
rm -f $PGDATA/log/*
time sudo systemctl start tantor-se-server-16
tail -n 7 $PGDATA/log/postgresql-*
```

```
waiting for server to shut down.... done
server stopped

real 1m34.397s
user 0m0.015s
sys 0m0.000s
04:07:27.828 MSK [100872] LOG: database system was not properly shut down; automatic recovery in progress
04:07:27.856 MSK [100872] LOG: redo starts at 3A/EC71EC38
04:07:37.561 MSK [100872] LOG: invalid record length at 3B/769AFC8: expected at least 26, got 0
04:07:37.561 MSK [100872] LOG: redo done at 3B/769AF80 system usage: CPU: user: 5.54 s, system: 1.27 s, elapsed: 9.70 s
04:07:37.618 MSK [100870] LOG: checkpoint starting: end-of-recovery immediate wait
04:09:01.824 MSK [100870] LOG: checkpoint complete: wrote 48483 buffers (74.0%); 0 WAL file(s) added, 0 removed, 27 recycled; write=3.816 s , sync=80.096 s , total=84.213 s ; sync files=40049 , longest=0.087 s, average=0.002 s; distance=441841 kB , estimate=441841 kB; lsn=3B/769AFC8, redo lsn=3B/769AFC8
04:09:01.860 MSK [100868] LOG: database system is ready to accept connections
```

The instance took 94 seconds to run .

First there was a rollover of log data, it lasted 10 seconds from `04:07:27` until `04:07:37` .

Next, a post-recovery checkpoint was performed, which lasted `total =84.213 s` . Transferring blocks from the buffer cache to the Linux page cache took only `write =3.816 s . Distance =441841 kB` was written to the WAL files .

The f sync call was sent on `40049` files and lasted `sync =80.096 s` .

For the most part, this is the same 73 seconds that fsync performed on individual files in Part 2 of the practice. If a syncfs or sync call were sent over the file system, it would last no more than 80-73=7 seconds.

For on-demand checkpoints, it would be better for PostgreSQL code to perform syncfs on file systems with PGDATA, instead of fsync on individual files, at least at a checkpoint when stopping an instance. The `recovery_init_sync_method parameter` was introduced in PostgreSQL version 14 and currently has the default value fsync. The `-- sync - method = syncfs parameter` for the `pg_basebackup utility was introduced in version 17. That is, support for full synchronization of syncfs` file systems is in development.

You can compare it with the log from point 5 of this part of the practice, the values are almost the same.

Test result: Restarting the instance took 10 seconds longer than if the instance had been stopped gracefully. The log rollover took 10 seconds.

3) Let's check how the `drop command is executed schema test cascade` . Let's make sure that this command does not inflate the transaction and multitransaction counter. Run commands :

```
postgres@tantor:~$ psql -c "SELECT datname, age(datfrozenxid), mxid_age(datminmxid) FROM pg_database where datname='postgres';"
psql -c "select pg_current_xact_id();"
time psql -c " drop schema test cascade; "
psql -c "SELECT datname, age(datfrozenxid), mxid_age(datminmxid) FROM pg_database where datname='postgres';"
select pg_current_xact_id();"
datname | age | mxid_age
----------+--------+----------
postgres | 594926 | 0
(1 row)
```

```
pg_current_xact_id
--------------------
             6949699
(1 row)

NOTICE: drop cascades to 10000 other objects
DETAIL: drop cascades to table test.film_summary1
drop cascades to table test.film_summary2
...
drop cascades to table test.film_summary100
and 9900 other objects (see server log for list)
DROP SCHEMA

real 0m 7.658s
user 0m0.011s
sys 0m0.000s
datname | age | mxid_age
----------+--------+----------
postgres | 594928 | 0
(1 row)

pg_current_xact_id
--------------------
             6949706
(1 row)
```

Team **drop schema test cascade** Not they are winding up counters transactions .

The xid value has a difference of **6949706 - 6949699 =** 5 transactions (6th transaction was created by the function call). The drop schema command used less than 5 transactions or subtransactions (savepoint).

Test result: The drop schema cascade command executed slower (7.658 s) than dropping a plpgsql block in 100 transactions with 100 tables in each transaction (6979 ms) and slower than dropping a single transaction (6399.334 ms).

The advantage of deleting in 100 transactions is that each transaction lasts 6979 ms / 100 = 69 ms and the database horizon retention is 100 times shorter than other table deletion methods.

Part 6. Checkpoint on request

I would like to see how to read the log if a time checkpoint has started and a checkpoint has been issued using the checkpoint command.

Also check if **max _ connections affects** the speed of checkpoint execution and creation, deletion of objects. Therefore, we will reduce it to the default value of 100.

To avoid waiting too long, let's set **checkpoint _ timeout** the default value is 5 minutes.

1) Execute the commands (execution time is approximately 7 minutes):

```
postgres @ tantor :~$
psql - c " alter system reset checkpoint_timeout ; "
psql -c "alter system reset max_connections;"
psql -c " vacuum full;"
rm -f $PGDATA/log/*
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
psql -c " select backend_type name, sum(writes) buffers_written, sum(writebacks) writebacks, sum(evictions) evictions, sum(fsyncs) fsyncs, sum(fsync_time) fsync_time from pg_stat_io group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or sum(fsyncs)>0 or sum(evictions)>0; "
sleep 100
psql -c "select * buffercache_usage_counts();"
sleep 10
psql -c "select * buffercache_usage_counts();"
time psql -c "checkpoint;"
tail -6 $PGDATA/log/postgresql-*
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
psql -c " select backend_type name, sum(writes) buffers_written, sum(writebacks) writebacks, sum(evictions) evictions, sum(fsyncs) fsyncs, sum(fsync_time) fsync_time from pg_stat_io group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or sum(fsyncs)>0 or sum(evictions)>0; "

VACUUM
ALTER SYSTEM
ALTER SYSTEM

real 3m31 .394s
user 0m1.400s
sys 0m0.678s

usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
0 | 4765 | 0 | 0
1 | 20125 | 10001 | 0
2 | 10024 | 0 | 0
3 | 10 | 0 | 0
4 | 19 | 0 | 0
5 | 30593 | 26780 | 0
(6 rows)

usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
0 | 4765 | 0 | 0
1 | 20119 | 10000 | 0
2 | 10026 | 0 | 0
3 | 12 | 0 | 0
4 | 19 | 0 | 0
5 | 30595 | 25364 | 0
(6 rows)

name | buffers_written | writebacks | evictions | fsyncs |fsync_time
-------------------+-----------------+-----------+-----------+--------+-----------
client backend | 65 | 0 | 0 | 0 | 0
autovacuum worker | 1259 | 0 | 0 | 0 | 0
checkpointer | 640928 | 640928 | | 591409 | 0
(3 rows)

CHECKPOINT

real 1m22.741s
user 0m0.009s
sys 0m0.000s

07:51:35.353 MSK [104999] LOG: database system is ready to accept connections
07:56:35 .861 MSK [105001] LOG: checkpoint starting: time
07:58:19 .555 MSK [105001] LOG: checkpoint complete: wrote 38493 buffers (58.7%); 0 WAL file(s) added, 0 removed, 27 recycled; write=23.442 s , sync=79.949 s , total=103.694 s ; sync files=40045 , longest=0.072 s, average=0.002 s; distance=441819 kB , estimate=441819 kB; lsn=3B/F9ABE3F0, redo lsn=3B/F9ABE3A8
07:58:19.565 MSK [105001] LOG: checkpoint starting: immediate force wait
07:58:19.604 MSK [105001] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.040 s; sync files=0, longest=0.000 s, average=0.000 s; distance=0 kB, estimate=397637 kB; lsn=3B/F9ABE518, redo lsn=3B/F9ABE4D0
07:58:19.604 MSK [105033] LOG: duration: 82727.854 ms statement: checkpoint ;

name | buffers_written | writebacks | evictions | fsyncs | fsync_time
-------------------+-----------------+-----------+-----------+--------+-----------
client backend | 65 | 0 | 0 | 0 | 0
autovacuum worker | 1259 | 0 | 0 | 0 | 0
checkpointer | 679404 | 679404 | | 631443 | 0
(3 rows)
  difference with previous request: | 38476 | | 40034 |
```

Explanation of commands: a pause of 100 seconds was made (command `sleep 100`) because the creation of tables lasts approximately 3 minutes 30 seconds. 3 minutes 30 seconds plus 100 seconds equals 5 minutes 10 seconds, i.e. the checkpoint in time will already start working.

Values `duration : 82727.854 ms` equals `real 1 m 22.741 s` (the execution time of the checkpoint command **at maximum speed**).

Immediately after the instance is launched `at 07:51:35.353` , the checkpoint is not launched. At the same time, the checkpointer process accumulates lists of blocks for writeback and fsync from other processes.

writeback s in the source code this statistic is called `IOOP_WRITEBACK , incremented by the number of blocks in the` write queue `nr_pending` and is schematically performed as follows:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
  * Executing the writes in-order can make them a lot faster, and allows to
  * merge writeback requests to consecutive blocks into larger writebacks .
  sort_pending_writebacks(wb_context->pending_writebacks, wb_context->nr_pending);
  * Coalesce neighboring writes , but nothing else. For that we iterate
  * through the, now sorted, array of pending flushes, and look forward to
  * find all neighbors (or identical) writes .
  /* and finally tell the kernel to write the data to storage */
  reln = smgropen(currlocator, InvalidBackendId);
  smgrwriteback(reln, BufTagGetForkNum(&tag), tag.blockNum, nblocks); -- Trigger kernel
writeback for the supplied range of blocks .
  pgstat_count_io_op_time(IOOBJECT_RELATION, io_context, IOOP_WRITEBACK , io_start,
wb_context-> nr_pending );
```

`pg_stat_io.fsyncs` - statistics in the source code are called `IOOP_FSYNC`, `are performed` by the system call and are `incremented by` 1 for each file : `fsync ( fd )` .

statistics from tick pg_stat_io . **`fsync`** **`s`** - this is just the number of data files to which the checkpointer wrote dirty blocks. One fsync is performed on one file per checkpoint.

The fsyncs statistics are of no interest, as they are determined by the number of data files. The complexity of execution and the delay in obtaining the result cannot be determined from these statistics, since by the time fsync (fd) is called, Linux may have written the file blocks from the page cache to disk, or it may not have written them.

The checkpoint started at **`07:56:35`** and started sending dirty blocks for writeback. According to the `checkpoint_completion_target parameter,` the checkpoint should writeback dirty blocks in 90% of its allotted interval **`checkpoint _ timeout =`** 5 minutes. The number of dirty buffers gradually decreases, in the example **`dirty=25364`** , after 10 seconds **`26780`** .

`The checkpoint_completion_target` parameter does not affect the frequency of checkpoints, it affects how much time the writeback phase of files takes. If all files are sent for writing and there is no activity, then the checkpoint will not complete by time. The checkpoint will complete either by `checkpoint_timeout` or `if an on` - demand checkpoint is called (admin command, instance stop command, max_wal_size exceeded). After one checkpoint completes, the next one starts immediately.

When does fsync start sending block ranges of individual files? This is determined by the `checkpoint_flush_after configuration parameter .`

`During the` checkpoint, `the pg_stat_io.writes and` pg_stat_io.writebacks `columns are updated .` The `pg_stat_io.fsyncs and` pg_stat_io.fsync_time `columns were updated after the checkpoint` was completed `in the test . However , the pg_stat_io.fsyncs values are also updated during the` checkpoint `. Most likely , the fsyncs statistics` is passed to the view once - after the checkpoint is completed.

For an extended checkpoint, the values **`write =23.442 s , sync =79.949 s , total =103.694 s`** , as well as its beginning (`07:56:35 .861`) and end (`07:58:19 .555`) do not make sense .

Example of indicators for different values of sleep :

```
360 - 1m32.562s (write=269.924 s, sync= 98.376 s, total=368.681 s;) 6:08 duration: 92547.992 ms
310 - 1m39.601s (write=242.044 s, sync= 98.750 s, total=341.174 s;) 5:41 duration: 99587.851 ms
260 - 1m41.145s (write=180.222 s, sync= 99.321 s, total=279.923 s;) 4:40 duration: 101131.358 ms
110 - 1m44.681s (write=30.342 s, sync=100.731 s, total=131.473 s;) 2:11 duration: 104667.304 ms
   real = duration
```

The values before duration are the time elapsed from the start of the extended time checkpoint to its completion, it does not make sense, since the time checkpoint works extended in time, and when the command is given to execute the checkpoint on demand, then the time checkpoint is completed without delay, the moment of completion is the end of the time checkpoint. Values `sync = 98.376 s` in the example are almost identical and correspond to the actual duration of the sync_file_range system calls executed at maximum speed.

The results of queries against **`the pg_stat_io view will`** be discussed in the following chapters. The `*_time columns with time metrics` **`are of interest in this view, but to populate these columns ,`** you need to change the value of a configuration parameter. The query results in the example show that checkpointer sent *`38476`* writebacks and *`40034`* fsync s (sent by

Oleg Ivanov

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

operating system page ranges). The number corresponds to the number of dirty blocks and the number of `files =40045` , since each table consists of 2 blocks (with 300 rows in them).

2) Check what is the start of the last checkpoint according to the control file:

```
postgres@tantor:~$ pg_controldata | grep checkpoint | grep location
Latest checkpoint location: 3B/F9ABE518
Latest checkpoint's REDO location: 3B/F9ABE4D0
```

The values correspond to the entry `lsn =3 B / F 9 ABE 518, redo lsn =3 B / F 9 ABE 4 D 0` in log.

The beginning (`lsn =`) is separated from the end (`redo lsn =`) not far because there was no activity on the cluster from the moment the checkpoint command was issued until it completed.

3) Delete tables :

```
postgres@tantor:~$ time psql -f initdrop.sql 2> /dev/null
DO
Time : 6339.328 ms (00:06.339)
```

The removal was noticeably faster: it was `6979 ms` became `6339 ms` .

The result of this point: after decreasing `max_connections` , deleting tables became faster. Changing the `max_connections` parameter value did not affect creating tables . Excessive increase of `max_connections` reduces performance.

4) This point is not necessary to complete, since the time to execute commands is large. It is enough to look at the results of the commands. It is tested how the increase in `max_connections` and `max_files_per_process` affects the execution time of commands :

```
echo "reset parameters to default values"
psql -c "alter system reset max_locks_per_transaction;"
psql -c "alter system reset log_statement;"
psql -c "alter system reset autovacuum_naptime;"
psql -c "alter system reset max_locks_per_transaction;"
psql -c "alter system reset shared_buffers;"
psql -c "alter system reset checkpoint_timeout;"
echo "max_connections=100, max_files_per_process=1000"
psql -c "alter system reset max_connections;"
psql -c "alter system reset max_files_per_process;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
echo "max_connections=1000, max_files_per_process=1000"
psql -c "alter system set max_connections=1000;"
psql -c "alter system reset max_files_per_process;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
echo "max_connections=100, max_files_per_process=40000"
psql -c "alter system reset max_connections;"
psql -c "alter system set max_files_per_process=40000;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
echo "max_connections=1000, max_files_per_process=40000"
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
psql -c "alter system set max_connections=1000;"
psql -c "alter system set max_files_per_process=40000;"
sudo systemctl restart tantor-se-server-16
time psql -f init1.sql > /dev/null
sudo systemctl restart tantor-se-server-16
time psql -f initdrop.sql 2> /dev/null
psql -c "alter system reset max_connections;"
psql -c "alter system reset max_files_per_process;"
```

```
reset parameters to default values
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
max_connections=100, max_files_per_process=1000
ALTER SYSTEM
ALTER SYSTEM

real 3m27.062s
user 0m1.163s
sys 0m0.462s
Timing is on.
DO
Time: 6692.016 ms (00:06.692)

real 0m6.710s
user 0m0.008s
sys 0m0.001s
max_connections=1000, max_files_per_process=1000
ALTER SYSTEM
ALTER SYSTEM

real 3m47.364s
user 0m1.240s
sys 0m0.483s
Timing is on.
DO
Time: 7257.740 ms (00:07.258)

real 0m7.277s
user 0m0.009s
sys 0m0.000s
max_connections=100, max_files_per_process=40000
ALTER SYSTEM
ALTER SYSTEM

real 3m47.401s
user 0m1.259s
sys 0m0.395s
Timing is on.
DO
Time: 6617.566 ms (00:06.618)

real 0m6.636s
user 0m0.009s
sys 0m0.000s
max_connections=1000, max_files_per_process=40000
ALTER SYSTEM
ALTER SYSTEM

real 3m47.462s
user 0m1.239s
sys 0m0.453s
Timing is on.
DO
Time: 7232.038 ms (00:07.232)

real 0m7.250s
user 0m0.009s
sys 0m0.000s

ALTER SYSTEM
ALTER SYSTEM
```

Increasing `max_files_per_process from 1000 to 40000` did not affect the `test results` .

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Practice for Chapter 12

Vacuum command parameters

1) Look how many transactions ago the TOAST table with the name pg_toast_3394 was frozen :

```
postgres=# select (select b.oid::regclass from pg_class b where
reltoastrelid=a.oid::regclass) table, relname, relfrozenxid, age(relfrozenxid),
relminmxid, mxid_age(relminmxid) from pg_class a where relfrozenxid<>0 and
relname='pg_toast_3394';
 table | relname | relfrozenxid | age | relminmxid | mxid_age
---------------+-------------+-------------+---------+-----------+---------
 pg_init_privs | pg_toast_3394 | 730 | 6748505 | 1 | 0
 (1 row )
```

Table `pg_toast_3394` , as well as its main table `pg_init_privs , have not been frozen for a long time` : 6748505 transactions, multitransactions 1.

2) Freeze rows in the pg_toast_3394 table :

```
postgres=# vacuum (freeze, verbose) pg_toast.pg_toast_3394;
INFO: aggressively vacuuming "postgres.pg_toast.pg_toast_3394"
INFO: finished vacuuming "postgres.pg_toast.pg_toast_3394": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 6749235, which was 0 XIDs old when operation ended
new relfrozenxid: 6749235 , which is 6748505 XIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item
identifiers removed
avg read rate: 29.260 MB/s, avg write rate: 29.260 MB/s
buffer usage: 25 hits, 1 misses, 1 dirtied
WAL usage: 1 records, 1 full page images, 8177 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

3) See how the transaction number for which the freeze was performed and the freeze age have changed for TOAST and its main table:

```
postgres=# select (select b.oid::regclass from pg_class b where
reltoastrelid=a.oid::regclass) table, relname, relfrozenxid, age(relfrozenxid),
relminmxid, mxid_age(relminmxid) from pg_class a where relfrozenxid<>0 and
relname in ('pg_toast_3394', 'pg_init_privs');
 table | relname | relfrozenxid | age | relminmxid | mxid_age
---------------+-------------+-------------+---------+-----------+----------
 pg_init_privs | pg_toast_3394 | 6749235 | 0 | 1 | 0
               | pg_init_privs | 730 | 6748505 | 1 | 0
 (1 row)
```

For a TOAST table, the transaction number for which the table was frozen became `6749235` , age `0` .
You can freeze TOAST tables separately from the main tables.
Vacuum with the free e zee option refers to the "aggressive" vacuum mode.

3) The aggressive vacuuming mode includes the DISABLE _ PAGE _ SKIPPING mode .

```
postgres=# vacuum (DISABLE_PAGE_SKIPPING, verbose) pg_init_privs;
INFO: aggressively vacuuming "postgres.pg_catalog.pg_init_privs"
INFO: finished vacuuming "postgres.pg_catalog.pg_init_privs": index scans: 0
pages: 0 removed, 3 remain, 3 scanned (100.00% of total)
tuples: 4 removed, 227 remain, 0 are dead but not yet removable
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
removable cutoff: 6749235, which was 0 XIDs old when operation ended
new relfrozenxid: 6749235 , which is 6748505 XIDs ahead of previous value
frozen: 1 pages from table (33.33% of total) had 4 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
avg read rate: 158.056 MB/s, avg write rate: 135.477 MB/s
buffer usage: 28 hits, 7 misses, 6 dirtied
WAL usage: 7 records, 6 full page images, 46887 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: aggressively vacuuming "postgres.pg_toast.pg_toast_3394"
INFO: finished vacuuming "postgres.pg_toast.pg_toast_3394": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 6749235, which was 0 XIDs old when operation ended
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 1 hits, 0 misses, 0 dirtied
WAL usage: 0 records, 0 full page images, 0 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Blocks marked in the freeze map are skipped by default. This parameter **DISABLE _ PAGE _ SKIPPING** disables skipping of any blocks, including those marked in the freeze map. The parameter can be used if there is a suspicion that the visibility map (in which the freeze map is stored) is damaged and the map needs to be recreated. The parameter can also be used for tests: to check how long it will take to scan all the blocks of the table.

By default, the TOAST table was vacuumed and frozen again by this command. To disable vacuuming of TOAST tables, there is an option: vaccum (process_toast off).

4) DISABLE_PAGE_SKIPPING mode freezes table rows :

```
postgres=# select (select b.oid::regclass from pg_class b where
reltoastrelid=a.oid::regclass) table, relname, relfrozenxid, age(relfrozenxid),
relminmxid, mxid_age(relminmxid) from pg_class a where relfrozenxid<>0 and
relname in ('pg_toast_3394', 'pg_init_privs');
 table | relname | relfrozenxid | age | relminmxid | mxid_age
---------------+--------------+--------------+-----+------------+----------
 pg_init_privs | pg_toast_3394 | 6749235 | 0 | 1 | 0
               | pg_init_privs | 6749235 | 0 | 1 | 0
(2 rows )
```

Why vacuum or freeze tables separately and TOAST separately? In TOAST, large fields are extracted and stored in the TOAST table rowset. The size of the TOAST table can be much larger than the main table. Usually, the fields stored in TOAST change less often than the fields stored in the main table. When new versions of rows appear in the main table, new versions of rows in TOAST may not be generated.

All tables are usually processed by autovacuum, but you can perform vacuuming separately so that the next autovacuum cycle works faster - skips blocks marked in the table visibility map. If the TOAST table has grown, and the autovacuum threshold is not exceeded, then you can vacuum the TOAST table separately.

5) Do it commands :

```
postgres=# select relname, relfrozenxid, relminmxid from pg_class where
relminmxid<>0 and relname = ' t3 ' order by relfrozenxid::text::numeric desc
limit 100;
 relname | relfrozenxid | relminmxid
---------+--------------+-----------
 t3 | 21854345 | 108925
(1 row)

postgres=# vacuum (verbose) t3 ;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
INFO: vacuuming "postgres.public.t3"
INFO: finished vacuuming "postgres.public. t3 ": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin: 21883901
removable cutoff: 21883901, which was 0 XIDs old when operation ended
new relfrozenxid: 21883901 , which is 29556 XIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers removed
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 65.104 MB/s
buffer usage: 6 hits, 0 misses, 1 dirtied
WAL usage: 1 records, 1 full page images, 7715 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: vacuuming "postgres.pg_toast.pg_toast_3243313"
INFO: finished vacuuming "postgres. pg_toast.pg_toast_3243313 ": index scans: 0
pages: 0 removed, 0 remain, 0 scanned (100.00% of total)
tuples: 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin: 21883901
removable cutoff: 21883901, which was 0 XIDs old when operation ended
new relfrozenxid: 21883901 , which is 29556 XIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (100.00% of total) had 0 dead item identifiers removed
I/O timings: read: 0.037 ms, write: 0.000 ms
avg read rate: 104.167 MB/s, avg write rate: 52.083 MB/s
buffer usage: 28 hits, 2 misses, 1 dirtied
WAL usage: 1 records, 1 full page images, 8103 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
postgres=# select relname, relfrozenxid, relminmxid from pg_class where relminmxid<>0 and relname = ' t3 ' order by relfrozenxid::text::numeric desc limit 100;
relname | relfrozenxid | relminmxid
---------+--------------+-----------
t3 | 21883901 | 108925
(1 row)
```

If table t3 is missing, any other table can be used.

The commands illustrate that the regular vacuum and autovacuum, if possible (if it processes all the blocks that it has not processed before and all the lines in all the blocks end up frozen), perform freezing.

Also, vacuum and autovacuum at the end of their work update `pg_database.datfrozenxid` - the oldest unfrozen XID in database objects.

To obtain a new value for `pg_database.datfrozenxid` **,** XIDs are retrieved from all relations in the pg_class table **that belong to the** database . If **pg_class** many rows, then such a query loads the instance. Loads because the pg_class table can be actively updated. For example, by creating-truncating-deleting-changing temporary and permanent tables. The table may have a large number of row versions, an ineffective index, competition for access to the table buffers in the buffer cache.

6) The VACUUM command has an option SKIP_DATABASE_STATS, which disables query execution and updating of the pg_database.datfrozenxid value. There is also a parameter that does not clear anything, but only executes the query and updates the value:

```
postgres=# \timing on \\
VACUUM(ONLY_DATABASE_STATS VERBOSE);
ERROR: only_database_stats requires a Boolean value
Time: 0.263 ms
```

The result of this command **allows** you to estimate the speed of access to **pg_class . In the example ,** the access speed is 0.263 milliseconds and there are no problems.

7) View the autovacuum configuration parameters:

```
postgres=# select name, setting||coalesce(unit,'') unit, context, min_val, max_val from pg_settings where category='Autovacuum' or name like '%autovacuum%' order by 1;
name | unit | context | min_val | max_val
--------------------------------------+------------+-----------+---------+-----------
autovacuum | on | sighup | |
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
autovacuum_analyze_scale_factor | 0.1 | sighup | 0 | 100
autovacuum_analyze_threshold | 50 | sighup | 0 | 2147483647
autovacuum_freeze_max_age | 10000000000 | postmaster | 100000 | 9223372036854775807
autovacuum_max_workers | 5 | postmaster | 1 | 262143
autovacuum_multixact_freeze_max_age | 20000000000 | postmaster | 10000 | 9223372036854775807
autovacuum_naptime | 60s | sighup | 1 | 2147483
autovacuum_vacuum_cost_delay | 2ms | sighup | -1 | 100
autovacuum_vacuum_cost_limit | -1 | sighup | -1 | 10000
autovacuum_vacuum_insert_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_vacuum_insert_threshold | 1000 | sighup | -1 | 2147483647
autovacuum_vacuum_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_vacuum_threshold | 50 | sighup | 0 | 2147483647
autovacuum_work_mem | -1kB | sighup | -1 | 2147483647
log_autovacuum_min_duration | 600000ms | sighup | -1 | 2147483647
(15 rows )
```

On PostgreSQL versions with a 32-bit transaction counter:

```
postgres@tantor:~$ psql -h 172.18.0.2 -p 5432 -U postgres
postgres=# select name, setting||coalesce(unit,'') unit, context, min_val,
max_val from pg_settings where category='Autovacuum' or name like '%autovacuum%'
order by 1;
```

```
name | unit | context | min_val | max_val
---------------------------------------+-----------+-----------+---------+------------
autovacuum | on | sighup | |
autovacuum_analyze_scale_factor | 0.1 | sighup | 0 | 100
autovacuum_analyze_threshold | 50 | sighup | 0 | 2147483647
autovacuum_freeze_max_age | 200000000 | postmaster | 100000 | 2000000000
autovacuum_max_workers | 3 | postmaster | 1 | 262143
autovacuum_multixact_freeze_max_age | 400000000 | postmaster | 10000 | 2000000000
autovacuum_naptime | 60s | sighup | 1 | 2147483
autovacuum_vacuum_cost_delay | 2ms | sighup | -1 | 100
autovacuum_vacuum_cost_limit | -1 | sighup | -1 | 10000
autovacuum_vacuum_insert_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_vacuum_insert_threshold | 1000 | sighup | -1 | 2147483647
autovacuum_vacuum_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_vacuum_threshold | 50 | sighup | 0 | 2147483647
autovacuum_work_mem | -1kB | sighup | -1 | 2147483647
log_autovacuum_min_duration | 600000ms | sighup | -1 | 2147483647
(15 rows )
```

Part 2: Vacuum Observation

1) Create 10000 tables. To do this, run the commands:

```
postgres@tantor:~$ time psql -f init1.sql > /dev/null

real 3m45.019s
user 0m1.262s
sys 0 m 0.438 s
```

2) Run the commands:

```
postgres@tantor:~$
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &

for (( i=1; i <= 10; i+=1 ))
do
sleep 0.7
psql -c "select pid, relid::regclass, phase, heap_blks_total total,
heap_blks_scanned scanned, max_dead_tuples max_dead, num_dead_tuples dead from
pg_stat_progress_vacuum;"
done
[1] 142424
[2] 142425
[3] 142427
pid | release | phase | total | scanned | max_dead | dead
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
--------+--------------+--------------+------+---------+---------+------
142431 | pg_attribute | scanning heap | 3757 | 1242 | 1093287 | 2562
142432 | bencher | scanning heap | 54055 | 2805 | 291 | 0
142433 | t2 | scanning heap | 3847 | 1202 | 291 | 0
(3 rows)

pid | release | phase | total | scanned | max_dead | dead
--------+---------+-------------+-------+---------+---------+------
142432 | bencher | scanning heap | 54055 | 17732 | 291 | 0
(1 row)

pid | release | phase | total | scanned | max_dead | dead
--------+---------+-------------+-------+---------+---------+------
142431 | bencher | scanning heap | 54055 | 7192 | 291 | 0
142432 | test.film_summary561 | scanning heap | 2 | 2 | 291 | 0
(2 rows)

pid | release | phase | total | scanned | max_dead | dead
--------+--------+----------+-------+---------+---------+------
142431 | bencher | scanning heap | 54055 | 21602 | 291 | 0
142432 | test.film_summary2137 | performing final cleanup | 2 | 2 | 291 | 0
(2 rows)


pid | release | phase | total | scanned | max_dead | dead
--------+-------------------------+-------------+-------+---------+---------+------
142431 | pg_toast.pg_toast_7245268 | initializing | 0 | 0 | 0 | 0
142432 | test.film_summary7072 | scanning heap | 2 | 0 | 291 | 0
142433 | bencher | scanning heap | 54055 | 7625 | 291 | 0
(3 rows)

postgres@tantor:~$ VACUUM

real 0m7.666s
user 0m0.009s
sys 0m0.000s
VACUUM

real 0m10.108s
user 0m0.010s
sys 0m0.000s
VACUUM

real 0m12.971s
user 0m0.011s
sys 0m0.000s
```

launched simultaneously on the database. The DISABLE_PAGE_SKIPPING option **was used so that the vacuums scanned all table blocks, i.e. worked** for some time.

The periodic calls to the pg_stat_progress_vacuum view **show which vacuum or autovacuum processes are executing vacuuming at the time of the request. In the example, it** is clear that 3,1,2,2,3 commands processed objects in sequence **.**

To obtain the table name, it is convenient to use type casting: **relid :: regclass** .

3) Press <Enter> on your keyboard to return the prompt:

```
<Enter>
[1] Done time psql -c "vacuum (DISABLE_PAGE_SKIPPING);"
[2]- Done time psql -c "vacuum (DISABLE_PAGE_SKIPPING);"
[3]+ Done time psql -c "vacuum (DISABLE_PAGE_SKIPPING);"

postgres @ tantor :~$
```

Presentation **pg_stat_progress_vacuum does not** show the operation of the VACUUM FULL command. The operation of the VACUUM_FULL command is shown in the pg_stat_progress_cluster view :

```
postgres=# select * from pg_stat_progress_ cluster\ gx
-[ RECORD 1 ]-------+------------
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
pid | 140712
datid | 5
datname | postgres
relid | 7251678
command | VACUUM FULL
phase | initializing
cluster_index_relid | 0
heap_tuples_scanned | 0
heap_tuples_written | 0
heap_blks_total | 0
heap_blks_scanned | 0
index_rebuild_count | 0
```

VACUUM FULL is variety CLUSTER commands .

4) It is inconvenient that the time the command lasts is not shown and there is no data on whether the process is blocked. To obtain this data, you can connect the `query with the pg_stat_activity view` . Run commands :

```
postgres@tantor:~$
time psql -c "vacuum (DISABLE_PAGE_SKIPPING);" &
sleep 3

psql -c "SELECT p.pid, clock_timestamp() - a.xact_start AS duration,
coalesce(wait_event_type ||'.'|| wait_event, 'f') AS waiting,
CASE
WHEN a.query ~*'^autovacuum.*to prevent wraparound' THEN 'wraparound'
WHEN a.query ~*'^vacuum' THEN 'user' ELSE 'regular' END AS mode,
p.datname AS database, p.relid::regclass AS table, p.phase,
pg_size_pretty(p.heap_blks_total * current_setting('block_size')::int) AS
table,
pg_size_pretty(pg_total_relation_size(relid)) AS total,
pg_size_pretty(p.heap_blks_scanned * current_setting('block_size')::int) AS
scanned,
pg_size_pretty(p.heap_blks_vacuumed * current_setting('block_size')::int) AS
vacuumed,
round(100.0 * p.heap_blks_scanned / p.heap_blks_total, 1) AS scan_pct,
round(100.0 * p.heap_blks_vacuumed / p.heap_blks_total, 1) AS vacuum_pct,
p.index_vacuum_count Loops,
round(100.0 * p.num_dead_tuples / p.max_dead_tuples,1) AS dead_pct
FROM pg_stat_progress_vacuum p
JOIN pg_stat_activity a using (pid)
ORDER BY clock_timestamp()-a.xact_start desc;"

pid |duration|waiting|mode|database| table | phase | table | total | scanned | vacuumed | scan_pct | vacuum_pct | loops | dead_pct
------------+--------+-------+----+--------+---------+---------------+--------+--------+---------+----------+----------+------------+-------+----
-----
142669|00:00:03| f |user|postgres| bencher | scanning heap | 422 MB | 422 MB | 402 MB | 0 bytes | 95.2 | 0.0 | 0 | 0.0
(1 row )
```

5) Run a query that shows whether the table needs to be vacuumed or analyzed according to the configuration parameters :

```
postgres=# WITH s AS (
SELECT
 current_setting('autovacuum_analyze_scale_factor')::float8 AS analyze_factor,
current_setting('autovacuum_analyze_threshold')::float8 AS analyze_threshold,
current_setting('autovacuum_vacuum_scale_factor')::float8 AS vacuum_factor,
current_setting('autovacuum_vacuum_threshold')::float8 AS vacuum_threshold
)
SELECT nspname, relname, n_dead_tup, v_threshold, n_mod_since_analyze, a_threshold,
CASE WHEN n_dead_tup > v_threshold THEN 'yes' ELSE 'no' END AS do_vacuum,
CASE WHEN n_mod_since_analyze > a_threshold THEN 'yes' ELSE 'no' END AS do_analyze,
```

```
pg_relation_size(relid) AS relsize,
pg_total_relation_size(relid) AS total
FROM (SELECT n.nspname, c.relname, c.oid AS relid, t.n_dead_tup, t.n_mod_since_analyze,
trunc(c.reltuples * s.vacuum_factor + s.vacuum_threshold) AS v_threshold,
trunc(c.reltuples * s.analyze_factor + s.analyze_threshold) AS a_threshold
FROM s, pg_class c
JOIN pg_namespace n ON c.relnamespace = n.oid
JOIN pg_stat_all_tables t ON c.oid = t.relid
WHERE c.relkind = 'r')
WHERE n_dead_tup > v_threshold OR n_mod_since_analyze > a_threshold
ORDER BY nspname, relname limit 5;

nspname | relname | n_dead_tup | v_threshold | n_mod_since_analyze | a_threshold | do_vacuum | do_analyze | relsize | total
------------+----------------+------------+------------+--------------------+-------------+-----------+------------+-----------+-----------
pg_catalog | pg_statistic | 0 | 142 | 19709 | 96 | no | yes | 229376 | 376832
public | bencher | 0 | 2000050 | 60000000 | 1000050 | no | yes | 442818560 | 442966016
test | film_summary1 | 0 | 110 | 301 | 80 | no | yes | 16384 | 57344
test | film_summary10 | 0 | 110 | 301 | 80 | no | yes | 16384 | 57344
test | film_summary100 | 0 | 110 | 301 | 80 | no | yes | 16384 | 57344
(5 rows)
```

The query does not take into account table-level settings, only cluster-level settings, so the list of tables that will be vacuumed or analyzed is not exact.

6) Look at the example commands:

```
postgres=# VACUUM FULL FREEZE VERBOSE ANALYZE t3;
INFO: vacuuming "public.t3"
INFO: "public.t3": found 0 removable, 0 nonremovable row versions in 0 pages
DETAIL: 0 dead row versions cannot be removed yet.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO: analyzing "public.t3"
INFO: "t3": scanned 0 of 0 pages, containing 0 live rows and 0 dead rows; 0 rows in sample, 0 estimated total rows
VACUUM
postgres=# VACUUM VERBOSE FULL t3;
ERROR: syntax error at or near "FULL"
LINE 1: VACUUM VERBOSE FULL t3;
^
postgres=# VACUUM (VERBOSE, FULL) t3;
INFO: vacuuming "public.t3"
INFO: "public.t3": found 0 removable, 0 nonremovable row versions in 0 pages
DETAIL: 0 dead row versions cannot be removed yet.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

The command without parentheses requires that the parameters be specified only in the order specified in the first example. When using parentheses and commas, the order is not important.
The case of keywords is not important.

7) Delete the created tables so that they don't take up space:

```
postgres@tantor:~$ psql -f initdrop.sql 2> /dev/null
```

Part 3. Extension for viewing visibility map and freezing pg_visibility

1) Install extension :

```
postgres=# create extension pg_visibility;
CREATE EXTENSION
```

Allows you to view the visibility map and freeze map and recreate them.
These maps are stored in a vm layer file.

2) See what objects are included in the extension:

```
postgres=# \dx+ pg_visibility
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
Objects in extension "pg_visibility"
Object description
-----------------------------------------------
function pg_check_frozen(regclass)
function pg_check_visible(regclass)
function pg_truncate_visibility_map(regclass)
function pg_visibility_map(regclass)
function pg_visibility_map(regclass,bigint)
function pg_visibility_map_summary(regclass)
function pg_visibility(regclass)
function pg_visibility(regclass,bigint)
(8 rows)
```

The extension has 8 functions.

3) Create a temporary table and see how it affects the visibility map of the pg_class table:

```
postgres=# select * from pg_visibility_map_summary('pg_class');
create temp table temp1 (n numeric);
select * from pg_visibility_map_summary('pg_class');
drop table temp1;
all_visible | all_frozen
-------------+------------
792 | 6
(1 row)

CREATE TABLE
all_visible | all_frozen
-------------+------------
790 | 6
(1 row )

DROP TABLE
```

The visibility map has changed. The number of blocks in which all rows are visible has decreased.

4) Perform request :

```
postgres=# select pd_all_visible, count(all_visible) filter (where
all_visible=true) true, count(all_visible) filter (where all_visible=false) false
from pg_visibility('pg_class') group by pd_all_visible;
pd_all_visible | true | false
----------------+-----------+-------
 f | 0 | 27
t | 786 | 0
(2 rows)
```

Using this query, you can get detailed information on table blocks from the visibility map. Grouping is performed by the PD_ALL_VISIBLE bit. The function reads not only the visibility map, but also all the table blocks, since the PD _ ALL _ VISIBLE bit is only in the block headers. There is a similar function that does not have **pd _ all _ visible** and that uses the visibility map: `pg_visibility_map()` .

5) Run the commands:

```
postgres=# select pd_all_visible, count(all_visible) filter (where
all_visible=true) true, count(all_visible) filter (where all_visible=false) false
from pg_visibility('pgbench_accounts') group by pd_all_visible;
select pg_truncate_visibility_map('pgbench_accounts');
```

```
  select pd_all_visible, count(all_visible) filter (where all_visible=true) true,
count(all_visible) filter (where all_visible=false) false from
pg_visibility('pgbench_accounts') group by pd_all_visible;
  vacuum pgbench_accounts;
  select pd_all_visible, count(all_visible) filter (where all_visible=true) true,
count(all_visible) filter (where all_visible=false) false from
pg_visibility('pgbench_accounts') group by pd_all_visible;

  pd_all_visible | true | false
  ----------------+-----------+-------
  t | 1640 | 0
  (1 row)

  pg_truncate_visibility_map
  ----------------------------

  (1 row)

  pd_all_visible | true | false
  ----------------+-----------+-------
  t | 0 | 1640
  (1 row)

  VACUUM
  pd_all_visible | true | false
  ----------------+-----------+-------
  t | 1640 | 0
  (1 row)
```

Function **pg_truncate_visibility_map()** got it map visibility . After truncation, all blocks were considered **not-** "all_visible". The VACUUM command scanned **all blocks** in the table and rebuilt the visibility map.

Part 4. Interval between autovacuum cycles, parameter autovacuum_naptime

1) Run the commands:

```
postgres @ tantor :~$
rm -f $ PGDATA / log / *
psql -c "alter system set log_autovacuum_min_duration = 0;"
psql -c "alter system set logging_collector = on;"
psql -c "alter system set autovacuum_naptime = '3s';"
sudo systemctl restart tantor-se-server-16
psql -c "drop table if exists test;"
time psql -c "CREATE TABLE test AS SELECT * FROM generate_series(1, 10000000) x(id);"
time psql -c "CREATE INDEX ON test(id);"
cat $PGDATA/log/postgresql-* | tail -3
sleep 35
cat $PGDATA/log/postgresql-* | tail -20
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
DROP TABLE
SELECT 10000000

real 0m14.780s
user 0m0.009s
sys 0m0.000s
CREATE INDEX

real 0m8.712s
user 0m0.009s
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
sys 0m0.000s
00:19: 25 .571 MSK [17483 0 ] LOG: skipping vacuum of "test" --- lock not available
00:19: 28 .573 MSK [17483 1 ] LOG: skipping vacuum of "test" --- lock not available
00:19: 31 .575 MSK [17483 2 ] LOG: skipping vacuum of "test" --- lock not available
00:19:10.454 MSK [174811] LOG: database system is ready to accept connections
00:19: 25 .571 MSK [17483 0 ] LOG: skipping vacuum of "test" --- lock not available
00:19: 28 .573 MSK [17483 1 ] LOG: skipping vacuum of "test" --- lock not available
00:19: 31 .575 MSK [17483 2 ] LOG: skipping vacuum of "test" --- lock not available
00:20:00 .777 MSK [ 174835 ] LOG: automatic vacuum of table "postgres.public.test": index scans:
0
pages: 0 removed, 44248 remain, 44248 scanned (100.00% of total)
tuples: 0 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin: 22095797
removable cutoff: 22095797, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
I/O timings: read: 419.394 ms, write: 529.628 ms
avg read rate: 12.566 MB/s, avg write rate: 12.567 MB/s
buffer usage: 46403 hits, 42137 misses, 42141 dirtied
WAL usage: 44252 records, 5 full page images, 2917445 bytes
system usage: CPU: user: 2.19 s, system: 1.68 s, elapsed: 26.19 s
00:20:02.498 MSK [ 174835 ] LOG: automatic analyze of table "postgres.public.test"
I/O timings: read: 202.989 ms, write: 0.289 ms
avg read rate: 129.678 MB/s, avg write rate: 0.014 MB/s
buffer usage: 1526 hits, 28583 misses, 3 dirtied
system usage: CPU: user: 0.37 s, system: 0.46 s, elapsed: 1.72 s
```

The commands set the output of all autovacuum messages to the diagnostic log, the interval for starting autovacuum cycles is 3 seconds.

2) A table with a large number of rows was created. In accordance with the parameters:

```
postgres =# \ dconfig * autovac * inse *
       List of configuration parameters
Parameter | Value
------------------------+-------
autovacuum_vacuum_insert_scale_factor | 0.2
autovacuum_vacuum_insert_threshold | 1000
(2 rows)
```

worker process `[17483 0 ]` started within 3 seconds of table creation.

3) Workflows are not running all the time, only autovacuum is running all the time launcher :

```
ps -ef | grep vacuum
postgres 1019 929 0 Jan12 ? 00:00:01 postgres: 15/main: autovacuum launcher
postgres 174817 174811 0 00:19 ? 00:00:00 postgres: autovacuum launcher
postgres 175144 140533 0 00:31 pts/0 00:00:00 grep vacuum
```

One of the worker processes is sent to the database with the table test and tries to vacuum it. The table was locked by the create index command for `8,712 s` .

autovacuum worker process was launched every 3 seconds and reported that it tried to vacuum the table, but could not and skipped it, after which it stopped. There may be 2 or 3 such messages, since the autovacuum launch cycle can begin at any time from zero to 3 seconds after the table is created. There may be one message if the process vacuumed for more than 8 seconds. In this example, except for the test table, nothing should have been vacuumed and the autovacuum processes stopped immediately after launch.

For the fourth time, at `00:19: 31 + 3` seconds, autovacuum worker started with process number `[ 174835 ]` it vacuumed and analyzed the table, worked `26.19 s` + `1.72 s` and created a message in the log after 26 seconds, at **`00:20:00.`**

Part 5. Comparison of vacuum version 17 with previous versions

The algorithm of vacuum and autovacuum operation has significantly improved since version 17. PostgreSQL version 17 was installed in a virtual machine in a docker container. Let's check which vacuum indicators have improved.

1) Connect to PostgreSQL 17 in the container:

```
postgres@tantor:~$
sudo docker restart postgres
sudo docker inspect postgres | grep 172
                  " Gateway ": "172.18.0.1",
" IPAddress ": " 172.18.0.2 ",
```

If the command does not produce any results, proceed to the next step.
If the command outputs an IPAdderess, connect to it:

```
postgres@tantor:~$ psql -h 172.18.0.2 -p 5432 -U postgres
psql (16.2, server 17.2 (Debian 17.2-1.pgdg120+1))
WARNING: psql major version 16, server major version 17.
Some psql features might not work.
Type "help" for help.
postgres=# \timing on
Timing is on .
```

If the connection is successful, skip the next step and go to step 3.

2) Commands for recreating the container if the previous step failed to connect to PostgreSQL 17:

```
postgres@tantor:~$
sudo docker rm -f postgres
sudo docker network create postgres
sudo docker run -d --init --network postgres -e DB_HOST=127.0.0.1 -e
POSTGRES_USER=postgres -e POSTGRES_PASSWORD=postgres -e POSTGRES_INITDB_ARGS="--
data-checksums" -e POSTGRES_HOST_AUTH_METHOD=trust -p 5434:5434 -e
PGDATA=/var/lib/postgresql/data -d -v /root/data:/var/lib/postgresql/data --name
postgres postgres
sudo docker inspect postgres | grep 172
"Gateway": "172.18.0.1",
" IPAddress ": " 172.18.0.2 ",
postgres@tantor:~$ psql -h 172.18.0.2 -p 5432 -U postgres
psql (16.2, server 17.2 (Debian 17.2-1.pgdg120+1))
WARNING: psql major version 16, server major version 17.
Some psql features might not work.
Type "help" for help.
postgres=# \timing on
Timing is on.
```

3) Do it test :

```
postgres=# alter system set max_wal_size = '2GB';
alter system set checkpoint_timeout='30min';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;
```

Increase **max_wal_size** It is necessary that during the test the checkpoint is not executed on demand and does not load the I/O, which will affect the metrics.

The table is created with autovacuum disabled to make it easier to obtain the test result: with the vacuum verbose command. With autovacuum the result would be the same, only it would need to be viewed in the cluster diagnostic log.

Result of executing commands:

```
ALTER SYSTEM
Time : 23.768 ms
ALTER SYSTEM
Time: 19.945 ms
pg_reload_conf
----------------
t
(1 row)

Time: 4.246 ms
DROP TABLE
Time: 153.615 ms
SELECT 10000000
Time: 13811.083 ms (00:13.811)
CREATE INDEX
Time: 7457.785 ms (00:07.458)
UPDATE 10000000
Time: 67207.156 ms (01:07.207)
CHECKPOINT
Time: 3631.462 ms (00:03.631)
INFO: vacuuming "postgres.public.test"
INFO: finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 88535 remain, 88535 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable
removable cutoff: 757, which was 0 XIDs old when operation ended
new relfrozenxid: 757, which is 3 XIDs ahead of previous value
frozen: 44249 pages from table (49.98% of total) had 10000000 tuples frozen
index scan needed: 44248 pages from table (49.98% of total) had 10000000 dead item identifiers
removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 45.658 MB/s, avg write rate: 49.977 MB/s
buffer usage: 106711 hits, 169524 misses, 185560 dirtied
WAL usage: 364397 records , 143159 full page images, 1005718014 bytes
system usage: CPU: user: 8.41 s, system: 1.05 s, elapsed: 29.00 s
VACUUM
Time: 29007.776 ms (00:29.008)
DROP TABLE
data_checksums
----------------
on
(1 row)
```

4) Connect to PostgreSQL 16 or earlier installed in the virtual machine:

```
postgres@tantor:~$ sudo systemctl restart postgresql
postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
postgres@tantor:~$ psql -p 543 5
psql (16.2, server 15.6 (Debian 15.6-astra.se2))
Type "help" for help.
postgres=# \timing on
Timing is on.
```

5) Perform the test:

```
postgres=# alter system set max_wal_size = '2GB';
alter system set checkpoint_timeout='30min';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;
show data_checksums;
ALTER SYSTEM
Time: 36.365 ms
ALTER SYSTEM
Time: 21.414 ms
pg_reload_conf
----------------
t
(1 row)

Time: 3.999 ms
ERROR: table "test" does not exist
Time: 1.948 ms
SELECT 10000000
Time: 19315.446 ms (00:19.315)
CREATE INDEX
Time: 13423.928 ms (00:13.424)
UPDATE 10000000
Time: 81646.528 ms (01:21.647)
CHECKPOINT
Time: 3838.599 ms (00:03.839)
INFO: clearing "postgres.public.test"
INFO: Finished cleaning "postgres.public.test": index scans: 1
pages removed: 0, remaining: 108109, scanned: 108109 (100.00% of total)
string versions: removed: 10000000, remaining: 10000000, "dead" but not yet subject to removal: 0
XID of deletion cutoff: 1098324, at the time of operation completion it had age: 0 XID
new value relfrozenxid: 1098323, it moved forward from the previous value by 2 XID
index scan required: dead element IDs removed from table pages (54055, 50.00% of total): 10000000
index "test_id_idx": total pages: 77011, currently deleted: 0, deleted at the moment: 0, free: 0
Average read speed: 37.580 MB/s, Average write speed: 40.359 MB/s
buffer usage: hits: 126331, misses: 221000, dirty writes: 237341
WAL usage: records: 292850 , full page images: 130689, bytes: 362929097
system load: CPU: user: 6.53 sec, system: 1.09 sec, elapsed: 45.94 sec
VACUUM
Time: 45947.234 ms (00:45.947)
DROP TABLE
Time: 218.371 ms
data_checksums
----------------
 off
(1 row )
```

In the example, checksum calculation is disabled, so it is impossible to compare the test results with clusters with checksums enabled: vacuum writes 2.77 times fewer `bytes to WAL: 362929097` , not `1005718014 bytes` . Even so, vacuum version 17 worked faster: in **`29007.776 ms`** , compared to version 15, which worked in **`45947.234 ms`** .

The test shows that version 17 in Docker executed commands overall faster than version 15 **not** in Docker.

6) Run the test by connecting to the cluster with checksums enabled:

```
postgres@tantor:~$ sudo systemctl stop postgresql
postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
postgres@tantor:~$ psql
postgres=# \timing on \\
alter system set max_wal_size = '2GB';
alter system set checkpoint_timeout='30min';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
vacuum verbose test;
drop table test;
show data_checksums;

ALTER SYSTEM
Time: 36.604 ms
ALTER SYSTEM
Time: 22.717 ms
pg_reload_conf
----------------
t
(1 row)

Time: 3.678 ms
NOTICE: table "test" does not exist, skipping
DROP TABLE
Time: 2.644 ms
SELECT 10000000
Time: 14866.174 ms (00:14.866)
CREATE INDEX
Time: 10817.373 ms (00:10.817)
UPDATE 10000000
Time: 80073.956 ms (01:20.074)
CHECKPOINT
Time: 3870.459 ms (00:03.870)
INFO: vacuuming "postgres.public.test"
INFO: finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 88496 remain, 88496 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin: 22095818
removable cutoff: 22095818, which was 0 XIDs old when operation ended
new relfrozenxid: 22095818, which is 3 XIDs ahead of previous value
frozen: 44249 pages from table (50.00% of total) had 10000000 tuples frozen
index scan needed: 44248 pages from table (50.00% of total) had 10000000 dead item identifiers removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 2326.127 ms, write: 1499.600 ms
avg read rate: 21.517 MB/s, avg write rate: 23.612 MB/s
buffer usage: 106917 hits, 169199 misses, 185676 dirtied
WAL usage: 364398 records , 143159 full page images, 1008665968 bytes
system usage: CPU: user: 11.12 s, system: 6.48 s, elapsed: 61.43 s
VACUUM
Time: 61437.534 ms (01:01.438)
DROP TABLE
Time: 181.806 ms
data_checksums
----------------
on
(1 row)

Time: 0.117 ms
```

The test shows that version 17 in Docker executed commands overall faster than version 16 **without** Docker. The checkpoint was executed slightly faster in Docker `3631.462 ms` versus `3870.459 ms` in both DBMS without Docker. Vacuum execution time for version 17 in Docker: `29.00 s` vs. `61.43 s` in version 16 outside of Docker. Probably, running in Docker changes the interaction with I/O. The smaller discrepancy in checkpoint execution times allows us to hope that running in Docker does not worsen fault tolerance.

The result of this test: it is impossible to compare DBMSs running in Docker and outside of Docker. For comparison, you need the same environment in which the programs are executed and the same settings.

For example, the inclusion of checksums in version 15 can be compared to the operation of the vacuum with version 16.

7) Let's fix the shortcoming of the previous test. Enable checksum calculation for data blocks in the cluster version 15 and repeat the test:

```
postgres@tantor:~$
sudo systemctl stop postgresql
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
/usr/lib/postgresql/15/bin/pg_checksums -e -D /var/lib/postgresql/15/main
sudo systemctl start postgresql
psql -p 5435
Checksum operation completed
Files scanned: 1262
Blocks scanned: 6290
Files written: 1045
Blocks written: 6290
pg_checksums: syncing data directory
pg_checksums: updating control file
Checksums enabled in cluster
postgres@tantor:~$ psql -p 543 5
psql (16.2, server 15.6 (Debian 15.6-astra.se2))
Type "help" for help.
postgres=# \timing on \\
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;

SELECT 10000000
Time: 15648.954 ms (00:15.649)
CREATE INDEX
Time: 9550.452 ms (00:09.550)
UPDATE 10000000
Time: 74725.766 ms (01:14.726)
CHECKPOINT
Time: 4463.721 ms (00:04.464)
INFO: clearing " postgres . public . test "
INFO: Finished cleaning up " postgres . public . test ": index scans: 1
pages removed: 0, remaining: 108109, scanned: 108109 (100.00% of total)
string versions: removed: 10000000, remaining: 10000000, "dead" but not yet subject to removal: 0
XID of deletion cutoff: 1098418, at the time of operation completion it had age: 0 XID
new value relfrozenxid : 1098417, it moved forward from the previous value by 2 XID
index scan required: dead element IDs removed from table pages (54055, 50.00% of total): 10000000
index " test _ id _ idx ": total pages: 77011, currently deleted: 0, deleted at the moment: 0,
free: 0
Average read speed: 26.511 MB/s, Average write speed: 28.472 MB/s
buffer usage: hits: 126332, misses: 220999, dirty writes: 237349
WAL usage : records: 400976, full page images: 184760, bytes: 1229390628
system load: CPU : user: 7.23 sec, system: 1.90 sec, elapsed: 65.12 sec
VACUUM
Time : 65130.002 ms (01:05.130)
```

The volume of data written `1229386288` is greater than `1005718014 bytes` because the size of the header of each row in Astralinux PostgreSQL is 8 bytes larger, which are used to store the access label. The vacuum time of **`65.12 s`** , taking into account the correction for the record size, corresponds to **`61.43 s`** . There were no changes in the vacuum code between versions 15 and 16. The increase in the checkpoint duration is proportional to the increase in the volume of log records (with an accuracy of up to the spread of test results, if you repeat the tests several times): **`4463.721 ms / 3838.599 ms = 1.1628`** , `1229390628 / 1005718014 = 1.2224` .

Part 6. Number of index scans during vacuuming

The main advantage of the vacuum version 17 is not in the reduction of the operating time, but in significantly smaller ones: memory consumption, the probability of repeated passage through all table indexes. Let's set 2MB of memory for the vacuum and check how many passes the vacuum will make through the index in version 17 and the previous one.

1) Connect to PosqgreSQL version 17:

```
postgres @ tantor :~$ psql - h 172.18.0.2 - p 5432 - U postgres
psql (16.2, server 17.2 (Debian 17.2-1.pgdg120+1))
WARNING: psql major version 16, server major version 17.
Some psql features might not work.
Type "help" for help.
postgres=#
```

2) Perform a test with reduced memory size for vacuuming:

```
postgres=# \timing on\\
alter system set maintenance_work_mem='2MB';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;
alter system reset maintenance_work_mem;
alter system reset max_wal_size;
select pg_reload_conf();
checkpoint;
ALTER SYSTEM
Time: 24.098 ms
pg_reload_conf
----------------
t
(1 row)

Time: 1.005 ms
SELECT 10000000
Time: 13332.139 ms (00:13.332)
CREATE INDEX
Time: 11931.761 ms (00:11.932)
UPDATE 10000000
Time: 74425.280 ms (01:14.425)
CHECKPOINT
Time: 4748.177 ms (00:04.748)
INFO: vacuuming "postgres.public.test"
INFO: finished vacuuming "postgres.public.test": index scans: 2
pages: 0 removed, 88535 remain, 88535 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable
removable cutoff: 765, which was 0 XIDs old when operation ended
new relfrozenxid: 765, which is 3 XIDs ahead of previous value
frozen: 44249 pages from table (49.98% of total) had 10000000 tuples frozen
index scan needed: 44248 pages from table (49.98% of total) had 10000000 dead item identifiers
removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
avg read rate: 57.308 MB/s, avg write rate: 49.041 MB/s
buffer usage: 114465 hits, 216614 misses, 185368 dirtied
WAL usage: 364398 records, 143159 full page images, 1005723308 bytes
system usage: CPU: user: 8.90 s, system: 1.04 s, elapsed: 29.53 s
VACUUM
Time: 29543.681 ms (00:29.544)
DROP TABLE
ALTER SYSTEM
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row )
```

2 passes through the index were performed.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

3) Run the same test on version 16:

```
postgres@tantor:~$ psql
postgres=# \timing on \\
alter system set maintenance_work_mem='2MB';
select pg_reload_conf();
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
drop table test;
alter system reset max_wal_size;
select pg_reload_conf();
checkpoint;
ALTER SYSTEM
ALTER SYSTEM
Time: 28.164 ms
pg_reload_conf
----------------
t
(1 row)

Time: 9.007 ms
SELECT 10000000
Time: 15005.658 ms (00:15.006)
CREATE INDEX
Time: 14352.266 ms (00:14.352)
UPDATE 10000000
Time: 73097.796 ms (01:13.098)
CHECKPOINT
Time: 3982.094 ms (00:03.982)
INFO: vacuuming "postgres.public.test"
INFO: finished vacuuming "postgres.public.test": index scans: 29
pages: 0 removed, 88496 remain, 88496 scanned (100.00% of total)
tuples: 10000000 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin:
22095822
removable cutoff: 22095822, which was 0 XIDs old when operation ended
new relfrozenxid: 22095822, which is 3 XIDs ahead of previous value
frozen: 44249 pages from table (50.00% of total) had 10000000 tuples frozen
index scan needed: 44248 pages from table (50.00% of total) had 10000000 dead item identifiers
removed
index "test_id_idx": pages: 54840 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 10293.486 ms, write: 1492.231 ms
avg read rate: 142.216 MB/s, avg write rate: 17.738 MB/s
buffer usage: 326195 hits, 1485553 misses, 185284 dirtied
WAL usage: 364426 records, 143159 full page images, 1008723444 bytes
system usage: CPU: user: 19.79 s, system: 20.94 s, elapsed: 81.60 s
VACUUM
Time: 81608.506 ms (01:21.609)
DROP TABLE
ALTER SYSTEM
```

Number of passes through index `index scans : 29` , and on version 17 `index scans : 2` .

4) How much memory is needed to use 2 passes on versions before 17? Connect to port 5435 of the version and run the test with `maintenance _ work _ mem ='29 MB '` :

```
postgres@tantor:~$ psql -p 543 5
psql (16.2, server 15.6 (Debian 15.6-astra.se2))
Type "help" for help.
postgres=# \timing on \\
alter system set maintenance_work_mem='29MB';
select pg_reload_conf();
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
drop table if exists test;
create table test with (autovacuum_enabled=off) as select * from
generate_series(1, 10000000) x(id);
create index on test(id);
update test set id = id - 1;
checkpoint;
vacuum verbose test;
alter system reset max_wal_size;
select pg_reload_conf();
checkpoint;
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row)

SELECT 10000000
Time: 15607.619 ms (00:15.608)
CREATE INDEX
Time: 10548.036 ms (00:10.548)
UPDATE 10000000
Time: 73894.497 ms (01:13.894)
CHECKPOINT
Time: 5350.487 ms (00:05.350)
INFO: clearing " postgres . public . test "
INFO: Finished cleaning "postgres.public.test": index scans: 2
pages removed: 0, remaining: 108109, scanned: 108109 (100.00% of total)
string versions: removed: 10000000, remaining: 10000000, "dead" but not yet subject to removal: 0
XID of deletion cutoff: 1098442, at the time of operation completion it had age: 0 XID
new value relfrozenxid: 1098441, it moved forward from the previous value by 2 XID
index scan required: dead element IDs removed from table pages (54055, 50.00% of total): 10000000
index "test_id_idx": total pages: 77011, currently deleted: 0, deleted at the moment: 0, free: 0
Average read speed: 34.614 MB/s, Average write speed: 28.333 MB/s
buffer usage: hits: 134768, misses: 289578, dirty writes: 237034
WAL usage: records: 400977, full page images: 184760, bytes: 1229393586
system load: CPU: user: 7.57 sec, system: 1.94 sec, elapsed: 65.35 sec
VACUUM
Time: 65358.960 ms (01:05.359)
DROP TABLE
Time: 214.219 ms
ALTER SYSTEM
Time: 29.511 ms
```

If install `maintenance_work_mem='28MB'` , then there will be 3 passes By index :

```
INFO: Finished cleaning "postgres.public.test": index scans: 3
Average read speed: 41.495 MB/s, Average write speed: 27.448 MB/s
buffer usage: hits: 143212, misses: 358149, dirty writes: 236911
system load: CPU: user: 7.97 sec, system: 2.27 sec, elapsed: 67.43 sec
Time: 67433.238 ms (01:07.433)
```

When performing vacuuming, you can output the memory allocation of the server process or autovacuum worker to the cluster log. The bulk of the memory will be allocated in the transaction context `TopTransactionContext` . Example:

```
LOG: logging memory contexts of PID 464865
[464865] LOG: level: 0; TopMemoryContext: 151040 total in 7 blocks; 14032 free (29 chunks);
137008 used
[464865] LOG: level: 1; TopTransactionContext: 33562672 total in 2 blocks; 2904 free (0 chunks);
33559768 used
...
[464865] LOG: Grand total: 35502216 bytes in 233 blocks; 727992 free (221 chunks); 34774224 used
```

When you repeatedly execute the log output while the vacuum is running, only the occupied memory of the transactional context will change. The allocated memory does not change while the transaction is running.

```
level : 1; TopTransactionContext : 33587248 total in 4 blocks ; 27400 free (325 chunks ); 33559848 used
level: 1; TopTransactionContext: 33587248 total in 4 blocks; 31528 free (592 chunks); 33555720 used
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Vacuum and autovacuum process each table (including TOAST) in a separate transaction. Before each table is processed, a snapshot is created that holds the database horizon until the table and its TOAST are completed.

the process memory cluster log was discussed in Part 4 of the practice for Chapter 9. For this, the pg_log_backend_memory_contexts(PID) function is used **, where is the number of the process whose local memory needs to be output to the log. The number of the process servicing the current session can** be obtained using **the pg_backend_pid ( ) function .**

5) By default, vacuuming uses the INDEX _ CLEANUP option AUTO . This means that if the blocks with at least one dead row are less than 2% of all blocks in the table and at the same time the memory for the TID (6 bytes) of dead rows is less than 32 MB (2 ^ 25 / 6 = 5592400 rows **before** version 17), then the remaining phases are not executed and indexes are not scanned. Updating 1 million rows will exceed these limits.

Example commands :

```
postgres=# set maintenance_work_mem='3MB';
UPDATE test SET id = id - 1 where id>9000000;
vacuum verbose test;
SET
Time: 0.215 ms
UPDATE 999999
Time: 7499.271 ms (00:07.499)
INFO : clearing "postgres.public.test"
INFO: Finished cleaning "postgres.public.test": index scans: 2
pages removed: 0, remaining: 108109, scanned: 10813 (10.00% of total)
line versions: removed: 999814, remaining: 9999812, "dead" but not yet subject to removal: 0
due to clear lock conflict, row versions skipped: 185, on pages: 1
XID of deletion cutoff: 1098464, at the time of operation completion it had age: 0 XID
index scan required: 999814 dead element IDs removed from table pages (5408, 5.00% of total)
index "test_id_idx": total pages: 77011, currently deleted: 0, deleted at the moment: 0, free: 0
Average read speed: 525.782 MB/s, Average write speed: 18.311 MB/s
buffer usage: hits: 37880, misses: 143223, dirty writes: 4988
WAL usage: records: 29297, full page images: 0, bytes: 7572755
system load: CPU: user: 1.54 sec, system: 0.00 sec, elapsed: 2.12 sec
VACUUM
Time: 2128.807 ms (00:02.129)
postgres=# set maintenance_work_mem='2MB';
UPDATE test SET id = id - 1 where id>9000000;
vacuum verbose test;
SET
Time: 0.280 ms
UPDATE 999998
Time: 7492.749 ms (00:07.493)
INFO : clearing "postgres.public.test"
INFO: Finished cleaning "postgres.public.test": index scans: 3
pages removed: 0, remaining: 108109, scanned: 10813 (10.00% of total)
line versions: deleted: 999998, remaining: 9999469, "dead" but not yet subject to deletion: 0
XID of deletion cutoff: 1098465, at the time of operation completion it had age: 0 XID
index scan required: 1000183 dead element IDs removed from table pages (5409, 5.00% of total)
index "test_id_idx": total pages: 77011, currently deleted: 0, deleted at the moment: 0, free: 0
Average read speed: 642.501 MB/s, Average write speed: 16.255 MB/s
buffer usage: hits: 45705, misses: 212417, dirty writes: 5374
WAL usage: records: 29303, full page images: 0, bytes: 7576494
system load: CPU: user: 1.99 sec, system: 0.03 sec, elapsed: 2.58 sec
VACUUM
Time: 2583.433 ms (00:02.583)
postgres=# set maintenance_work_mem='1MB';
UPDATE test SET id = id - 1 where id>9000000;
vacuum verbose test;
SET
Time: 0.213 ms
UPDATE 999997
Time: 7490.491 ms (00:07.490)
INFO : clearing "postgres.public.test"
INFO: Finished cleaning "postgres.public.test": index scans: 6
pages removed: 0, remaining: 108109, scanned: 10813 (10.00% of total)
line versions: removed: 999997, remaining: 9999327, "dead" but not yet subject to removal: 0
XID of deletion cutoff: 1098466, at the time of operation completion it had age: 0 XID
999997 dead element IDs removed from table pages (5409, 5.00% of total)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
index "test_id_idx": total pages: 77011, currently deleted: 0, deleted at the moment: 0, free: 0
Average read speed: 820.235 MB/s, Average write speed: 10.130 MB/s
buffer usage: hits: 68842, misses: 420324, dirty writes: 5191
WAL usage: records: 29303, full page images: 0, bytes: 7574017
system load: CPU: user: 3.41 sec, system: 0.02 sec, elapsed: 4.00 sec
VACUUM
Time: 4003.828 ms (00:04.004)
```

Increasing **the number of scans of** one **index** from **2** to **6** increased the vacuum time by 2 times: from **2.12 s** to **4.00 s** , which is significant.

When setting up autovacuum, it is important that the number of index scans is 1, i.e. the indexes are scanned in one pass. To do this, set the value of the `autovacuum_work_mem parameter` so that the structure with the identifiers of the rows of the table that autovacuum fills in, marked for deletion, fits into the memory allocated by this parameter. Since the volume of old versions of rows marked for deletion (in **the UPDATE example) test SET id = id - 1 where id >9000000** generates 1 million row versions) usually depends on time (their rate of appearance is usually the same during working hours), then also make sure that the autovacuum cycle does not work significantly longer than usual. If the autovacuum cycle increases, then more changed rows will accumulate during the cycle. Also, the table can be locked and autovacuum will skip processing in the cycle. Therefore, commands, the lock level of which prevents autovacuum from working (an example was considered in Part 4 of this practice, error: `LOG : skipping vacuum of " test " --- lock not available`) try to give infrequently. An example of such a command: `ANALYZE test` .

6) How to calculate the number of index passes depending on the number of lines to be cleaned before version 17?

How to calculate **maintenance_ work_mem or** autovacuum_work_mem **so that there** is one pass through the indices, if the number of lines that will be cleaned up to version **17 is known ?**

The number of rows that can be deleted is populated by the processes in the column `pg_stat_all_tables.n_dead_tup` . This is the upper bound for the number of rows that can be deleted, i.e. the maximum that can be deleted .

Memory for storing row identifiers is calculated using the formula: `maintenance_ work _ mem = n _ dead _ tup` * 6. **But no more than 1 gigabyte** .

Check if the formula matches the data in the previous paragraph. The number of deleted rows is shown in the output of the vacuum command. Example:

" **dead element IDs removed: 999997** " * 6 bytes = 5999982 bytes for one pass. In the last command of the previous item " **set maintenance _ work _ mem = '1 MB '** ", which means 5999982 bytes / 1 megabyte will be required = 5999982 / 1048576 = 5.722 rounded up and we get " **index scans: 6** ".

When exceeding what number of deleted rows on version 16 there will be 2 passes through indexes regardless of the allocated memory? 179 million rows (1GB/6=178956970 rows).

In version 17, memory correlates with **n_dead_tup , but there is no exact dependence** . The required memory size is ~20 times smaller .

the test table in the databases where you created it, since the table takes up a lot of space and reduce **max_wal_size to the** default **value** :

```
postgres=# drop table test;
alter system reset max_wal_size;
checkpoint;
select pg_reload_conf();
DROP TABLE
```

8) Stop the docker container and delete the cluster directory in docker:

```
postgres @ tantor :~$ sudo docker rm -f postgres
sudo rm -rf /root/data
postgres
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Part 7. Autovacuum logging

The `log_autovacuum_min_duration` parameter sets the autovacuum duration, after which autovacuum messages will be recorded in the diagnostic log.

1) Run the commands:

```
postgres @ tantor :~$ rm - f $ PGDATA / log /*
psql -c "alter system set log_autovacuum_min_duration = 0;"
psql -c "alter system set logging_collector = off;"
psql -c "alter system set autovacuum_naptime = '1s';"
psql -c "alter system set max_wal_size = '2GB';"
psql -c "alter system set checkpoint_timeout='30min';"
psql -c "alter system set log_min_duration_statement = '15s';"
psql -c "drop table if exists t;"
psql -c "create table t (a text storage external);"
pg_ctl stop
pg_ctl start
```

```
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
DROP TABLE
CREATE TABLE
...
done
server stopped
...
done
server started
```

A table is created into which the rows will be inserted. To prevent checkpoint messages from being displayed, the checkpoint interval by time is set to 30 minutes and the checkpoint volume by WAL size is set to 2 GB .

2) Create a script for the test:

```
postgres @ tantor :~$ mcedit t . sql

select format(E'insert into t values (repeat(\'a\',2010))') from
generate_series(1, 1000) as g(id)
\ gexec
```

The size of the column value is set so that, taking into account **`storage external`** values will be TOASTed.

3) Run the test and observe the output of diagnostic messages in the terminal:

```
postgres@tantor:~$ time psql -f t.sql > /dev/null

22:16:51.736 [238943] LOG: automatic analyze of table " postgres.public.t "
I/O timings: read: 0.511 ms, write: 0.000 ms
avg read rate: 125.000 MB/s, avg write rate: 7.812 MB/s
buffer usage: 103 hits, 32 misses, 2 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:52.741 [238944] LOG: automatic analyze of table " postgres.public.t "
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 133 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:52.754 [238944] LOG: automatic vacuum of table " postgres.pg_toast.pg_toast_11129217 ": index scans: 0
pages: 0 removed, 222 remain, 222 scanned (100.00% of total)
```

```
tuples: 0 removed, 1332 remain, 0 are dead but not yet removable, oldest xmin: 22106885
removable cutoff: 22106885, which was 2 XIDs old when operation ended
new relfrozenxid: 22106218, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
I/O timings: read: 0.000 ms, write: 0.040 ms
avg read rate: 0.000 MB/s, avg write rate: 3.500 MB/s
buffer usage: 476 hits, 0 misses, 3 dirtied
WAL usage: 224 records, 3 full page images, 39250 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:53.738 [238945] LOG: automatic analyze of table " postgres.public.t "
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 134 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:54.739 [238946] LOG: automatic analyze of table " postgres.public.t "
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 136 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s

real 0m4.807s
user 0m0.031s
sys 0m0.030s

22:16:55.745 [238947] LOG: automatic analyze of table "postgres.public.t "
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 136 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:16:55.757 [238947] LOG: automatic vacuum of table " postgres.pg_toast.pg_toast_11129217 ": index scans: 0
pages: 0 removed, 400 remain, 179 scanned (44.75% of total)
tuples: 0 removed, 2400 remain, 0 are dead but not yet removable, oldest xmin: 22107423
removable cutoff: 22107423, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 391 hits, 0 misses, 0 dirtied
WAL usage: 180 records, 0 full page images, 11837 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```

TOAST cannot be analyzed) and autovacuum (of tables t and its TOAST) were launched . Autovacuum did not clean anything, since there were no old versions of the rows, but it did not freeze any rows either.

The frequency of operation of the autovacuum, autoanalysis for insertions and changes is determined by the parameters:

```
postgres=# \dconfig *scale_factor*
List of configuration parameters
Parameter | Value
-------------------------+-------
autovacuum_analyze_scale_factor | 0.1
autovacuum_vacuum_insert_scale_factor | 0.2
autovacuum_vacuum_scale_factor | 0.2
(3 rows)
```

For automatic analysis it is enough to change 10% lines. If both vacuum and analysis are to be started in the autovacuum cycle, then first the vacuum is started for each table+TOAST and only then the analysis.

4) Try to analyze the TOAST table:

```
postgres =# analyze pg_toast . pg_toast_11129217 ;
WARNING: skipping "pg_toast.pg_ toast _11129217" --- cannot analyze non-tables or special system tables
ANALYZE
```

TOAST tables are not analyzed, statistics are not needed for them.

5) Update all lines:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres @ tantor :~$ psql - c " update t set a =( repeat (' b ',2010));"
UPDATE 1200
```

```
22:28:52.815 [239757] LOG: automatic vacuum of table " postgres.public.t ": index scans: 0
pages: 0 removed, 16 remain, 16 scanned (100.00% of total)
tuples: 1200 removed , 1200 remain, 0 are dead but not yet removable, oldest xmin: 22107424
removable cutoff: 22107424, which was 0 XIDs old when operation ended
new relfrozenxid: 22107423, which is 1206 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed : 8 pages from table (50.00% of total) had 1200 dead item identifiers removed
I/O timings: read: 0.000 ms, write: 0.024 ms
avg read rate: 0.000 MB/s, avg write rate: 29.444 MB/s
buffer usage: 54 hits, 0 misses, 3 dirtied
WAL usage: 35 records, 3 full page images, 31621 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:28:52.819 [239757] LOG: automatic analyze of table "postgres.public.t"
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 129 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
22:28:52.852 [239757] LOG: automatic vacuum of table " postgres.pg_toast.pg_toast_11129217 ": index scans: 1
pages: 0 removed, 800 remain, 800 scanned (100.00% of total)
tuples: 2400 removed , 2400 remain, 0 are dead but not yet removable, oldest xmin: 22107425
removable cutoff: 22107425, which was 0 XIDs old when operation ended
new relfrozenxid: 22107423, which is 1205 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan needed: 400 pages from table (50.00% of total) had 2400 dead item identifiers removed
index "pg_toast_11129217_index": pages: 16 in total, 6 newly deleted, 6 currently deleted, 0 reusable
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 2084 hits, 0 misses, 0 dirtied
WAL usage: 1622 records, 0 full page images, 113050 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.02 s
```

One cycle of autovacuum and autoanalysis was performed. Autoanalysis does not work for TOAST tables, since TOAST statistics are not needed for the planner. TOAST index access is always used to access TOAST rows .

6) Run the commands:

```
postgres@tantor:~$
psql -c "alter system set logging_collector = on;"
psql -c "alter system reset log_autovacuum_min_duration;"
psql -c " alter system reset max_wal_size;"
psql -p 5435 -c "alter system reset max_wal_size;"
psql -p 5435 -c "checkpoint;"
psql -p 5435 -c "alter system reset max_wal_size;"
psql -p 5435 -c "checkpoint;"
psql -h 172.18.0.2 -p 5432 -U postgres "alter system reset max_wal_size;"
psql -h 172.18.0.2 -p 5432 -U postgres "checkpoint;"
psql -c "drop table if exists t;"
pg_ctl stop
sudo systemctl restart tantor-se-server-16
```

Part 8. pgststtuple extension

1) Install the pgstattuple extension :

```
postgres=# create extension if not exists pgstattuple;
NOTICE: extension "pgstattuple" already exists, skipping
CREATE EXTENSION
```

2) Using the extension, you can get the filling density of the index leaf blocks:

```
postgres=# select relname name, index_size size, leaf_pages leaf, empty_pages
empty, deleted_pages del, avg_leaf_density density from pg_class,
```

```
pgstatindex(oid) b WHERE relkind='i' and avg_leaf_density<>'NaN' and
leaf_pages>10 order by 6 limit 5;
  name | size | leaf | empty | del | density
  ---------------------------------+---------+------+-------+-----+---------
  pg_type_typname_nsp_index | 2580480 | 14 | 0 | 297 | 27.24
  pg_class_relname_nsp_index | 3547136 | 11 | 0 | 418 | 35.39
  pg_depend_reference_index | 4055040 | 14 | 0 | 477 | 45.09
  pg_depend_depender_index | 4718592 | 15 | 0 | 557 | 61.95
  pg_attribute_relid_attnum_index | 6742016 | 18 | 0 | 801 | 81.56
  (5 rows )
```

3) Using the extension, you can get an approximate percentage of lines that can be cleared:

```
  postgres=# select relname name, approx_tuple_count count, dead_tuple_count
dead, trunc(dead_tuple_percent) "dead%", approx_tuple_len len,
trunc(approx_free_percent) "free%", trunc(scanned_percent) "sample%" from
pg_class, pgstattuple_approx(oid) b WHERE relkind='r' order by 4 desc limit 5;
  name | count | dead | dead% | len | free% | sample%
  --------------+-------+------+-------+--------+-------+---------
  pg_statistic | 415 | 104 | 15 | 174085 | 14 | 70
  pg_namespace | 9 | 10 | 11 | 1076 | 73 | 100
  pg_index | 279 | 23 | 6 | 50562 | 4 | 14
  pg_depend | 2985 | 121 | 3 | 186380 | 0 | 8
  pg_type | 898 | 28 | 2 | 170024 | 2 | 9
  (5 rows)
```

4) To estimate the bloat of tables and indexes, you can use queries against existing statistics. Statistics must be collected. Example:

```
  postgres @ tantor :~$ wget https :// raw . githubusercontent . com / NikolayS /
postgres_dba / refs / heads / master / sql / b 1_table_estimation . sql

  Saving to: 'b1_table_estimation.sql'

  postgres@tantor:~$ wget
https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/b2_
btree_estimation.sql

  Saving to: 'b2_btree_estimation.sql'

  postgres@tantor:~$ psql -f b1_table_estimation.sql
  Table | Size | Extra | Bloat estimate | Live |Last Vacuum
  -----------------+-----------+------------------+------------------+-------------+-------------
  pgbench_tellers | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:02
  shmem_reference | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |12:50:27
  pgbench_branches | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:02
  t2 | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:32 (auto)
  pgbench_history | 0 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~0 bytes |12:50:31
  test2 | 592 kB | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~592 kB |12:50:31
  t1 | 8192 bytes | ~0 bytes (0.00%) | ~0 bytes (0.00%) | ~8192 bytes |13:29:02 (auto)
  t | 3264 kB | | | ~4864 kB |
  pgbench_accounts | 13 MB | ~184 kB (1.40%) | | ~126 MB |15:25:50
  (9 rows )
```

Part 9. Index processing condition: 2% of rows

1) Check how much free space is in the file system of the PGDATA directory:

```
postgres@tantor:~$ df -BM | grep sda1
Filesystem 1M-blocks Used Available Use% Mounted on
udev 1921M 0M 1921M 0% /dev
tmpfs 393M 3M 391M 1% /run
/dev/sda1 47190M 31412M 13349M 71% /
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
tmpfs 1964M 4M 1960M 1% /dev/shm
tmpfs 5M 0M 5M 0% /run/lock
tmpfs 393M 1M 393M 1% /run/user/102
tmpfs 393M 1M 393M 1% /run/user/1000
```

If less than 3400MB, then the following steps do not need to be completed.
Space may be taken up by WAL directories and large tables.

2) Turn on time measurement. Time is useful for estimating the execution time of commands.

```
postgres=# \timing on
Timing is on.
```

If the number of blocks with at least one dead row is less than 2% of all blocks in the table and at the same time the memory for the TID (6 bytes) of dead rows is less than 32 MB (2 ^ 25 / 6 = 5592400 rows before version 17), then the indexes are not scanned.
That is, a large table is 5592400/0.02 = 279620000 rows.
In order for less than 2% of rows in a table to be updated and for vacuum to scan indexes (due to the fact that the TID of dead rows will exceed 32 MB), the table must have more than 279620000 rows.
How much space will such a table take? The maximum number of rows in a table is 226, a table with one column of type int4 or int8 will take 10135609344 bytes = 9.44 GB.
We will not test this.

3) Create a table with 10 million rows.

```
postgres=# drop table if exists test;
create table test with (autovacuum_enabled = off) as select * from
generate_series(1, 10000000) x(id);
CREATE INDEX ON test(id);

DROP TABLE
Time: 362.244 ms
SELECT 10000000
Time: 14903.225 ms (00:14.903)
CREATE INDEX
Time: 13010.045 ms (00:13.010)
```

4) Update 2% of rows :

```
postgres=# UPDATE test SET id = id - 1 where id<=200000;

UPDATE 200000
Time: 1352.337 ms (00:01.352)
```

5) See how much space the table and index take up:

```
postgres=# select pg_total_relation_size('test');
select pg_size_pretty(pg_total_relation_size('test'));

pg_total_relation_size
------------------------
599302144
(1 row)

pg_size_pretty
----------------
572 MB
(1 row)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

They occupy `572 MB` , which is equal to `599302144` bytes.

6) Look at the share and percentage of dead lines:

```
postgres=# select relname, n_dead_tup, n_live_tup,
round(100.0*n_dead_tup/n_live_tup, 2) pct1,
round(100.0*n_dead_tup/(n_live_tup+n_dead_tup),2) pct2 from pg_stat_user_tables
where relname = 'test';

relname | n_dead_tup | n_live_tup | pct1 | pct2
---------+------------+------------+------+------
test | 200000 | 10000000 | 2.00 | 1.96
(1 row)
```

Which of the "percentages" does the vacuum and autovacuum use to decide whether to process indexes? The answer will follow.

7) Look at how the pgstattuple function calculates the percentage and how long it takes them to calculate:

```
postgres=# select * from pgstattuple_approx('test')\gx
-[ RECORD 1 ]--------+--------------------
table_len | 370049024
scanned_percent | 100
approx_tuple_count | 10000000
approx_tuple_len | 280000000
approx_tuple_percent | 75.66565018152838
dead_tuple_count | 200000
dead_tuple_len | 5600000
dead_tuple_percent | 1.5133130036305675
approx_free_space | 862236
approx_free_percent | 0.23300588410685824

Time: 778.598 ms


postgres=# select * from pgstattuple('test')\gx
-[ RECORD 1 ]------+----------
table_len | 370049024
tuple_count | 10000000
tuple_len | 280000000
tuple_percent | 75.67
dead_tuple_count | 200000
dead_tuple_len | 5600000
dead_tuple_percent | 1.51
free_space | 543684
free_percent | 0.15

Time: 1857.216 ms (00:01.857)
```

The difference in execution time of functions is small.
dead_tuple_percent = 1.51% functions are calculated from the block size (files).

8) Perform vacuuming:

```
postgres =# vacuum (verbose) test;

INFO: vacuuming "postgres.public.test"
INFO: finished vacuuming "postgres.public.test": index scans: 0
pages: 0 removed, 45172 remain, 45172 scanned (100.00% of total)
```

```
tuples: 200000 removed, 10000000 remain, 0 are dead but not yet removable, oldest xmin: 22113906
removable cutoff: 22113906, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan bypassed: 885 pages from table ( 1.96% of total ) have 200000 dead item identifiers
I/O timings: read: 190.718 ms, write: 246.465 ms
avg read rate: 26.787 MB/s, avg write rate: 34.300 MB/s
buffer usage: 59491 hits, 30026 misses, 38448 dirtied
WAL usage: 45137 records, 44255 full page images, 365378454 bytes
system usage: CPU: user: 1.43 s, system: 1.08 s, elapsed: 8.75 s
VACUUM
Time: 8757.911 ms (00:08.758)
```

Vacuum considers percent dead lines By formula : `n_dead_tup/(n_live_tup+n_dead_tup)`

9) 5592400 dead lines will take exactly 32MB. Let's check that in addition to the TID array, `maintenance_work_mem` also limits memory for other structures.
Set maintenance_work_mem = '32MB' and update 5592200 rows so that the TID array is just under 32MB and run vacuum:

```
postgres=# set maintenance_work_mem = '32MB';
UPDATE test SET id = id - 1 where id<= 5592 2 00 ;
vacuum (verbose) test;
```

```
SET
UPDATE 5592200
Time: 41250.795 ms (00:41.251)
INFO: vacuuming "postgres.public.test"
INFO: finished vacuuming "postgres.public.test": index scans: 2
pages: 0 removed, 69877 remain, 50376 scanned (72.09% of total)
tuples: 5592200 removed, 9909829 remain, 0 are dead but not yet removable, oldest xmin: 22113907
removable cutoff: 22113907, which was 0 XIDs old when operation ended
frozen: 24748 pages from table (35.42% of total) had 5592548 tuples frozen
index scan needed: 25630 pages from table ( 36.68% of total ) had 5792200 dead item identifiers
removed
index "test_id_idx": pages: 43304 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 933.086 ms, write: 762.870 ms
avg read rate: 32.593 MB/s, avg write rate: 23.729 MB/s
buffer usage: 76928 hits, 136100 misses, 99088 dirtied
WAL usage: 204558 records, 102164 full page images, 525140763 bytes
system usage: CPU: user: 6.67 s, system: 3.78 s, elapsed: 32.62 s
VACUUM
Time: 32625.933 ms (00:32.626)
```

2 passes through the indices were performed, **`the maintenance _ work _ mem memory`** was not enough for one pass.

10) When installing with a small (1MB) memory reserve , it will be enough and one pass will be performed :

```
postgres=# set maintenance_work_mem = ' 33MB ';
UPDATE test SET id = id - 1 where id<=5592 5 00;
vacuum (verbose) test;
```

```
SET
Time: 0.166 ms
UPDATE 5592500
Time: 43902.661 ms (00:43.903)
INFO: vacuuming "postgres.public.test"
INFO: finished vacuuming "postgres.public.test": index scans: 1
pages: 0 removed, 69877 remain, 49494 scanned (70.83% of total)
tuples: 5592500 removed, 8483454 remain, 0 are dead but not yet removable, oldest xmin: 22113908
removable cutoff: 22113908, which was 0 XIDs old when operation ended
frozen: 21270 pages from table (30.44% of total) had 4806279 tuples frozen
index scan needed: 24749 pages from table (35.42% of total) had 5592500 dead item identifiers
removed
index "test_id_idx": pages: 43304 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 642.541 ms, write: 746.983 ms
avg read rate: 25.280 MB/s, avg write rate: 23.980 MB/s
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
buffer usage: 67906 hits, 99172 misses, 94072 dirtied
WAL usage: 196037 records, 91225 full page images, 517619377 bytes
system usage: CPU: user: 5.36 s, system: 3.31 s, elapsed: 30.64 s
VACUUM
Time: 30649.388 ms (00:30.649)
```

10) Delete the table so that it doesn't take up disk space:

```
postgres=# drop table test;
DROP TABLE
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Practice for Chapter 13

Part 1. Reading vacuum and autovacuum messages

1) Look at the value and description of the parameter `log _ autovacuum _ min _ duration` :

```
postgres=# select name, setting, unit, context, min_val, short_desc, extra_desc
from pg_settings where name ~ 'um_mi'\gx
-[ RECORD 1 ]------------------------------------------------------
name | log_autovacuum_min_duration
setting | 0
unit | ms
context | sighup
min_val | -1
short_desc | Sets the minimum execution time above which autovacuum actions will be logged.
extra_desc | Zero prints all actions . -1 turns autovacuum logging off.
```

The default value is 10 minutes, this is a reasonable value. In practice, the value was previously set to 0, which means logging all autovacuum actions .

`log_autovacuum_min_duration` according to the table or TOAST table .

If such messages appear, it is worth finding out the reason for the long vacuuming of the table.

The message is written to the log after the table and its indexes have been processed; before that, you can monitor the progress of the vacuuming through the `pg_stat_progress_vacuum view` , and the analysis through `pg_stat_progress_analyze` .

2) Run the command or look at any example of the autovacuum log or vacuum commands:

```
postgres=# vacuum verbose pg_class;
INFO: vacuuming " postgres.pg_catalog.pg_class "
INFO: launched 2 parallel vacuum workers for index vacuuming (planned: 2)
INFO: finished vacuuming "postgres.pg_catalog.pg_class": index scans: 1
pages: 0 removed, 29 remain, 29 scanned (100.00% of total)
tuples: 6 removed, 759 remain, 0 are dead but not yet removable , oldest xmin:
22113909
removable cutoff: 22113909, which was 0 XIDs old when operation ended
new relfrozenxid: 21894381, which is 3 XIDs ahead of previous value
frozen: 4 pages from table (13.79% of total) had 46 tuples frozen
index scan needed : 21 pages from table ( 72.41% of total ) had 143 dead item
identifiers removed
index "pg_class_oid_index": pages: 151 in total, 0 newly deleted, 145 currently
deleted, 145 reusable
index "pg_class_relname_nsp_index": pages: 433 in total, 0 newly deleted, 418
currently deleted, 418 reusable
index "pg_class_tblspc_relfilinode_index": pages: 151 in total, 0 newly
deleted, 144 currently deleted, 144 reusable
I/O timings: read: 37.648 ms, write: 0.088 ms
avg read rate: 94.197 MB/s, avg write rate: 6.524 MB/s
buffer usage: 707 hits, 592 misses, 41 dirtied
WAL usage: 85 records, 44 full page images, 273017 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.04 s
VACUUM
```

The result of the command is similar to that issued by autovacuum in the log.

Let's see what to pay attention to first.

`index scans : 1` number of passes through indexes. 1 - ideal case, one pass. 0 - no indexes or few changes and they were not scanned. The reason for not scanning is in the message " `72.41% of total` " lines " `index scan needed` ". How to estimate and adjust the number of scans is described in Part 9 of the previous practice.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

launched 2 parallel vacuum workers means that the indexes were processed by two parallel processes and the one that vacuums. The degree of parallelism is usually calculated correctly.

elapsed : 0.04 s time to vacuum table postgres . pg_catalog . pg_class . user : 0.00 s and system : 0.00 s should not deviate much from the 60/40 ratio. The sum of user + system may be less than elapsed . This means that vacuum encountered locks when accessing shared pool buffers.

tuples: 6 removed, 759 remain, 0 are dead but not yet removable - efficiency this cycle vacuum . If the number 6 is removed small, then the vacuum worked in vain and you need to look 0 are dead but not yet removable . If it is large, then the base horizon was held. It is worth assessing what holds the horizon and if it is possible to reduce the holding duration, only then adjust the autovacuum.

frozen : 4 pages from table (13.79% of total) indicates that there are no dead lines left on some blocks after vacuuming, all lines are current, and it was possible to freeze all lines in this number of blocks. These values do not need to be paid attention to, they depend on the horizon retention by snapshots.

WAL usage : shows how many log records were created, the size at the end of the line is 273017 bytes . The volume is most affected by writing full page images 44 full page images .

avg values read rate : 94.197 MB / s , avg write rate : 6.524 MB / s nothing can be said about the work of the input-output, since these values are influenced to a greater extent by the speed of the processor core, and not by the input-output.

buffer usage : vacuum, like all instance processes, works only through the buffer cache. Using the buffer cache makes the processes more efficient. Autovacuum uses a VACUUM-type buffer ring, in which dirty pages are not removed from the ring, but sent for writing. The ring size is specified by the vacuum_buffer_usage_limit configuration parameter . The default is 256 KB, the range is from 128 KB to 16 MB. In the VACUUM command (BUFFER_USAGE_LIMIT 0), you can set the value 0, which means that the use of the buffer ring is prohibited, and the blocks of all objects processed by the command, both during cleaning and during analysis, can occupy all buffers. This will speed up the vacuuming process and, if the buffer cache is large, will load the processed blocks into it. The value 0 cannot be set for autovacuum, but if autovacuum is launched to protect against transaction counter overflow, the buffer ring is not used and autovacuum is performed in an aggressive mode.

Part 2. Reading checkpoint messages

1) This step is optional. To create checkpoint records, perform the following steps:

In the first terminal, log files are deleted to make it easier to find the last file. The checkpoint frequency is set to 2 minutes to avoid long waits. The instance is restarted. A small load is started:

```
postgres @ tantor :~$ rm - f $ PGDATA / log / postgresql *
psql -c "alter system set checkpoint_timeout=120;"
sudo systemctl restart tantor-se-server-16
while:; do pgbench -i; sleep 10; done
```

In the second terminal, the following commands are run:

```
postgres@tantor:~$ sleep 300
time psql -c "drop table if exists test; create table test with
(autovacuum_enabled = off) as select * from generate_series(1, 10000000) x(id);
drop table if exists test;"
sleep 260
psql -c "alter system reset checkpoint_timeout;"
psql -c "select pg_reload_conf ();"
cat $PGDATA/log/postgresql*

real 0m14.942s
user 0m0.014s
sys 0m0.008s
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
12:00:04.950 MSK [331789] LOG: starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
12:00:04.951 MSK [331789] LOG: listening on IPv4 address "127.0.0.1", port 5432
12:00:04.951 MSK [331789] LOG: listening on IPv6 address "::1", port 5432
12:00:04.959 MSK [331789] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
12:00:04.980 MSK [331793] LOG: database system was shut down at 12:00:04 MSK
12:00:04.993 MSK [331789] LOG: database system is ready to accept connections
12:02:05.079 MSK [331791] LOG: checkpoint starting: time
12:03:53.791 MSK [331791] LOG: checkpoint complete: wrote 174 buffers (1.1%); 0 WAL file(s) added, 10 removed, 0 recycled; write=107.775 s, sync=0.630 s, total=108.712 s; sync files=293, longest=0.009 s, average=0.003 s; distance=154031 kB, estimate=154031 kB; lsn= 248/63E6F100 , redo lsn=248/5C112AF0
12:04:05.804 MSK [331791] LOG: checkpoint starting: time
12:05:15.109 MSK [331791] LOG: checkpoint complete: wrote 145 b uffers (0.9%); 0 WAL file(s) added, 8 removed, 0 recycled; write=67.819 s, sync=1.343 s, total=69.306 s; sync files=133, longest=0.928 s, average=0.011 s; distance=141390 kB, estimate=152767 kB; lsn=248/864CD020, redo lsn=248/64B26448
12:05:15.109 MSK [331791] LOG: checkpoint starting: wal
12:07:03.773 MSK [331791] LOG: checkpoint complete: wrote 117 buffers (0.7%); 0 WAL file(s) added, 0 removed, 34 recycled; write=107.923 s, sync=0.365 s, total=108.664 s; sync files=152, longest=0.014 s, average=0.003 s; distance=557039 kB, estimate=557039 kB; lsn=248/9854BCB8, redo lsn=248/86B22090
12:07:15.783 MSK [331791] LOG: checkpoint starting: time
12:09:03.659 MSK [331791] LOG: checkpoint complete: wrote 2140 buffers (13.1%); 0 WAL file(s) added, 0 removed, 19 recycled; write=107.211 s, sync=0.430 s, total=107.876 s; sync files=179, longest=0.009 s, average=0.003 s; distance=301854 kB, estimate=531520 kB; lsn=248/A1BE70A8, redo lsn=248/991E9908
12:09:15.667 MSK [331791] LOG: checkpoint starting: time
12:11:03.524 MSK [331791] LOG: checkpoint complete: wrote 156 buffers (1.0%); 0 WAL file(s) added, 0 removed, 9 recycled; write=107.323 s, sync=0.394 s, total=107.857 s; sync files=168, longest=0.016 s, average=0.003 s; distance=154209 kB, estimate=493789 kB; lsn=248/AA5D0EA0, redo lsn=248/A2882090
12:11:15.537 MSK [331791] LOG: checkpoint starting: time
12:13:03.507 MSK [331791] LOG: checkpoint complete: wrote 125 buffers (0.8%); 0 WAL file(s) added, 0 removed, 9 recycled; write=107.521 s, sync=0.312 s, total=107.971 s; sync files=153, longest=0.009 s, average=0.003 s; distance=141196 kB, estimate=458530 kB; lsn=248/B23578D8, redo lsn=248/AB265368
12:13:15.529 MSK [331791] LOG: checkpoint starting: time
12:15:03.241 MSK [331791] LOG: checkpoint complete: wrote 1792 buffers (10.9%); 0 WAL file(s) added, 0 removed, 7 recycled; write=107.491 s, sync=0.107 s, total=107.712 s; sync files=45, longest=0.011 s, average=0.003 s; distance=115797 kB, estimate=424256 kB; lsn=248/B237A898, redo lsn=248/B237A808
12:15:38.289 MSK [331789] LOG: received SIGHUP, reloading configuration files
12:15:38.290 MSK [331789] LOG: parameter "checkpoint_timeout" removed from configuration file, reset to default
12:20:15.392 MSK [331791] LOG: checkpoint starting: time
12:23:25.949 MSK [331791] LOG: checkpoint complete: wrote 163 buffers (1.0%); 0 WAL file(s) added, 0 removed, 19 recycled; write=189.294 s, sync=1.032 s, total=190.557 s; sync files=501, longest=0.009 s, average=0.003 s; distance=320376 kB, estimate=413868 kB; lsn=248/D24C4778, redo lsn=248/C5C589C0
12:25:16.016 MSK [331791] LOG: checkpoint starting: time
12:28:25.985 MSK [331791] LOG: checkpoint complete: wrote 1813 buffers (11.1%); 0 WAL file(s) added, 1 removed, 12 recycled; write=189.692 s, sync=0.113 s, total=189.970 s; sync files=51, longest=0.010 s, average=0.003 s; distance=205385 kB, estimate=393020 kB; lsn=248/D24EB1C0, redo lsn=248/D24EB178
12:41:21.274 MSK [331791] LOG: checkpoint starting: immediate force wait
12:41:21.322 MSK [331791] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.049 s; sync files=0, longest=0.000 s, average=0.000 s; distance=0 kB, estimate=353718 kB; lsn=248/D24EB2E8, redo lsn=248/D24EB2A0
```

After 520 seconds, the contents of the log are displayed in the second terminal, and you can interrupt the <ctrl+c> cycle that creates the load in the first terminal.

Description contents magazine :

```
12:00:04.950 MSK [331789] LOG: starting Tantor Special Edition 16.2.0
12:00:04.951 MSK [331789] LOG: listening on IPv4 address "127.0.0.1", port 5432
12:00:04.951 MSK [331789] LOG: listening on IPv6 address "::1", port 5432
12:00:04.959 MSK [331789] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
12:00:04.980 MSK [331793] LOG: database system was shut down at 12:00:04 MSK
12:00:04.993 MSK [331789] LOG: database system is ready to accept connections
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The instance started at `12:00:04` . The checkpoint started two minutes later (via checkpoint_timeout) at `12:02:05` :

```
12:02:05.079 MSK [331791] LOG : checkpoint starting : time
```

of the interval, according to checkpoint_complete_target = 0.9. Calculation time : 120*0.9=108 seconds . `12:02:05` + 108= `12:03:53`

```
12:03:53.791 MSK [331791] LOG: checkpoint complete: wrote 174 buffers (1.1%); 0 WAL file(s) added, 10 removed, 0 recycled; write=107.775 s, sync=0.630 s, total=108.712 s; sync files=293, longest=0.009 s, average=0.003 s; distance=154031 kB, estimate=154031 kB; lsn=248/63E6F100, redo lsn=248/5C112AF0
```

Exactly two minutes after the start of the first checkpoint, the next checkpoint began at `12:04:05` :

```
12:04:05.804 MSK [331791] LOG : checkpoint starting : time
```

About 300 seconds after the instance was started, a command was run that created more than 600MB of log records and ran for `14.942 s` . As soon as more than half of the max_wal_size log records had been accumulated, the time checkpoint began to complete and completed in `sync = 1.343 s.` having worked `total =69.306 s` :

```
12:05:15.109 MSK [331791] LOG: checkpoint complete: wrote 145 buffers (0.9%); 0 WAL file(s) added, 8 removed, 0 recycled; write=67.819 s, sync=1.343 s, total=69.306 s; sync files=133, longest=0.928 s, average=0.011 s; distance=141390 kB, estimate=152767 kB; lsn=248/864CD020, redo lsn=248/64B26448
```

A record about the start of the checkpoint by size was immediately created:

```
12:05:15.109 MSK [331791] LOG : checkpoint starting : wal
```

Since the volume of log records created by the command that worked for 14 seconds did not exceed max_wal_size=1Gb, the checkpoint did not complete at maximum speed and worked for `108.664 s` . If the volume had exceeded 1Gb, then a checkpoint of type `wal` would have completed a little faster, the next checkpoint of type `wal would have started` and warning messages would have been written to the log immediately after the second checkpoint in size completed, which would have completed at maximum speed in 16 seconds :.

```
LOG: checkpoint complete: wrote 28 buffers (0.2%); 0 WAL file(s) added, 0 removed, 33 recycled; write=14.989 s, sync=0.136 s, total= 15.966 s ; sync files=20, longest=0.083 s, average=0.007 s;
LOG: checkpoints are occurring too frequently (16 seconds apart)
HINT: Consider increasing the configuration parameter "max_wal_size" .
```

But in the example the size checkpoint completed in `108.664 s` **and wrote** `295869480 bytes` to WAL :

```
select ' 248/9854 BCB 8 ':: text :: pg_lsn - '248/86 B 22090 ' :: text :: pg_lsn written ;
  written
-----------
295869480
(1 row)
```

```
12:07:03.773 MSK [331791] LOG: checkpoint complete: wrote 117 buffers (0.7%); 0 WAL file(s) added, 0 removed, 34 recycled; write=107.923 s, sync=0.365 s, total=108.664 s; sync files=152, longest=0.014 s, average=0.003 s; distance= 557039 kB, estimate=557039 kB; lsn= 248/9854BCB8 , redo lsn= 248/86B22090
```

Time checkpoints started 2 minutes (via checkpoint_timeout) after the start of the `12:05:15` size checkpoint:

```
12:07:15.783 MSK [331791] LOG: checkpoint starting: time
12:09:03.659 MSK [331791] LOG: checkpoint complete: wrote 2140 buffers (13.1%); 0 WAL file(s) added, 0 removed, 19 recycled; write=107.211 s, sync=0.430 s, total=107.876 s; sync files=179, longest=0.009 s, average=0.003 s; distance=301854 kB, estimate=531520 kB; lsn=248/A1BE70A8, redo lsn=248/991E9908
```

Time checkpoints start checkpoint_timeout after the start of the previous checkpoint:

```
12:09:15.667 MSK [331791] LOG: checkpoint starting: time
```

```
12:11:03.524 MSK [331791] LOG: checkpoint complete: wrote 156 buffers (1.0%); 0 WAL file(s) added, 0 removed, 9 recycled; write=107.323 s, sync=0.394 s, total=107.857 s; sync files=168, longest=0.016 s, average=0.003 s; distance=154209 kB, estimate=493789 kB; lsn=248/AA5D0EA0, redo lsn=248/A2882090
12:11:15.537 MSK [331791] LOG: checkpoint starting: time
12:13:03.507 MSK [331791] LOG: checkpoint complete: wrote 125 buffers (0.8%); 0 WAL file(s) added, 0 removed, 9 recycled; write=107.521 s, sync=0.312 s, total=107.971 s; sync files=153, longest=0.009 s, average=0.003 s; distance=141196 kB, estimate=458530 kB; lsn=248/B23578D8, redo lsn=248/AB265368
12:13:15.529 MSK [331791] LOG: checkpoint starting: time
12:15:03.241 MSK [331791] LOG: checkpoint complete: wrote 1792 buffers (10.9%); 0 WAL file(s) added, 0 removed, 7 recycled; write=107.491 s, sync=0.107 s, total=107.712 s; sync files=45, longest=0.011 s, average=0.003 s; distance=115797 kB, estimate=424256 kB; lsn=248/B237A898, redo lsn=248/B237A808
12:15:38.289 MSK [331789] LOG: received SIGHUP, reloading configuration files
12:15:38.290 MSK [331789] LOG: parameter "checkpoint_timeout" removed from configuration file, reset to default
```

2) The checkpoint can be called by the checkpoint command . In this case, the log will contain the following message:

```
13:10:36.200 MSK [331791] LOG: checkpoint starting: time
13:12:38.670 MSK [331791] LOG: checkpoint complete: wrote 137 buffers (0.8%); 0 WAL file(s) added, 0 removed, 24 recycled; write=121.546 s, sync=0.645 s, total=122.470 s; sync files=301, longest=0.009 s, average=0.003 s; distance=402997 kB, estimate=530375 kB; lsn=249/5740D278, redo lsn=249/4DD94270
13:12:38 .670 MSK [331791] LOG: checkpoint starting: immediate force wait
13:12:39.004 MSK [331791] LOG: checkpoint complete: wrote 1790 buffers (10.9%); 0 WAL file(s) added, 0 removed, 10 recycled; write=0.077 s, sync=0.121 s, total=0.334 s; sync files=46, longest=0.028 s, average=0.003 s; distance=154084 kB, estimate=492746 kB; lsn=249/5740D358 , redo lsn=249/5740D310
13:17:38 .106 MSK [331791] LOG: checkpoint starting: time
13:20:48.009 MSK [331791] LOG: checkpoint complete: wrote 178 buffers (1.1%); 0 WAL file(s) added, 1 removed, 21 recycled ; write=188.370 s, sync=1.278 s, total=189.904 s; sync files=560, longest=0.042 s, average=0.003 s ; distance=358909 kB, estimate=479362 kB; lsn=249/7C088840 , redo lsn=249/6D28C9A0
```

The checkpoint will start `at 13:17:38` checkpoint_timeout after the start of the previous checkpoint `at 13:12:38` .

3) An example of the last message from the previous point, if they were issued in Russian:

```
MESSAGE: checkpoint completed: buffers written: 178 (1.1%); WAL files added : 0, deleted: 1, recycled: 21 ; write=188.370 sec, sync=1.278 sec, total=189.904 sec; files_synced=560, longest_sync=0.042 sec, avg=0.003 sec ; distance=358909 kB , expected=479362 kB
```

Description of values:

`wrote 178 buffers (1.1%); written buffers` - the number of dirty buffers written by checkpoint. Along with checkpointer, dirty blocks can be written by server processes and bgwriter. In brackets is the percentage of the total number of buffers in the buffer cache, specified by the shared_buffers parameter.
`0 WAL file(s) added, files added` - number of WAL files created
`1 removed, removed` - number of removed WAL files
`21 recycled; recycled` - the number of WAL files reused
`write=188.370 s, write` - total duration of commands to write to the Linux page cache
`sync=1.278 s, sync` - total duration of f data sync calls on WAL files
`total=189.904 s; total` - sum `write+sync`
`sync files=560, synced_files` - the number of processed data files (files in tablespaces and their blocks in the buffer cache)
`longest=0.042 s, longest_sync` - the longest duration of fsync processing of one file
`average =0.003 s ; average` - average speed of fsync execution for all synchronized files

distance =358909 kB , distance is the volume of WAL records between the beginning of the previous checkpoint and the beginning of the current one to which the record belongs

estimate =479362 kB ; expected - calculated to estimate how many WAL segments will be used at the next checkpoint. Formula calculation :

```
Distance = nbytes;
if (Estimate < nbytes)
Estimate = nbytes;
else
CheckPointDistanceEstimate =
(0.90 * CheckPointDistanceEstimate + 0.10 * (double) nbytes);
```

Comment To formula :

```
/*
* To estimate the number of segments consumed between checkpoints, keep a
* moving average of the amount of WAL generated in previous checkpoint
*cycles. However, if the load is bursty, with quiet periods and busy
* periods, we want to cater for the peak load. So instead of a plain
* moving average, let the average decline slowly if the previous cycle
* used less WAL than estimated, but bump it up immediately if it used
* more.
*
* When checkpoints are triggered by max_wal_size, this should converge to
* CheckpointSegments * wal_segment_size,
*
* Note: This doesn't pay any attention to what caused the checkpoint.
* Checkpoints triggered manually with CHECKPOINT command, or by eg
* starting a base backup, are counted the same as those created
* automatically. The slow-decline will largely mask them out, if they are
*not frequent. If they are frequent, it seems reasonable to count them
* in as any others; if you issue a manual checkpoint every 5 minutes and
* never let a timed checkpoint happen, it makes sense to base the
* preallocation on that 5 minute interval rather than whatever
* checkpoint_timeout is set to.
 */
```

Consequence of the formula: if `max_wal_size is decreased` , then the number of WAL files decreases gradually: after ~10 checkpoints (consequence `0.90 * previous estimate`).

`lsn =249/7 C 088840,` - LSN of the checkpoint end record. If this is the last checkpoint, this value is present in the control file.

`redo lsn=249/6 D 28 C 9 A 0` - LSN of the record from which the checkpoint began. This record is not generated by the checkpointer, but by any process.

Part 3. Reading checkpoint messages with `pg_waldump utility`

1) The control file contains LSN records about the end and beginning of the last checkpoint. Run command :

```
postgres@tantor:~$ pg_controldata | grep check | head -n 3


Latest checkpoint location: 7/F7 F28E08
Latest checkpoint's REDO location: 7/F1BF0018
Latest checkpoint's REDO WAL file: 0000000100000007000000 F1
```

2) Look at what is contained in the journal entries. Display the entries that contain the letter combination **CHECKPOINT** :

```
postgres@tantor:~$ pg_waldump -s 7/F1BF0018 -e 7/F7 FFFFFF | grep CHECKPOINT

rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 7/F1373F70, prev 7/F1373F20, desc:
CHECKPOINT_ONLINE redo 7/EBA5F1A8; tli 1; prev tli 1; fpw true; xid 7140020; oid 33402; multi 1;
```

```
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7140019; online
 rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 7/F7F28E08 , prev 7/F7F28DB8, desc:
CHECKPOINT_ONLINE redo 7/F1BF0018 ; tli 1; prev tli 1; fpw true; xid 7325493; oid 33402; multi 1;
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7325493; online
```

The -e parameter can be omitted, the result will be the same:

```
pg_waldump -s 7/F1BF0018 | grep CHECKPOINT
```

The digits of the numbers that refer to the offset within the WAL file can be replaced with zeros, the result will be the same:

```
pg_waldump -s 7/F1 000000| grep CHECKPOINT
```

3) LSN in the log will match lsn in the log. Look at the messages in the diagnostic log:

```
postgres@tantor:~$ tail -n 4 $PGDATA/log/postgresql* | grep checkpoint

LOG: checkpoint complete: wrote 3031 buffers (18.5%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.899 s, sync=0.008 s, total=269.926 s; sync files=13, longest=0.003 s, average=0.001 s;
distance=100908 kB, estimate=101792 kB; lsn=7/EB059E10, redo lsn=7/E5712C48
LOG: checkpoint starting: time
LOG: disconnection: session time: 0:28:19.585 user=postgres database=postgres host=[local]
LOG: checkpoint complete: wrote 3215 buffers (19.6%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.992 s, sync=0.014 s, total=270.027 s; sync files=14, longest=0.009 s, average=0.001 s;
distance=101681 kB, estimate=101781 kB; lsn=7/F1373F70, redo lsn=7/EBA5F1A8
LOG: checkpoint starting: time
LOG: checkpoint complete: wrote 3018 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6 recycled;
write=269.877 s, sync=0.007 s, total=269.901 s; sync files=13, longest=0.003 s, average=0.001 s;
distance=99907 kB, estimate=101593 kB; lsn=7/F7F28E08 , redo lsn=7/F1BF0018
LOG: checkpoint starting: time
```

4) Perform Not need . Look example :

```
pg_controldata | grep check | head -n 3
Latest checkpoint location: 8/AA004C8
Latest checkpoint's REDO location: 8/5177990
Latest checkpoint's REDO WAL file: 000000010000000800000 0 5
```

Because leading zeros are not printed in LSN 8/ 0 5177990 , it is possible to get the following error:

```
pg_waldump -s 8/5 1000000 | grep CHECKPOINT
pg_waldump: error: could not find file "000000010000000800000 5 1": No such file or directory
```

because in reality the LSN value is:

```
pg_waldump -s 8/ 0 5 000000 | grep CHECKPOINT
 rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 8/0AA004C8, prev 8/0AA00478, desc:
CHECKPOINT_ONLINE redo 8/5177990; tli 1; prev tli 1; fpw true; xid 7880789; oid 33402; multi 1;
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7880788; online
 pg_waldump: error: error in WAL record at 8/F9D83B8: invalid record length at 8/F9D83E0: expected
at least 26, got 0
```

When specifying the full LSN, the command passes successfully:

```
pg_waldump -s 8/5177990 | grep CHECKPOINT
 rmgr: XLOG len (rec/tot): 148/148, tx: 0, lsn: 8/ 0 AA004C8 , prev 8/ 0 AA00478 , desc:
CHECKPOINT_ONLINE redo 8/5177990 ; tli 1; prev tli 1; fpw true; xid 7880789; oid 33402; multi 1;
offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0;
oldest running xid 7880788; online
```

In the output of **pg_waldump in the** lines with **lsn** and **prev** leading zeros **are saved,** and in the **redo field** are not saved , which creates confusion.

Leading zeros **are not preserved** in the diagnostic log and output of the pg_controldata utility .

5) When using 16MB WAL files, the last two characters of the file name are the same as the two characters if you add a leading zero. Example execution commands :

pg_controldata | grep check | head -n 3

```
Latest checkpoint location: 7/F7F28E08
Latest checkpoint's REDO location: 7/ F1 BF0018
Latest checkpoint's REDO WAL file: 0000000100000007000000 F1
Latest checkpoint location: 8/1164B2E8
Latest checkpoint's REDO location: 8/ 0 B C98978
Latest checkpoint's REDO WAL file: 0000000100000008000000 0B
Latest checkpoint location: 7/FE08BAC0
Latest checkpoint's REDO location: 7/ F8 78C390
Latest checkpoint's REDO WAL file: 0000000100000007000000 F8
Latest checkpoint location: 8/ 0 497B1F0
Latest checkpoint's REDO location: 7/ FF 054A78
Latest checkpoint's REDO WAL file: 0000000100000007000000 FF
Latest checkpoint location: 8/ 0 AA004C8
Latest checkpoint's REDO location: 8/ 05 177990
Latest checkpoint's REDO WAL file: 0000000100000008000000 05
```

pg_controldata does not output the zeros marked in red . *The zeros* are added for clarity.

How to visually understand whether a zero is needed or not? The number of letters after the slash should be 8. The same as before the slash, but before the slash, the absence of zeros does not create ambiguity. Example:

8/ **B** C 98978 should be *0000000* 8/ ***0*** **B** C 98978 .

6) Look at the example of a dump of three consecutive journal entries:

```
pg_waldump -s 8/1164B298 -e 8/1164B3E8
 rmgr: Standby len (rec/tot): 76 / 76 , tx: 0, lsn: 8/1164B298 , prev 8/1164B240, desc:
RUNNING_XACTS nextXid 8232887 latestCompletedXid 8232885 oldestRunningXid 8232886; 1
xacts: 8232886

 rmgr: XLOG len (rec/tot): 148 / 148 , tx: 0, lsn: 8/1164B2E8 , prev 8/1164B298 , desc:
CHECKPOINT_ONLINE redo 8/BC98978; tli 1; prev tli 1; fpw true; xid 8064948; oid 33402;
multi 1; offset 0; oldest xid 723 in DB 1;
 oldest multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid
8064947; online

 rmgr: Heap len (rec/tot): 86 / 86 , tx: 8232886, lsn: 8/1164B380 , prev 8/1164B2E8 ,
desc: HOT_UPDATE old_xmax: 8232886, old_off: 113, old_infobits: [], flags: 0x20,
new_xmax: 0, new_off: 114, blkref #0: rel 1663/5/25198 blk 1
```

prev points to the LSN of the start of the previous log record. Using this field, you can move backwards through the WAL.

In the example, before the record about the end of the checkpoint, there are records of 72-80 bytes in length.

The LSN of the next log record is not stored because the length of the record in len (rec / tot) is stored . lsn + len + padding to 8 bytes = LSN of the start of the next record.

```
select pg_wal_lsn_diff(' 8/1164B2E8 ',' 8/1164B298 ');
pg_wal_lsn_diff
-----------------
              80
(1 row)
```

The record length is 80 instead of len = 76 due to padding (padding up to 8 bytes).

```
select pg_wal_lsn_diff(' 8/1164B380 ', ' 8/1164B2E8 ');
pg_wal_lsn_diff
-----------------
             152
(1 row)
```

The record length is 152 , not len = 148 due to padding (padding up to 8 bytes).

pg_walinspect extension , whose functions have capabilities similar to those of the pg_waldump utility, **is discussed in** the next chapter.

Part 4. Size of PGDATA / pg_wal directory

1) Look at the values of the parameters that determine the size of the journal directory:

```
postgres@tantor:~$ psql -c "\dconfig *wal_size"
List of configuration parameters
Parameter | Value
--------------+-------
max_wall_size | 1GB
min_wall_size | 80MB
(2 rows )
```

2) See how much space the directory with logs takes up:

```
postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
945M /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

Values can range from 900MB to 1.1GB.

3) I want to reduce the size of files in the directory. Change the configuration parameter that is taking up a lot of space in the directory:

```
postgres@tantor:~$
psql -c "alter system set max_wal_size='100MB';"
psql -c "select pg_reload_conf();"
pg_reload_conf
----------------
t
(1 row)
```

4) After changing the parameter, the directory space will not decrease. It will not decrease after switching to a new file (unless a checkpoint has been performed):

```
postgres@tantor:~$
psql -c "select pg_switch_wal();"
psql -c "select pg_switch_wal();"
psql -c "select pg_switch_wal();"
psql -c "select pg_switch_wal();"

pg_switch_wal
---------------
249/ABBE616A
(1 row)

pg_switch_wal
```

```
----------------
249/AC000000
(1 row)

pg_switch_wal
----------------
249/AC00008A
(1 row)

pg_switch_wal
----------------
249/AD00008A
(1 row)

postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
897M /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

In the example, a time checkpoint was executed.

5) Perform a checkpoint:

```
postgres @ tantor :~$ psql - c " checkpoint ; "
du -h $PGDATA/pg_wal
CHECKPOINT
4.0K /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
881M /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

After the checkpoint the place became smaller.

6) Perform a checkpoint:

```
postgres @ tantor :~$ psql - c " checkpoint ; "
CHECKPOINT
postgres@tantor:~$ du -h $PGDATA/pg_wal/
4.0K /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
881M /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

The space did not shrink because there was no activity.

7) Perform a log file switch to create an activity and checkpoint:

```
postgres@tantor:~$ psql -c "select pg_switch_wal();"
pg_switch_wal
----------------
249/AF000292
(1 row)

postgres@tantor:~$ psql -c "checkpoint;"
CHECKPOINT
postgres@tantor:~$ du -h $PGDATA/pg_wal/
4.0K /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
865M /var/lib/postgresql/tantor-se-16/data/pg_wal/
```

After the checkpoint, the space occupied by the logs decreased.

8) Look at the log messages to see how many WAL files were removed :

```
postgres@tantor:~$ cat $PGDATA/log/postgresql*

21:42:45.914 MSK [331791] LOG: checkpoint starting: time
```

tantor

```
21:42:49.240 MSK [331791] LOG: checkpoint complete: wrote 31 buffers (0.2%); 0 WAL file(s) added, 0 removed, 0 recycled; write=3.270 s, sync=0.019 s, total=3.326 s; sync files=3, longest=0.012 s, average=0.006 s; distance=226 kB, estimate=397099 kB; lsn=249/ABBE6070, redo lsn=249/ABBE6028
11:20:33.612 MSK [331789] LOG: received SIGHUP, reloading configuration files
11:20:33.613 MSK [331789] LOG: parameter "max_wal_size" changed to "100MB"
11:21:36.243 MSK [331791] LOG: checkpoint starting: wal
11:21:36.314 MSK [331791] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 3 removed , 0 recycled; write=0.001 s, sync=0.001 s, total=0.071 s; sync files=0, longest=0.000 s, average=0.000 s; distance=36968 kB , estimate=361086 kB; lsn=249/AE000070, redo lsn=249/AE000028
11:23:45.824 MSK [331791] LOG: checkpoint starting: immediate force wait
11:23:45.877 MSK [331791] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 1 removed , 0 recycled; write=0.001 s, sync=0.001 s, total=0.054 s; sync files=0, longest=0.000 s, average=0.000 s; distance=16384 kB , estimate=326616 kB; lsn=249/AF0000B8, redo lsn=249/AF000070
11:23:55.592 MSK [331791] LOG: checkpoint starting: immediate force wait
11:23:55.632 MSK [331791] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 0 removed , 0 recycled; write=0.001 s, sync=0.001 s, total=0.041 s; sync files=0, longest=0.000 s, average=0.000 s; distance=0 kB, estimate=293954 kB; lsn=249/AF000198, redo lsn=249/AF000150
11:24:07.754 MSK [331791] LOG: checkpoint starting: immediate force wait
11:24:07.801 MSK [331791] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) added, 1 removed , 0 recycled; write=0.001 s, sync=0.001 s, total=0.047 s; sync files=0, longest=0.000 s, average=0.000 s; distance=16383 kB , estimate=266197 kB; lsn=249/B0000070, redo lsn=249/B0000028
```

`distance =36968 kB` rounded up to 16MB (segment size) and it turns out that checkpointer removed three files (`3 removed`). This corresponds to the change in the space occupied by files in the directory: `945 M -897 M = 48MB`

`distance =16384 kB` corresponds to the deletion of one file (`1 removed`). This corresponds to a change in the space occupied by files in the directory: `897 M -881 M = 16 MB` and `881 M -865 M = 16 MB`

`You cannot use the pg_archivecleanup utility` to delete files from the pg_wal directory . Otherwise , the utility will not delete anything:

```
pg_controldata | grep " REDO WAL "
Latest checkpoint's REDO WAL file: 000000010000024A0000004B
pg_archivecleanup -n $PGDATA/pg_wal 000000010000024A0000004B
```

Or it will delete the file that is needed for recovery and without which **the** cluster will be destroyed if you specify an arbitrary file name in the **pg_wal directory** :

```
pg_controldata | grep "REDO WAL"
Latest checkpoint's REDO WAL file: 000000010000024A000000 4B
pg_archivecleanup -n $PGDATA/pg_wal 000000010000024A000000 88 | grep 0004B
/var/lib/postgresql/tantor-se-16/data/pg_wal/000000010000024A000000 4B
```

The checkpoint process creates WAL files in advance so that there is no delay during fast WAL switching . In the directory, the files **000000010000024 A 000000 4 B** ... **000000010000024 A 000000 88** are not historical.
files in **directory pg_wal** on a running instance is dangerous, since there is a possibility of deleting a file needed for recovery, replicas, subscriptions.

9) Reset the value to default:

```
postgres@tantor:~$
psql -c "alter system reset max_wal_size;"
psql -c "select pg_reload_conf();"
pg_reload_conf
----------------
t
(1 row)
```

10) This step does not need to be completed. See the following example.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The instance stops:

```
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
```

The cluster directory is deleted:

```
postgres@tantor:~$ rm -rf $PGDATA/*
```

A new cluster is created:

```
postgres @ tantor :~$ initdb - k
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "en_US.UTF-8".
The default database encoding has accordingly been set to "UTF8".
The default text search configuration will be set to "english".

Data page checksums are enabled.

fixing permissions on existing directory /var/lib/postgresql/tantor-se-16/data ... ok
creating subdirectories...ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... Europe/Moscow
creating configuration files...ok
running bootstrap script...ok
performing post-bootstrap initialization ... ok
syncing data to disk...ok

initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or --auth-local and --auth-
host, the next time you run initdb.

Success. You can now start the database server using:

pg_ctl -D /var/lib/postgresql/tantor-se-16/data -l logfile start
```

It's starting up instance :

```
postgres@tantor:~$ sudo restart
```

one log file in the directory :

```
postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000000000000001  archive_status
```

After switching the journal and checkpoint:

```
postgres@tantor:~$ psql -c "select pg_switch_wal();"
pg_switch_wal
---------------
0/175C7E2
(1 row)

postgres@tantor:~$ psql -c "checkpoint;"
CHECKPOINT
```

another log file is added:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000000000000002 000000010000000000000003 archive_status
```

After another switch and the checkpoint is not added:

```
postgres@tantor:~$ psql -c "select pg_switch_wal();"
pg_switch_wal
----------------
0/200016A
(1 row)

postgres@tantor:~$ psql -c "checkpoint;"
CHECKPOINT

postgres@tantor:~$ ls $PGDATA/pg_wal
000000010000000000000003 000000010000000000000004 archive_status
```

To generate a large volume of changes, pgbench starts creating tables with a scale of 64:

```
postgres@tantor:~$ pgbench -i -s 64
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
6400000 of 6400000 tuples (100%) done (elapsed 15.78 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 22.08 s (drop tables 0.00 s, create tables 0.03 s, client-side generate 15.92 s, vacuum 0.40 s, primary
keys 5.74 s).
```

The number of magazines has increased:

```
postgres @ tantor :~$ ls $ PGDATA / pg _ wal

000000010000000000000025 000000010000000000000032 00000001000000000000003F 00000001000000000000004C
000000010000000000000026 000000010000000000000033 000000010000000000000040 00000001000000000000004D
000000010000000000000027 000000010000000000000034 000000010000000000000041 00000001000000000000004E
000000010000000000000028 000000010000000000000035 000000010000000000000042 00000001000000000000004F
000000010000000000000029 000000010000000000000036 000000010000000000000043 000000010000000000000050
00000001000000000000002A 000000010000000000000037 000000010000000000000044 000000010000000000000051
00000001000000000000002B 000000010000000000000038 000000010000000000000045 000000010000000000000052
00000001000000000000002C 000000010000000000000039 000000010000000000000046 000000010000000000000053
00000001000000000000002D 00000001000000000000003A 000000010000000000000047 000000010000000000000054
00000001000000000000002E 00000001000000000000003B 000000010000000000000048 000000010000000000000055
00000001000000000000002F 00000001000000000000003C 000000010000000000000049 000000010000000000000056
000000010000000000000030 00000001000000000000003D 00000001000000000000004A archive_status
000000010000000000000031 00000001000000000000003E 00000001000000000000004B

postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
801M /var/lib/postgresql/tantor-se-16/data/pg_wal
```

`801 M` magazines .

Reduce the `value of the max_wal_size parameter :`

```
postgres@tantor:~$
psql -c "alter system set max_wal_size='100MB';"
psql -c "select pg_reload_conf();"
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row )
```

Let's perform switching of logs and checkpoints by the number of log files:

```
postgres@tantor:~$
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
for (( i=0; i <= 50 ; i+=1 ));
do
psql -c "select pg_switch_wal();" > /dev/null 2> /dev/null
psql -c "checkpoint;" > /dev/null
done
```

The number of files **has** decreased. The total size of log files has decreased to **the value of the min_wal_size** parameter :

```
postgres@tantor:~$ ls $PGDATA/pg_wal

000000010000000000000068 00000001000000000000006A 00000001000000000000006C
000000010000000000000069 00000001000000000000006B archive_status

postgres@tantor:~$ du -h $PGDATA/pg_wal
4.0K /var/lib/postgresql/tantor-se-16/data/pg_wal/archive_status
81M /var/lib/postgresql/tantor-se-16/data/pg_wal

postgres@tantor:~$ psql -c "show min_wal_size;"
min_wall_size
--------------
 80MB
(1 row)
```

Return **max_wal_size** to the default value and set the configuration parameters to the values before the cluster was deleted:

```
postgres@tantor:~$
psql -c "alter system set recovery_init_sync_method = 'syncfs'"
psql -c "alter system set track_io_timing = 'on'"
psql -c "alter system set logging_collector = 'on'"
psql -c "alter system reset max_wal_size;"
psql -c "create extension if not exists pageinspect;"
psql -c "create extension if not exists pg_visibility;"
psql -c "create extension if not exists pg_columnar;"
psql -c "create extension if bloom does not exist;"
psql -c "create extension if not exists pgstattuple;"
psql -c "create extension if not exists pg_buffercache;"
psql -c "alter system set shared_preload_libraries='pg_prewarm';"
psql -c "create extension if not exists pg_prewarm;"
sudo restart
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
CREATE EXTENSION
ALTER SYSTEM
CREATE EXTENSION
```

The result of this point: log files are created by the checkpointer process at the end of the checkpoint. The number of files is determined by the log generation activity. The number of files reaches a value approximately equal to **max_wal_size . When decreasing the** value gradually after the checkpoint is executed, the number of files decreases and reaches **min_wal_size . The number of files does not decrease** if there were checkpoints by volume **.**
Logs can also be held by replication slots.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Oleg Ivanov

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Practice for Chapter 14

Part 1. I/O statistics in the pg_stat_io view

the column names of the pg_stat_io view :

```
postgres=# select * from pg_stat_io where backend_type='checkpointer' limit 1
\gx
-[ RECORD 1 ]--+------------------------------
backend_type | checkpointer
object | relation
context | normal
reads |
read_time |
writes | 286
write_time | 4.04
writebacks | 286
writeback_time | 6.932
extends |
extend_time |
op_bytes | 8192
hits |
Evictions |
reuses |
fsyncs | 135
fsync_time | 313.348
stats_reset | 2035-01-01 01:01:26.002532+03
```

The view has many columns. To make the columns fit on the page, you can use column aliases:

```
postgres=# select backend_type, left(object,4) obj, context, writes w,
round(write_time::numeric,2) wt, writebacks wb, round(writeback_time::numeric,2)
wbt, extends e, extend_time et, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric,2) fst from pg_stat_io;
backend_type | obj | context | w | wt | wb | wbt | e | et | ev | ru | fs | fst
---------------------+------+-----------+----+------+----+------+-----+-------+----+----+----+--------
autovacuum launcher | rela | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
autovacuum launcher | rela | normal | 0 | 0.00 | 0 | 0.00 | | | 0 | | 0 | 0.00
autovacuum worker | rela | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
autovacuum worker | rela | normal | 0 | 0.00 | 0 | 0.00 | 3 | 0.085 | 0 | | 0 | 0.00
autovacuum worker | rela | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
client backend | rela | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
client backend | rela | bulkwrite | 0 | 0.00 | 0 | 0.00 | 6 | 0.136 | 0 | 0 | |
client backend | rela | normal | 0 | 0.00 | 0 | 0.00 | 165 | 1.967 | 0 | | 0 | 0.00
client backend | rela | vacuum | 24 | 0.24 | 0 | 0.00 | 0 | 0 | 0 | 24 | |
client backend | temp | normal | 50 | 0.39 | | | 151 | 1.567 | 51 | | |
background worker | rela | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
background worker | rela | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
background worker | rela | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
background worker | rela | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
background worker | temp | normal | 0 | 0.00 | | | 0 | 0 | 0 | | |
background writer | rela | normal | 0 | 0.00 | 0 | 0.00 | | | | | 0 | 0.00
checkpointer | rela | normal | 94 | 4.76 | 94 | 2.70 | | | | | 62 | 160.88
standalone backend | rela | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
standalone backend | rela | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
standalone backend | rela | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
standalone backend | rela | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
startup | rela | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
startup | rela | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
startup | rela | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
startup | rela | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
walsender | rela | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
walsender | rela | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
walsender | rela | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
walsender | rela | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
walsender | temp | normal | 0 | 0.00 | | | 0 | 0 | 0 | | |
(30 rows )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

In those fields where the values are empty - there can be no values in principle. In those fields where there are zeros there can be non-zero values. The rules in which fields the values cannot be are listed below.

Characteristics **of the pg_stat_io view** :

The writeback_time and fsync_time columns are populated **only** if **the track_io_timing = on configuration parameter .**

The columns of primary interest are writebacks and fsyncs , as they are the ones that cause system calls to be sent to the operating system.

writebacks are performed:

a) by server processes when expanding files (extends) and searching for a buffer to evict (by calling the GetVictimBuffer (..) function).

b) by the bgwriter and checkpointer processes in the normal context . The checkpointer process queue is used .

fsyncs :

a) are passed to the checkpointer process to be executed at the end of the checkpoint once for each file

b) if they cannot be transmitted, then they are executed by the process itself

c) when working with buffer rings are not counted.

object column can contain the values: relation (persistent storage object) and temp relation (temporary object). Writebacks and fsyncs are not performed with temporary objects , since temporary tables do not need to be guaranteed to be written - when the process crashes or after the instance is started, the files must be deleted.

context column can contain the following values: normal , bulkread , bulkwrite , vacuum . The last three are buffer rings. reuses are non-empty only for rings , the normal context is missing. fsyncs are not counted for rings . Bulkread does not have extends , since files do not change size when reading.

Autovacuum processes launcher and autovacuum worker do not work with bulkwrite type rings . autovacuum process launcher does not work with vacuum type ring .

bgwriter and checkpointer processes They don't work with rings.

U bgwriter And checkpointer V columns reads, hits, events always zeros.

At bgwriter and autovacuum launcher in column extends are zeros because they do not increase file sizes.

for the loggingg collector process because the loggingg collector process does not have access to shared memory.

Statistics are not collected for the archiver process, since the archiver runs commands or programs that perform input/output operations.

WAL processes Receiver and WAL Writers are missing from pg_stat_io (not implemented) .

When using rings, server processes can add a buffer to the ring and replace the block in the added buffer with a block of the object being worked with in the ring. This happens when:

a) there are no or few buffers in the buffer ring because the team has just started working

b) to replace a buffer in the ring with another buffer, because the previous buffer is pinned or used by another process and the block in the buffer cannot be replaced. Such a buffer is excluded from the ring.

evictions is filled with any context - both rings and normal context . evictions means that a block was replaced in the buffer.

reused is filled only for rings.

What is the difference between evictions and reused when working with rings?

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The replacement of a block in a buffer is taken into account in the `evictions statistics` of the corresponding type (`context`) of the ring. If the block is in a buffer related to a ring and the block is replaced with another block in a buffer of the same ring, then the block replacement is taken into account in the `reused statistics` of this type (`context`) of the ring. But if the table block is pinned by another process, then when the buffer is pinned to the ring, `the evictions` statistics are increased , not `the reused statistics` .

Columns **`of the pg_stat_io view`** :
`op_bytes` - values are always equal to the data block size: 8192.
`hits` - have almost no diagnostic value.

2) Do it commands :

```
postgres@tantor:~$
psql -c "alter system set track_io_timing = on;"
psql -c "alter system reset shared_buffers;"
psql -c "alter system set logging_collector = off;"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
pg_ctl stop
pg_ctl start
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
pg_stat_reset_shared
----------------------

(1 row)


pg_stat_reset_shared
----------------------

(1 row)

LOG: received fast shutdown request
LOG: database system is shut down
...
done
server stopped
done
server started
```

Disabling `logging_collector` and restarting the instance with `pg_ctl` will allow the diagnostic log to be output to the terminal. This is useful for monitoring checkpoints. The statistics in `the pg_stat_io` and `pg_ stat _ bgwriter views` are related and produce overlapping values, so they should be cleared at the same time. The `pg_stat_io view` was introduced in PostgreSQL version 16. `The pg_ stat _ bgwriter` view used before version 16 and the monitoring system and formulas for setting up checkpoints and bgwriter parameters are linked to this view, so pg_ stat _ bgwriter can be used for monitoring and analyzing statistics. Starting with version 17 from `pg_ stat _ bgwriter` columns related to server processes and checkpointer have been removed and the pg_stat_checkpointer view has appeared. In practice, there are no examples of queries to pg_stst_bgwriter, the view is relevant if PostgreSQL versions up to 16 are used and if it is necessary to migrate monitoring scripts to new versions.

3) Open **a second** linux terminal and run:

```
postgres@tantor:~$ pgbench -i -s 64
pgbench -T 6000 -c 64 -P 10
```

The test is run in 64 sessions.

4) In the first terminal, run psql and execute the command:

```
postgres@tantor:~$ psql
postgres=# select backend_type name, sum(writes) buffers_written,
round(sum(write_time)) w_time, sum(writebacks) writebacks, sum(evictions)
evictions, sum(fsyncs) fsyncs, round(sum(fsync_time)) fsync_time from pg_stat_io
group by backend_type having sum(writes)> 0 or sum(writebacks)> 0 or
sum(fsyncs)>0 or sum(evictions)>0;
name |buffers_written |w_time |writebacks |evictions |fsyncs |fsync_time
--------------+-------+-------+----------+----------+-------+----------
client backend| 102848 | 1100 | 0 | 0 | 0 | 0
checkpointer | 105 | 3 | 96 | | 0 | 0
(2 rows)
```

5) B first terminal do it command :

```
postgres =# \watch 10
```

This psql command will repeat the last command with an interval of 10 seconds. You can interrupt the repetition with the key combination `< c t rl + c >` .

6) The terminal windows will look like this:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
1 — Terminal
File  Edit  View  Bookmarks  Plugins  Settings  Help

progress: 2540.0 s, 1667.0 tps, lat 37.670 ms
progress: 2550.0 s, 1599.2 tps, lat 39.170 ms
progress: 2560.0 s, 1244.0 tps, lat 50.541 ms
progress: 2570.0 s, 1320.7 tps, lat 47.845 ms
progress: 2580.0 s, 1277.5 tps, lat 49.423 ms
progress: 2590.0 s, 1701.1 tps, lat 36.689 ms
progress: 2600.0 s, 1734.3 tps, lat 36.272 ms
progress: 2610.0 s, 1507.8 tps, lat 41.567 ms
progress: 2620.0 s, 1171.8 tps, lat 54.015 ms
progress: 2630.0 s, 1377.4 tps, lat 45.871 ms
progress: 2640.0 s, 1687.8 tps, lat 37.078 ms
progress: 2650.0 s, 1744.1 tps, lat 36.113 ms
progress: 2660.0 s, 1556.9 tps, lat 40.333 ms
progress: 2670.0 s, 1339.2 tps, lat 47.272 ms
progress: 2680.0 s, 1349.2 tps, lat 46.640 ms
progress: 2690.0 s, 1670.6 tps, lat 37.735 ms
progress: 2700.0 s, 1684.9 tps, lat 37.097 ms
progress: 2710.0 s, 1772.3 tps, lat 35.460 ms
progress: 2720.0 s, 1333.5 tps, lat 47.138 ms
progress: 2730.0 s, 1220.4 tps, lat 51.843 ms
progress: 2740.0 s, 1666.9 tps, lat 37.710 ms
progress: 2750.0 s, 1667.8 tps, lat 37.509 ms
progress: 2760.0 s, 1707.8 tps, lat 36.675 ms
progress: 2770.0 s, 1255.9 tps, lat 50.412 ms
progress: 2780.0 s, 1360.4 tps, lat 46.197 ms
progress: 2790.0 s, 1364.1 tps, lat 46.305 ms
progress: 2800.0 s, 1694.9 tps, lat 36.982 ms
progress: 2810.0 s, 1734.8 tps, lat 36.114 ms
progress: 2820.0 s, 1413.9 tps, lat 44.549 ms
progress: 2830.0 s, 1254.8 tps, lat 50.282 ms
progress: 2840.0 s, 1604.8 tps, lat 39.163 ms
progress: 2850.0 s, 1727.7 tps, lat 36.364 ms
progress: 2860.0 s, 1697.4 tps, lat 36.956 ms
progress: 2870.0 s, 1306.9 tps, lat 48.251 ms
progress: 2880.0 s, 1143.5 tps, lat 55.269 ms
```

```
                       Wed 15 Jan 2025 12:03:33 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3005779 |  88248 |          0 |       0 |   5695490 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          774600 |  12404 |     774592 |  876097 |           |      0 |          0
 checkpointer      |           47779 |    829 |      47768 |   41103 |           |    675 |     218332
(4 rows)

2025-01-15 00:03:34.631 MSK [76376] LOG:  checkpoint complete: wrote 1768 buffers (10.8%); 0 WAL file(s
) added, 3 removed, 31 recycled; write=43.092 s, sync=7.219 s, total=51.773 s; sync files=16, longest=6
.760 s, average=0.452 s; distance=570116 kB, estimate=570116 kB; lsn=A9/75F4B160, redo lsn=A9/55CDE690
2025-01-15 00:03:34.632 MSK [76376] LOG:  checkpoint starting: wal
                       Wed 15 Jan 2025 12:03:43 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3016272 |  88817 |          0 |       0 |   5718169 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          777800 |  12451 |     777792 |  879522 |           |      0 |          0
 checkpointer      |           48639 |    846 |      48613 |   41579 |           |    690 |     225546
(4 rows)

                       Wed 15 Jan 2025 12:03:53 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3026505 |  89100 |          0 |       0 |   5738151 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          780700 |  12493 |     780672 |  883418 |           |      0 |          0
 checkpointer      |           48642 |    846 |      48613 |   41579 |           |    690 |     225546
(4 rows)

                       Wed 15 Jan 2025 12:04:03 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3039628 |  89436 |          0 |       0 |   5763777 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          784500 |  12553 |     784448 |  885711 |           |      0 |          0
 checkpointer      |           48642 |    846 |      48613 |   41579 |           |    690 |     225546
(4 rows)

                       Wed 15 Jan 2025 12:04:13 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3054015 |  89802 |          0 |       0 |   5791013 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          788200 |  12605 |     788160 |  888252 |           |      0 |          0
 checkpointer      |           48777 |    850 |      48773 |   41601 |           |    690 |     225546
(4 rows)

                       Wed 15 Jan 2025 12:04:23 AM MSK (every 10s)

       name        | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
-------------------+-----------------+--------+------------+---------+-----------+--------+------------
 client backend    |         3066084 |  90103 |          0 |       0 |   5813552 |      0 |          0
 autovacuum worker |          197094 |   3698 |          0 |       0 |     53361 |      0 |          0
 background writer |          791300 |  12645 |     791296 |  891938 |           |      0 |          0
 checkpointer      |           48778 |    850 |      48773 |   41601 |           |    690 |     225546
(4 rows)

2025-01-15 00:04:31.079 MSK [76376] LOG:  checkpoint complete: wrote 1003 buffers (6.1%); 0 WAL file(s)
 added, 3 removed, 30 recycled; write=44.029 s, sync=10.963 s, total=56.447 s; sync files=16, longest=1
0.521 s, average=0.686 s; distance=535028 kB, estimate=566608 kB; lsn=A9/97D32AB8, redo lsn=A9/7675B940
```

You can observe how the input/output indicators change across processes.

`fsyncs` on files in tablespaces executes the process che c kpointer and the field is updated only after a checkpoint of any type. Diagnostic messages about the execution of checkpoints are output to the terminal. In the example, a checkpoint "on demand" is visible - due to the generation of a large volume of WAL. `max _ wal _ size = 1 GB` and upon reaching **`distance = 570116 kB`** the checkpoint begins.

`sync` blocks were synchronized **`files =16`** files, which corresponds to the difference in the fsyncs column `690 - 675` =15. It was written **`wrote 1768 buffers`** . Executing system calls `sync _ file _ range ( fd , offset , nbytes , SYNC _ FILE _ RANGE _ WRITE ) .`

took **`sync =7.219 s`** , which corresponds to the value in the column fsync_time = `225546 - 218332` = 7.214 seconds.

Blocks from the buffer cache were sent to the write queue (to the checkpoint process):

server processes `buffers _ written` **`3066084`** - **`3005779`** = 60305. Server processes that spent `w _ time on this 90103-88248=1.855` seconds.

`background writer` column `buffers_written` **`791300`** - **`774600`** = 16700 for `12645-12404=241` milliseconds .

At the same time, the background recording process itself sent blocks to the Linux page cache in approximately the same quantity and spent `wb _ time 891938 - 876097 = 15.841` seconds on this.

7) Quite a large number of block identifiers were sent to the write queue for the checkpointer process. However, the checkpoint process sent `48773-47768 = 1005` blocks to the operating system page cache.

To calculate the values of fields that changed each time a query was made, you need to take the results of queries that correspond to the intervals between checkpoints. For example, queries immediately after a checkpoint. You can't get exact values, but you can estimate them.

One call to `sync _ file _ range` took, on average, `21833/675 = 32.345` milliseconds to execute.

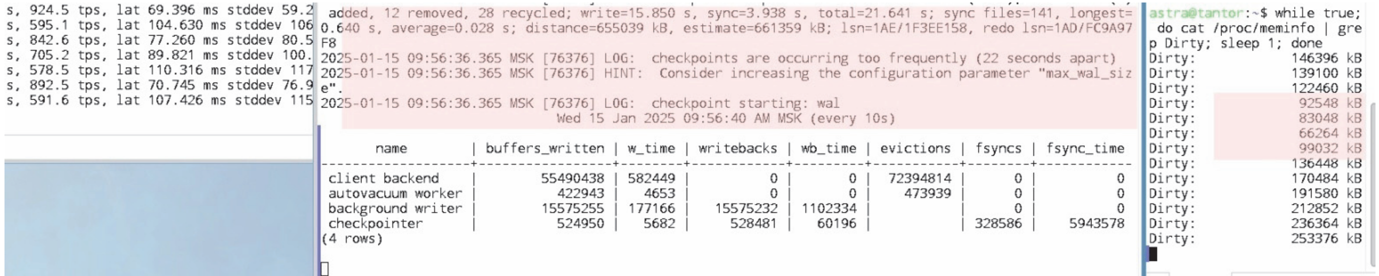
```
                              12:03:33 AM MSK (every 10s)

   name | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
   ------------------+--------+--------+-----------+---------+-----------+--------+------------
   client backend | 3005779 | 88248 | 0 | 0 | 5695490 | 0 | 0
   autovacuum worker | 197094 | 3698 | 0 | 0 | 53361 | 0 | 0
   background writer | 774600 | 12404 | 774592 | 876097 | | 0 | 0
   checkpointer | 47779 | 829 | 47768 | 41103 | | 675 | 218332
   (4 rows)

   00:03:34.631 MSK [76376] LOG: checkpoint complete: wrote 1768 buffers (10.8%); 0 WAL file(s) added, 3
removed, 31 recycled; write=43.092 s, sync=7.219 s , total=51.773 s; sync files=16 , longest=6.760 s,
average=0.452 s; distance=570116 kB , estimate=570116 kB; lsn=A9/75F4B160, redo lsn=A9/55CDE690
   00:03:34.632 MSK [76376] LOG: checkpoint starting: wal
   12:03:43 AM MSK (every 10s)

   name | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
   ------------------+--------+--------+-----------+---------+-----------+--------+------------
   client backend | 3016272 | 88817 | 0 | 0 | 5718169 | 0 | 0
   autovacuum worker | 197094 | 3698 | 0 | 0 | 53361 | 0 | 0
   background writer | 777800 | 12451 | 777792 | 879522 | | 0 | 0
   checkpointer | 48639 | 846 | 48613 | 41579 | | 690 | 225546
   (4 rows)

   12:03:53 AM MSK (every 10s)

   name | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
   ------------------+--------+--------+-----------+---------+-----------+--------+------------
   client backend | 3026505 | 89100 | 0 | 0 | 5738151 | 0 | 0
   autovacuum worker | 197094 | 3698 | 0 | 0 | 53361 | 0 | 0
   background writer | 780700 | 12493 | 780672 | 883418 | | 0 | 0
   checkpointer | 48642 | 846 | 48613 | 41579 | | 690 | 225546
   (4 rows)

   12:04:03 AM MSK (every 10s)

   name | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
   ------------------+--------+--------+-----------+---------+-----------+--------+------------
   client backend | 3039628 | 89436 | 0 | 0 | 5763777 | 0 | 0
   autovacuum worker | 197094 | 3698 | 0 | 0 | 53361 | 0 | 0
   background writer | 784500 | 12553 | 784448 | 885711 | | 0 | 0
   checkpointer | 48642 | 846 | 48613 | 41579 | | 690 | 225546
   (4 rows)

   12:04:13 AM MSK (every 10s)

   name | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
   ------------------+--------+--------+-----------+---------+-----------+--------+------------
   client backend | 3054015 | 89802 | 0 | 0 | 5791013 | 0 | 0
   autovacuum worker | 197094 | 3698 | 0 | 0 | 53361 | 0 | 0
   background writer | 788200 | 12605 | 788160 | 888252 | | 0 | 0
   checkpointer | 48777 | 850 | 48773 | 41601 | | 690 | 225546
   (4 rows)

   12:04:23 AM MSK (every 10s)

   name | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
   ------------------+--------+--------+-----------+---------+-----------+--------+------------
   client backend | 3066084 | 90103 | 0 | 0 | 5813552 | 0 | 0
   autovacuum worker | 197094 | 3698 | 0 | 0 | 53361 | 0 | 0
   background writer | 791300 | 12645 | 791296 | 891938 | | 0 | 0
   checkpointer | 48778 | 850 | 48773 | 41601 | | 690 | 225546
   (4 rows)

   00:04:31.079 MSK [76376] LOG: checkpoint complete: wrote 1003 buffers (6.1%); 0 WAL file(s) added, 3 removed,
30 recycled; write=44.029 s, sync=10.963 s, total=56.447 s; sync files=16, longest=10.521 s, average=0.686 s;
distance=535028 kB, estimate=566608 kB; lsn=A9/97D32AB8, redo lsn=A9/7675B940
   00:04:31.080 MSK [76376] LOG: checkpoint starting: wal
   12:04:33 AM MSK (every 10s)

   name | buffers_written | w_time | writebacks | wb_time | evictions | fsyncs | fsync_time
   ------------------+--------+--------+-----------+---------+-----------+--------+------------
   client backend | 3074683 | 90654 | 0 | 0 | 5830219 | 0 | 0
   autovacuum worker | 197590 | 3707 | 0 | 0 | 54303 | 0 | 0
   background writer | 793500 | 12674 | 793472 | 897126 | | 0 | 0
    checkpointer | 49146 | 854 | 49131 | 42288 | | 704 | 236498
```

```
(4 rows)
```

8) Launch the third terminal and execute the command in it:

```
astra@tantor:~$ while true; do cat /proc/meminfo | grep Dirty; sleep 1; done
```

Example of a window with a command on the right:



You can see in dynamics that ~4 seconds (`sync =3.938 s` in the subsequent checkpoint message) before the checkpoint message appears, the volume of dirty pages in the linux page cache decreases. After the checkpoint, it increases, since pgbench (the left window in the picture) dirtys the pages.

You can interrupt the cycle by pressing the key combination `<ctrl+c> in the window` and if the terminal window is not needed, close it.

9) Stop the pgbench test with the key combination `<ctrl+c>` .

Part 2. Running fsyncs when checkpointer is stopped

1) Run the commands:

```
postgres@tantor:~$
time psql -f initdrop.sql > /dev/null 2> /dev/null
psql -c "\dconfig *flush*"
sudo restart

real 0m5.547s
user 0m0.009s
sys 0m0.000s

List of configuration parameters
Parameter | Value
------------------------+-------
backend_flush_after | 0
bgwriter_flush_after | 512kB
checkpoint_flush_after | 256kB
wal_writer_flush_after | 1MB
(4 rows)
```

The commands remove test tables if any, print buffer sizes, and restart the instance.

2) Find the process numbers checkpointer and background writer instance:

```
postgres@tantor:~$ ps -ef | egrep "backgr|checkp"
postgres 286008 286006 0 Jan18 ? 00:00:00 postgres: 15/main: checkpointer
postgres 286009 286006 0 Jan18 ? 00:00:12 postgres: 15/main: background writer
postgres 421548 421544 0 21:32 ? 00:00:03 postgres: checkpointer
postgres 421549 421544 0 21:32 ? 00:00:00 postgres: background writer
postgres 421692 421352 0 21:51 pts/0 00:00:00 grep -E backgr|checkp
```

3) Run psql in the second terminal and the commands:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)
```

The commands reset I/O statistics in two views.

4) Run psql in the second terminal and the commands:

```
postgres=#
select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
\ watch 1
```

print statistics from the **pg_stat_io view once per** second:

```
 backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
---------------+---------+---+------+----+------+----+----+------+----+----+----+----
 client backend | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 16 | 0 | | 0 | 0
(1 row )
```

5) Launch the third terminal and execute the command in it:

```
astra@tantor:~$ while true; do cat /proc/meminfo | grep Dirty; sleep 5; done
```

6) Run the test script in the first terminal:

```
postgres@tantor:~$
time psql -f init.sql > /dev/null 2> /dev/null
```

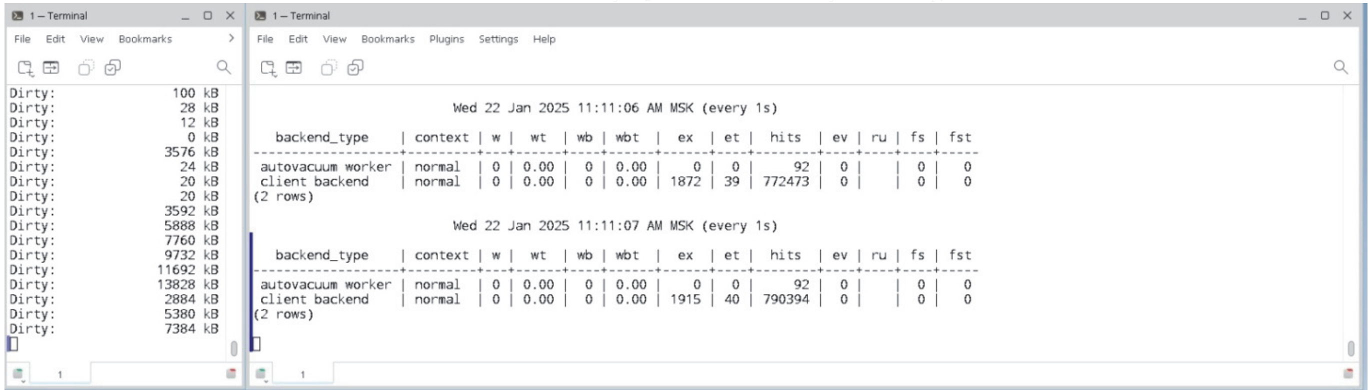
The second script will finish working in **2 m 23** :

```
real 2m23 .289s
user 0m0.396s
sys 0m0.168s
```

7) While the script is running, monitor the input/output statistics:

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+---+------+----+------+------+-----+---------+----+----+----+-----
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 19814 | 0 | | 0 | 0
autovacuum worker | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 16884 | 0 | 0 | |
client backend | normal | 0 | 0.00 | 0 | 0.00 | 5553 | 120 | 2310438 | 0 | | 0 | 0
(3 rows)
```

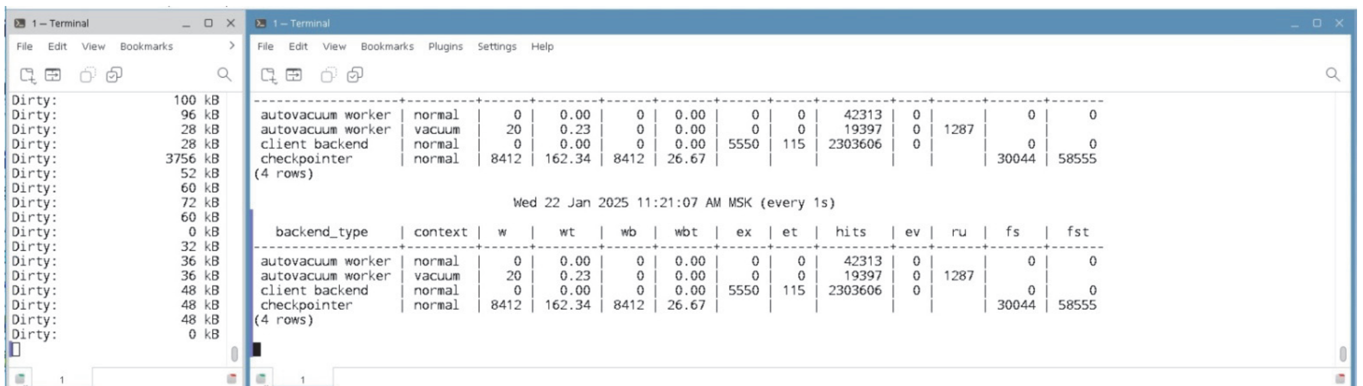
Empty fields cannot have values in principle. For example, when working with a `vacuum buffer ring,` fsyncs cannot be sent .



In **fsyncs columns fs , fsync_time fst** for all processes except the checkpointer process are zero. For the checkpointer process, non-zero values will be updated and increased only after the checkpoint ends. The interval between checkpoints is 5 minutes. If the checkpoint has begun, then for ~4.5 minutes the numbers in the chec kpointer line will increase :

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+------+-------+------+-------+------+-----+---------+----+----+----+-----
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 67705 | 0 | | 0 | 0
autovacuum worker | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 46055 | 0 | 0 | |
client backend | normal | 0 | 0.00 | 0 | 0.00 | 5553 | 120 | 4425701 | 0 | | 0 | 0
 checkpointer | normal | 3376 | 63.99 | 3360 | 24.57 | | | | | | 0 | 0
(4 rows)
```

Example after checkpoint, fsyncs=30044 which is the number of files in the cluster tablespaces whose blocks were updated during the checkpoint. This is 30000 tables and indexes created by the script and 44 relations of the system catalog:

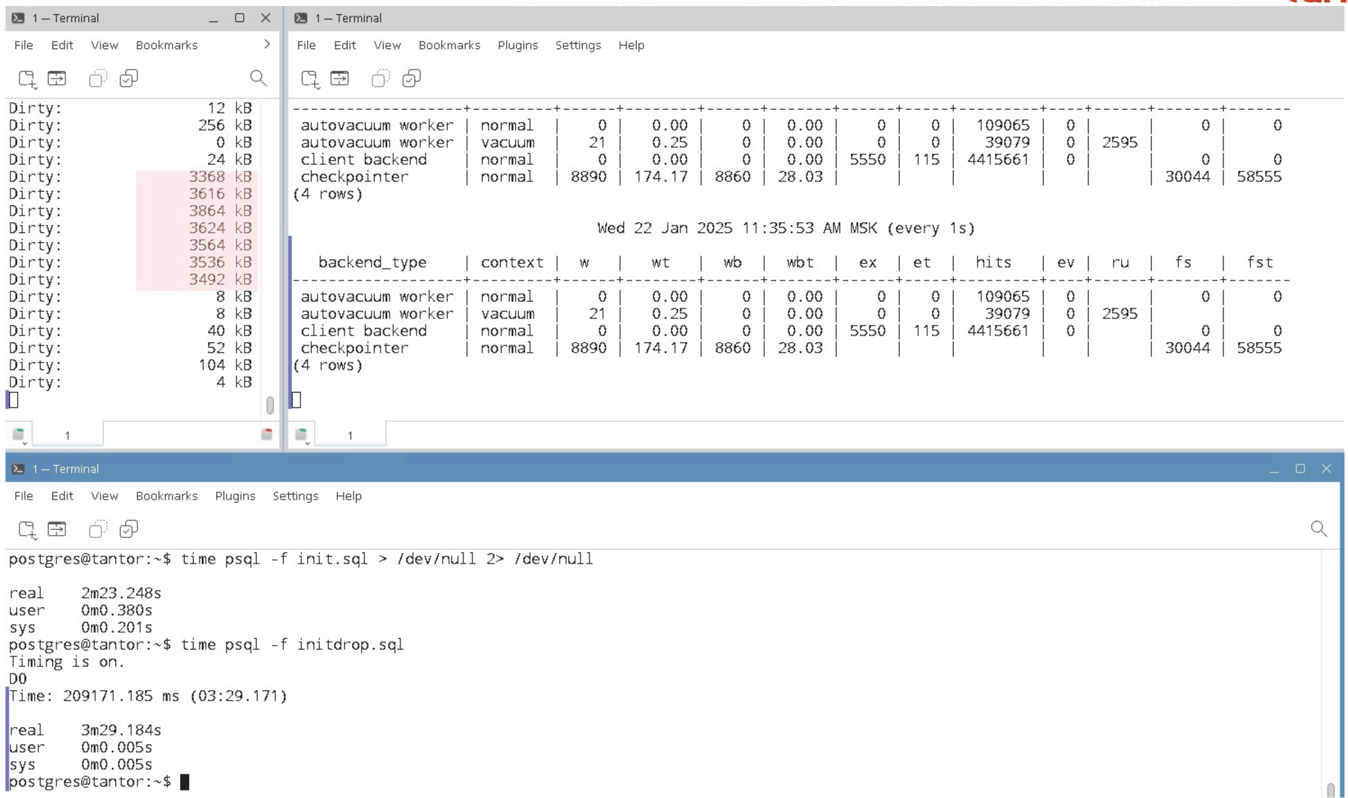


Whatever tests are run, the **fsyncs columns fs , fsync_time fst** for all processes except the checkpointer process will have only zeros. Zeros mean normal operation of the instance. However, under heavy load and non-uniform access to memory (NUMA), there is a possibility that checkpointer will not be able to process the queue of blocks for synchronization (writeback) and in this case, processes that have in the **fsyncs column fs zeros can send** writeback and fsync system calls to the operating system on their own. Let's look at such cases and how this affects performance.

8) Run the table deletion script:

```
postgres @ tantor :~$ time psql - f initdrop . sql > / dev / null 2> / dev /
null
real 0m5.614s
user 0m0.009s
sys 0 m 0.000 s
```

After some time after the script is executed, the number of dirty pages in the operating system page cache will increase :



Keeping pages in the linux cache is configured by linux options.

If you run the table drop script after stopping checkpointer , the drop commands will fail to execute, and the locks in pg_locks will not be reflected. In the example in the picture , the initdrop.sql script hung for 3m29.184s until the checkpointer process was resumed. You can use the pg_stat_activity view to monitor such delays. presentation will event expectations `RegisterSyncRequest` :

```
postgres=# select * from pg_stat_activity where wait_event_type='Timeout'\gx
-[ RECORD 1 ]----+--------------------------------------------------------------------------------
-
datid | 5
datname | postgres
pid | 577742
leader_pid |
usesysid | 10
usename | postgres
application_name | psql
client_addr |
client_hostname |
client_port | -1
backend_start | 13:18:01.647753+03
xact_start | 13:18:01.650781+03
query_start | 13:18:01.650781+03
state_change | 13:18:01.650785+03
wait_event_type | Timeout
wait_event | RegisterSyncRequest
state | active
backend_xid |
backend_xmin |
query_id |
query | DO +
| $$ +
| begin +
| for i in 1..10000 by 100 loop +
| for j in 0..99 loop +
| execute concat('drop table if exists test.film_summary',i+j);+
| end loop; +
| commit; +
| end loop; +
| execute 'drop schema if exists test cascade'; +
| end; +
| $$ +
| LANGUAGE plpgsql;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
backend_type | client backend
```

Waiting for `RegisterSyncRequest` is a transfer of synchronization requests to the checkpoint process due to the local request queue being full. The event is not related to the removal method (does not depend on the presence of plpgsql, execute, commit code). The queue size does not depend on the `* _ flush_after configuration parameters` .

9) Stop the checkpointer process by specifying its PID:

```
postgres @ tantor :~$ kill - STOP 421548
```

10) Check that the process is stopped:

```
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
 421548 T postgres
```

11) Run the script to create 10000 tables:

```
postgres@tantor:~$ time psql -f init.sql > /dev/null 2> /dev/null
```

12) After 1 minute (after increasing extends by ~ 3000 blocks) in the **`fsyncs , fsync_time columns`** of `the client backend` (server processes) non-zero values will appear **`and`** will start to grow:

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+---+------+----+------+------+-----+---------+----+------+----+-----
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 8588 | 0 | | 0 | 0
autovacuum worker | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 5679 | 0 | 2896 | |
client backend | normal | 0 | 0.00 | 0 | 0.00 | 3031 | 110 | 1260557 | 0 | | 19 | 165
(3 rows)
```

Once the server process has filled the `Checkpointer structure Data` block identifiers in shared memory, so there was no space left in it, the server process had to write to local memory. Size structures 512 Kb :

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) FROM
pg_shmem_allocations where name like '%Check%' ORDER BY size DESC;
 name | allocated_size | pg_size_pretty
----------------------+----------------+----------------
 Checkpointer Data | 524416 | 512 kB
 Checkpoint BufferIds | 327680 | 320 kB
 (2 rows)
```

In `Checkpointer Data` are written by all processes. The checkpointer process reads from the structure and moves the block identifiers to a hash table in its local memory, which is called the " `Pending Ops Table` " or " `pending sync hash` ".

13) After `5 m 32` the script will be completed:

```
real    5 m 32 .496 s
user 0m0.387s
sys 0m0.180s
```

The duration of the script execution with fsyncs issued by the server process, and not the checkpoint process, increased from `2 m 23` to `5 m 32` , by 3 times . If the checkpointer process fails and other processes cannot write pointers to their blocks into the checkpointer memory, then the efficiency of the work decreases.

During operation, the number of fsyncs initiated by the server process **`is 16138`** :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+--------+-------+--------+-------+-------+---------+----+----+-------+--------
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 168119 | 0 | | 0 | 0
autovacuum worker | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 77867 | 0 | 0 | |
client backend | normal | 0 | 0.00 | 0 | 0.00 | 11104 | 29495 | 6736834 | 0 | | 16138 | 164095
checkpointer | normal | 4286 | 81.14 | 4256 | 27.17 | | | | | | 0 | 0
(4 rows)
```

11) Resume the checkpointer process:

```
postgres@tantor:~$ kill -CONT 421548
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
```

12) Perform a checkpoint:

```
postgres @ tantor :~$ time psql - c " checkpoint ;"
CHECKPOINT

real 0m1.043s
user 0m0.009s
sys 0m0.000s
```

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+--------+-------+--------+-------+-------+---------+----+----+-------+--------
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 180324 | 0 | | 0 | 0
autovacuum worker | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 77867 | 0 | 0 | |
client backend | normal | 0 | 0.00 | 0 | 0.00 | 11104 | 29495 | 6736892 | 0 | | 16138 | 164095
checkpointer | normal | 13339 | 169.75 | 13339 | 201.58 | | | | | | 46456 | 90487
(4 rows )
```

13) See how the checkpoints are reflected in the log:

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*
22:26:46.992 MSK [427531] LOG: starting Tantor Special Edition 16.2.0 e12e484f on x86_64-pc-linux-gnu, compiled by gcc (Astra 12.2.0-14.astra3) 12.2.0, 64-bit
22:26:46.992 MSK [427531] LOG: listening on IPv4 address "127.0.0.1", port 5432
22:26:46.993 MSK [427531] LOG: listening on IPv6 address "::1", port 5432
22:26:47.002 MSK [427531] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
22:26:47.031 MSK [427536] LOG: database system was shut down at 22:26:46 MSK
22:26:47.052 MSK [427531] LOG: database system is ready to accept connections
22:33:20.721 MSK [427534] LOG: checkpoint starting: time
22:44:06.515 MSK [427534] LOG: checkpoint complete: wrote 5692 buffers (34.7%); 0 WAL file(s) added, 0 removed, 14 recycled; write=529.876 s, sync=115.114 s, total=645.794 s; sync files=46417 , longest=0.041 s, average=0.003 s; distance=220844 kB, estimate=220844 kB; lsn=3/5B929D50, redo lsn=3/4E4F28C0
22:48:01.468 MSK [427534] LOG: checkpoint starting: immediate force wait
22:48:02.770 MSK [427534] LOG: checkpoint complete: wrote 3068 buffers (18.7%); 0 WAL file(s) added, 0 removed, 15 recycled; write=0.131 s, sync=0.112 s, total=1.302 s; sync files=41, longest=0.027 s, average=0.003 s; distance=253551 kB, estimate=253551 kB; lsn=3/5DC8E8A8, redo lsn=3/5DC8E860
22:53:01.203 MSK [427534] LOG: checkpoint starting: time
23:28:28.751 MSK [427534] LOG: checkpoint complete: wrote 5922 buffers (36.1%); 0 WAL file(s) added, 0 removed, 8 recycled; write=2035.711 s, sync=91.719 s, total=2127.548 s; sync files=46423 , longest=0.028 s, average=0.002 s; distance=123698 kB, estimate=240566 kB; lsn=3/72C6A338, redo lsn=3/6555B2F0
23:28:48.533 MSK [427534] LOG: checkpoint starting: immediate force wait
23:28:49.561 MSK [427534] LOG: checkpoint complete: wrote 7430 buffers (45.3%); 0 WAL file(s) added, 0 removed, 13 recycled; write=0.285 s, sync=0.087 s, total=1.028 s; sync files=35, longest=0.018 s, average=0.003 s; distance=220220 kB, estimate=238531 kB; lsn=3/72C6A460, redo lsn=3/72C6A418
```

Part 3. I/O Performance Testing

In the previous test, tables were created, but a small number of blocks were dirty, all of which fit into the buffer cache, so autovacuum and bgwriter activity was not observed.
Let's see what the I/O load will be if all buffers in the buffer cache are occupied.

1) 3) Run psql in the second terminal and the commands:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
pg_stat_reset_shared
----------------------
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
(1 row)

pg_stat_reset_shared
----------------------

(1 row)
```

2) Run the script to insert 300 rows into each of the 10,000 tables:

```
postgres@tantor:~$ cat init2.sql
select format('insert into test.film_summary%1$s select i, %2$s || i, i from
generate_series(0, 153) as i;', g.id, E'\'text number \'') from
generate_series(1, 10000) as g(id) \gexec
postgres@tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null

real 0m44.707s
user 0m0.351s
sys 0 m 0.149 s
```

The insertion was completed in 44 seconds.

3) Stop the checkpointer process by specifying its PID :

```
postgres@tantor:~$ kill -STOP 421548
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
 421548 T postgres
```

4) Run the script to insert 300 rows into each of the 10,000 tables:

```
postgres@tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null
```

real 4m22.384s
user 0m0.400s
sys 0 m 0.220 s

m 22 instead of 44 seconds .

5) While the script is running, watch in the window with the cycle how the number of dirty pages in the Linux cache changes:

```
Dirty: 4396 kB
Dirty: 4436 kB
Dirty: 280 kB
Dirty: 88 kB
Dirty: 220 kB
Dirty: 668 kB
Dirty: 672 kB
Dirty: 888 kB
Dirty: 788 kB
Dirty: 844 kB
Dirty: 1428 kB
Dirty: 1500 kB
Dirty: 1580 kB
Dirty: 1200 kB
```

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+----------+-------+---------+-------+--------+---------+-------+----+-------+--------
autovacuum worker | normal | 5 | 123.04 | 0 | 0.00 | 0 | 0 | 26649 | 433 | | 5 | 113
client backend | normal | 4723 | 1826.44 | 0 | 0.00 | 50000 | 126423 | 3347825 | 58748 | | 10995 | 170635
background writer | normal | 33122 | 221750.88 | 33088 | 1959.54 | | | | | | 13326 | 195512
checkpointer | normal | 3535 | 66.54 | 3520 | 9.97 | | | | | | 0 | 0
(4 rows )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

6) Stop the bgwriter process by specifying its PID :

```
postgres@tantor:~$ kill -STOP 421549
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
 421548 T postgres
421549 T postgres
```

7) Run psql in the second terminal and the commands:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)
```

8) Run psql in the second terminal and the commands:

```
postgres=#
  select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
  \ watch 1
```

9) Run the script to insert 153 rows (one block) into each of the 10,000 tables:

```
postgres@tantor:~$ cat init2.sql
  select format('insert into test.film_summary%1$s select i, %2$s || i, i from
generate_series(0, 153 ) as i;', g.id, E'\'text number \'') from
generate_series(1, 10000) as g(id) \gexec

postgres @tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null
```

10) When bgwriter is stopped , the script execution time increased:

```
real 6m57.672s
user 0m0.426s
sys 0m0.165s
```

11) Resume the checkpointer and bgwriter processes :

```
postgres @ tantor :~$ kill - CONT 421549
```

12) Look how the statistics will change:

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+----------+-----+------+-------+--------+---------+-------+----+-------+--------
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 29544 | 0 | | 0 | 0
client backend | normal | 25551 | 263639.10 | 0 | 0.00 | 10000 | 102812 | 1675770 | 44402 | | 35551 | 320925
 background writer | normal | 200 | 1875.13 | 192 | 0.33 | | | | | | 200 | 1505
(3 rows)

Wed 22 Jan 2025 12:14:11 AM MSK (every 1s)

backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+----------+-----+------+-------+--------+---------+-------+----+-------+--------
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 29544 | 0 | | 0 | 0
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
client backend | normal | 25551 | 263639.10 | 0 | 0.00 | 10000 | 102812 | 1675770 | 44402 | | 35551 | 320925
background writer | normal | 300 | 2806.14 | 320 | 0.53 | | | | | | 300 | 2252
(3 rows)

Wed 22 Jan 2025 12:14:12 AM MSK (every 1s)

backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+----------+------+-------+--------+---------+---------+--------+----+-------+--------
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 29544 | 0 | | 0 | 0
client backend | normal | 25551 | 263639.10 | 0 | 0.00 | 10000 | 102812 | 1675770 | 44402 | | 35551 | 320925
background writer | normal | 400 | 3723.27 | 384 | 0.63 | | | | | | 400 | 2986
(3 rows)
```

Once V bgwriter writes in a second **bgwriter_lru_maxpages=** 100 blocks And So How V each table file main layers consists of from one block , then gives By to everyone file one fsync . If more than 153 rows were inserted into the table (so that there were more blocks in each file), then the number of fsyncs per second would be less than writes .

13) Resume the checkpointer process:

```
postgres @ tantor :~$ kill - CONT 421548
postgres@tantor:~$ ps -eo pid,s,comm | grep postgres | grep T
```

14) Run the script that vacuums full all tables:

```
postgres@tantor:~$ cat init4.sql
select format('vacuum (freeze, analyze) test.film_summary%s;', g.id) from
generate_series(1, 10000) as g(id) \gexec
postgres@tantor:~$ time psql -f init4.sql > /dev/null 2> /dev/null
```

While the vacuum is running, the number of dirty pages in the Linux cache is increased:

```
Dirty: 48688 kB
Dirty: 45060 kB
Dirty: 46588 kB
Dirty: 48048 kB
Dirty: 48860 kB
Dirty: 50196 kB
Dirty: 46844 kB
Dirty: 48644 kB
```

A new line will appear with a **vacuum type buffer ring** :

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+----------+------+---------+-------+--------+---------+-------+----+-------+--------
autovacuum worker | normal | 11 | 0.26 | 0 | 0.00 | 0 | 0 | 48157 | 34 | | 0 | 0
autovacuum worker | vacuum | 24 | 0.40 | 0 | 0.00 | 0 | 0 | 1366 | 27 | 0 | |
client backend | normal | 31060 | 263736.52 | 0 | 0.00 | 20864 | 103049 | 8178640 | 87519 | | 35551 | 320925
client backend | vacuum | 2670 | 45.12 | 0 | 0.00 | 0 | 0 | 21612 | 21615 | 0 | |
background writer | normal | 44522 | 25762.81 | 44544 | 5029.43 | | | | | | 2685 | 19948
checkpointer | normal | 1534 | 16.76 | 1541 | 9.91 | | | | | | 36425 | 83780
(6 rows )
```

The fsync number is not increased anywhere because checkpointer is running. It will perform fsyncs at the end of its cycle and the number will increase from 36425 to **65799** by the number of files.

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+-------+----------+-------+---------+-------+--------+----------+--------+-----+-------+--------
autovacuum worker | normal | 18 | 0.38 | 0 | 0.00 | 0 | 0 | 74617 | 125 | | 0 | 0
autovacuum worker | vacuum | 600 | 5.79 | 0 | 0.00 | 0 | 0 | 4433 | 86 | 561 | |
client backend | normal | 33246 | 263777.41 | 0 | 0.00 | 25081 | 103136 | 10702396 | 104216 | | 35551 | 320925
client backend | vacuum | 3743 | 62.30 | 0 | 0.00 | 0 | 0 | 30000 | 30000 | 0 | |
background writer | normal | 66713 | 26172.87 | 66688 | 7645.14 | | | | | | 2685 | 19948
checkpointer | normal | 1534 | 16.76 | 1552 | 9.92 | | | | | | 65799 | 161190
(6 rows)
```

If the number has not increased, then in order not to wait up to 5 minutes for a checkpoint, you can perform a checkpoint manually:

```
postgres@tantor:~$ time psql -c "checkpoint;"
CHECKPOINT
```

15) These tests can be used to evaluate how buffer sizes affect test execution time:

```
alter system set backend_flush_after = 1;
alter system set bgwriter_flush_after = 1;
alter system set checkpoint_flush_after = 1;
select pg_reload_conf();
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');

postgres@tantor:~$ time psql -f init2.sql > /dev/null 2> /dev/null

real 1m2.733s
user 0m0.403s
sys 0m0.163s
```

Time increased from `0 m 44` to `1 m 2`.

```
postgres@tantor:~$ time psql -f initdrop.sql > /dev/null 2> /dev/null

real 0m9.544s
user 0m0.010s
sys 0m0.000s
```

Table deletion time also increased from `0 m 5.547 s` up to `0 m 9.544 s` .

16) If you changed the parameter values, then return the parameter values to the default values:

```
postgres=# alter system reset backend_flush_after;
alter system reset bgwriter_flush_after;
alter system reset checkpoint_flush_after;
select pg_reload_conf();
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
ALTER SYSTEM
ALTER SYSTEM
ALTER SYSTEM
pg_reload_conf
----------------
t
(1 row)

pg_stat_reset_shared
--------------------

(1 row)

pg_stat_reset_shared
--------------------

(1 row)
```

4. Choosing the size of temp_buffers when working with temporary tables using pg_stat_io

1) In the first terminal, run:

```
astra@tantor:~$ while true; do cat /proc/meminfo | grep Dirty; sleep 3; done
```

2) In the second terminal in psql run the commands:

```
postgres=#
```

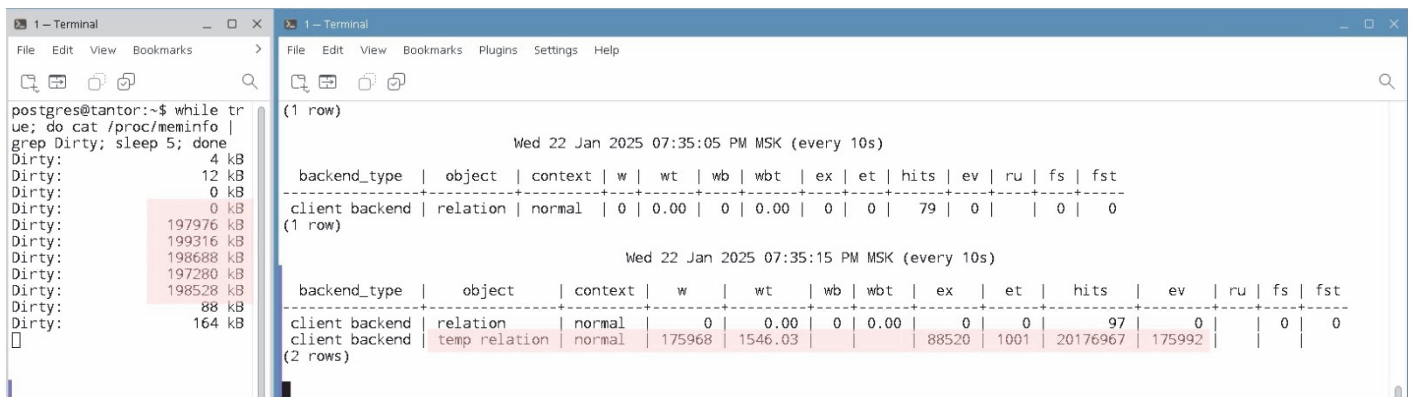
"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
  select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
  \ watch 10
```

3) In the third terminal in psql run the commands:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
create temp table temp1 (id integer);
\timing on\\
insert into temp1 select * from generate_series(1, 20000000);
explain (analyze, timing off, buffers) select * from temp1;
drop table temp1;
\timing off\\
```

```
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

CREATE TABLE
Timing is on.
INSERT 0 20000000
Time: 19265.840 ms (00:19.266)
QUERY PLAN
-------------------------------------------------------------------------------------------------
Seq Scan on temp1 (cost=0.00..314160.80 rows=22566480 width=4) (actual rows=20000000 loops=1)
Buffers: local read=88496 dirtied=88496 written=88494
I/O Timings: local read= 663.960 write= 794.682
Planning:
Buffers: shared hit=4
Planning Time: 0.048 ms
Execution Time: 5923.242 ms
(7 rows)

Time: 5923.524 ms (00:05.924)
Timing is off.
DROP TABLE
Time: 191.131 ms
```



4) In the third terminal, restart psql and change the local cache size:

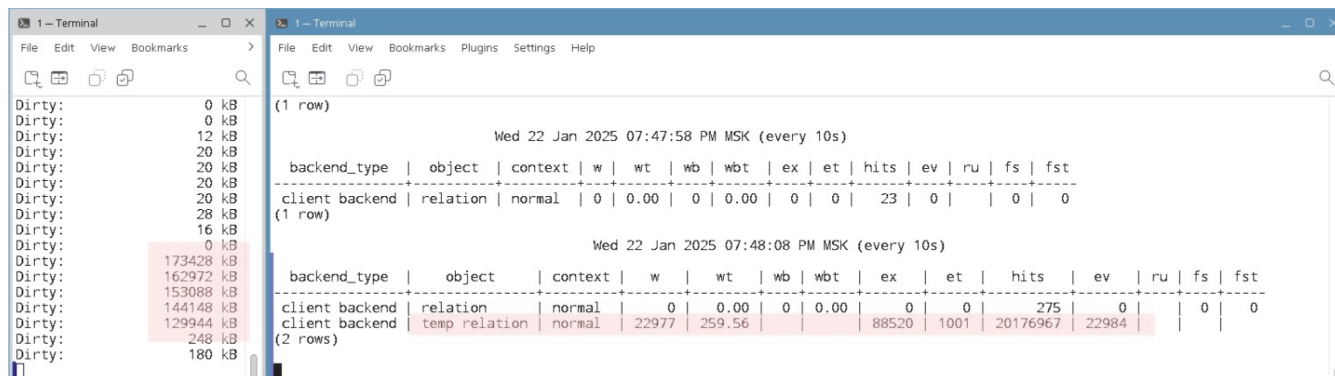
```
postgres=# \q
postgres@tantor:~$ psql
Type "help" for help.
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

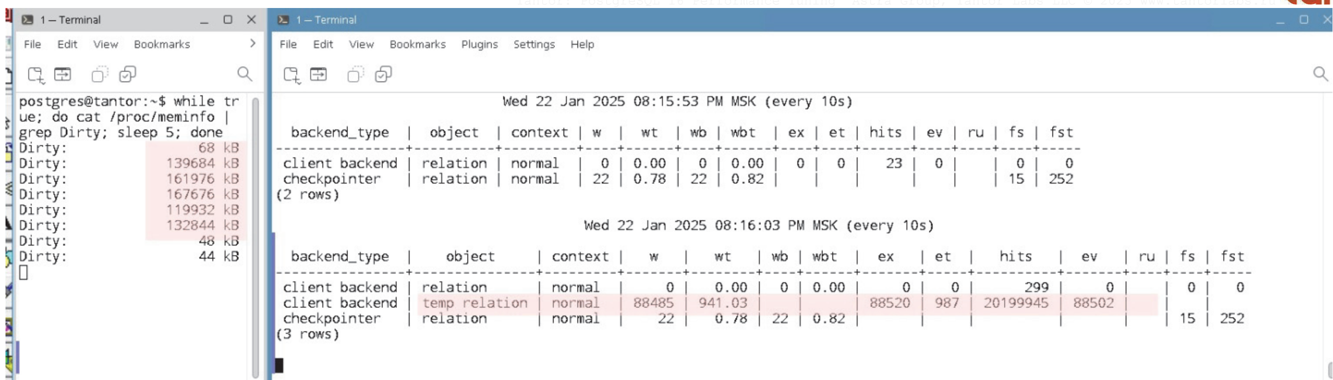
```
postgres=#
show temp_buffers;
set temp_buffers=' 512MB ';
temp_buffer
--------------
 8 MB
(1 row )
SET
```

5) In the third terminal in psql , repeat the test commands:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
create temp table temp1 (id integer);
\timing on\\
insert into temp1 select * from generate_series(1, 20000000);
explain (analyze, timing off, buffers) select * from temp1;
select pg_size_pretty(pg_table_size('temp1'));
drop table temp1;
\timing off\\
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

CREATE TABLE
Timing is on.
INSERT 0 20000000
Time: 19858.309 ms (00:19.858)
QUERY PLAN
------------------------------------------------------------------------------------------------
Seq Scan on temp1 (cost=0.00..314160.80 rows=22566480 width=4) (actual rows=20000000 loops=1)
Buffers: local hit=22978 read=65518 dirtied=65518 written=65508
I/O Timings: local read= 581.173 write= 693.463
Planning:
Buffers: shared hit=22
Planning Time: 0.126 ms
Execution Time: 5523.708 ms
(7 rows)

Time: 5524.133 ms (00:05.524)
pg_size_pretty
----------------
          692 MB
(1 row)
DROP TABLE
Time: 90.952 ms
Timing is off.
```



"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru



Temporary objects use the local cache of the server process.

The table size is `692 MB` . With a local buffer pool size of **`8 MB , writes = 175968`** blocks = 1374 MB were written . This is equal to the double table size minus `8 MB` .

Buffers are a reflection of the file contents and the file is created immediately when the temporary table is created. **The value in the `extends field = 88520` = 691MB corresponds to the table size and is the same in both tests.**

With a local buffer pool size `of 512 MB,` after the **`insert command,`** there were **`writes`** (passed to the Linux cache and later written to disk, which could be observed in the first terminal **`cat / proc / meminfo | grep Dirty`** - the number of dirty pages decreased, which means physical writing to disk) 22977 = 179 MB. `512 MB` + 179 MB = 691 MB, which corresponds to the size of the table. After **`explain analyze`** in total **`wrotes`** 88485 blocks = 691 MB.

The execution time of commands did not change much. Inserting rows with a buffer size of 8 MB was slightly faster than with a buffer size of 512 MB. In the test, rows were inserted en masse and read en masse once. In this mode, the cache size does not matter, this is a mode of working with rows similar to a buffer ring and it is effective. If there is multiple random access to the rows of a temporary table, the cache size will matter. Since only one session works with a temporary table, the "multiplicity" is less compared to regular tables accessible to hundreds of sessions. The typical mode of working with temporary tables is: data is inserted en masse, updated en masse, and the result is read. In this mode, even indexes are not needed. For this mode, the cache size does not matter. However, applications can use temporary tables in other modes, so when choosing the **`size of temp_buffer`** It is worth using commands that are close to those used by the application.

By files of temporary objects **`fsyncs`** and **`writebacks`** are not executed, as indicated by the empty values in the line with **`object=' temp relation '`** .

6) Look at the performance indicators for a non-journalized table:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
create unlogged table temp1 (id integer);
\timing on\\
insert into temp1 select * from generate_series(1, 20000000);
explain (analyze, timing off, buffers) select * from temp1;
select pg_size_pretty(pg_table_size('temp1'));
drop table temp1;
\timing off\\
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)
CREATE TABLE
Timing is on.
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
INSERT 0 20000000
Time: 20868.504 ms (00:20.869)
QUERY PLAN
---------------------------------------------------------------------------------------------
Seq Scan on temp1 (cost=0.00..314160.80 rows=22566480 width=4) (actual rows=20000000 loops=1)
Buffers: shared hit=16344 read=72152 dirtied=74086 written=72120
I/O Timings: shared read=1011.583 write=633.551
Planning:
Buffers: shared hit=1 read=3
I/O Timings: shared read=208.287
Planning Time: 208.366 ms
Execution Time: 5935.770 ms
(8 rows)

Time: 6144.520 ms (00:06.145)
DROP TABLE
Time: 260.237 ms
Timing is off .
```

The execution time of the commands is similar to the execution time when working with a temporary table and an **8 MB buffer** . When selecting from the table, a buffer ring was used :

```
backend_type | object |context | w | wt | wb | wbt | ex | et | hits | ev | ru |fs |fst
-----------------+--------+--------+-----+------+------+---------+-----+----+---------+------+-----+---+---
autovacuum worker|relation|normal | 0| 0.00| 0 | 0.00 | 0| 0| 13862 | 0 | | 0 | 0
client backend |relation| bulkread |72120|633.55| 0 | 0.00 | | | 16344 | 32 |72120| |
client backend |relation|normal |66840|684.00| 0 | 0.00 |88520|1396| 20177108 |73036 | | 0 | 0
background writer|relation|normal | 7291|118.17| 7296 | 4993.65 | | | | | | 0 | 0
(4 rows)
```

7) Why wasn't the write buffer ring used? Rewrite the commands to use the bulk write ring :

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
\timing on\\
create unlogged table temp1 as select id from generate_series(1, 20000000)
x(id);
explain (analyze, timing off, buffers) select * from temp1;
select pg_size_pretty(pg_table_size('temp1'));
drop table temp1;
\timing off\\
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

Timing is on.
SELECT 20000000
Time: 16304.857 ms (00:16.305)
QUERY PLAN
---------------------------------------------------------------------------------------------
Seq Scan on temp1 (cost=0.00..314217.60 rows=22570560 width=4) (actual rows=20000000 loops=1)
Buffers: shared hit=2048 read=86464 dirtied=86464 written=86432
I/O Timings: shared read=635.782 write=752.176
Planning:
Buffers: shared hit=3 read=1
I/O Timings: shared read=0.010
Planning Time: 0.097 ms
Execution Time: 5793.216 ms
(8 rows)

Time: 5793.607 ms (00:05.794)
pg_size_pretty
----------------
692 MB
(1 row)
```

```
Time: 14.803 ms
DROP TABLE
Time: 127.085 ms
Timing is off.
```

Teams steel to be carried out **faster** .

```
backend_type | object | context | w | wt |wb|wbt | ex |et | hits |ev | ru |fs |fst
-----------------+--------+--------+-----+------+--+----+------+----+------+---+------+---+---
autovacuum worker|relation|normal | 0| 0.00| 0|0.00| 0 | 0 | 2447 | 0 | | 0 | 0
client backend |relation| bulkread |86432|752.18| 0|0.00| | | 2048 | 0 |86432 | |
client backend |relation| bulkwrite |86464|885.37| 0|0.00|88512 |508 | 87107 | 0 |86464 | |
client backend |relation|normal | 0| 0.00| 0|0.00| 0 | 0 | 100 | 0 | | 0 | 0
(4 rows)
```

Part 5. Example of analysis of statistics of vacuum command operation with a ring

1) Create table For test :

```
postgres=# drop table if exists test;
create table test(a int, b int) with (autovacuum_enabled = 'false');
insert into test select i, i from generate_series(1, 4500) i;
SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty) FILTER
(WHERE b.isdirty = true) as buffer_dirty from pg_buffercache b JOIN pg_class c ON
b.relfilenode = pg_relation_filenode(c.oid) AND b.reldatabase IN (0, (SELECT oid
FROM pg_database WHERE datname = current_database())) JOIN pg_namespace n ON
n.oid = c.relnamespace GROUP BY n.nspname, c.relname having relname='test';
NOTICE: table "test" does not exist, skipping
DROP TABLE
CREATE TABLE
INSERT 0 4500
nspname | relname | buffers | buffer_dirty
---------+---------+---------+--------------
public | test | 23 | 20
(1 row )
```

There are 23 blocks in the buffer cache for table files, 20 of which are dirty.

2) To remove table file blocks from the buffer cache, run the command:

```
postgres=# vacuum full test;
SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty) FILTER
(WHERE b.isdirty = true) as buffer_dirty from pg_buffercache b JOIN pg_class c ON
b.relfilenode = pg_relation_filenode(c.oid) AND b.reldatabase IN (0, (SELECT oid
FROM pg_database WHERE datname = current_database())) JOIN pg_namespace n ON
n.oid = c.relnamespace GROUP BY n.nspname, c.relname having relname='test';
VACUUM
nspname | relname | buffers | buffer_dirty
---------+---------+---------+--------------
(0 rows)
```

3) Table takes up 20 blocks :

```
postgres=# select pg_table_size('test')/8192 blocks;
select pg_relation_size('test')/8192 blocks;
blocks
--------
20
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
(1 row )
 blocks
--------
20
(1 row )
```

The table contains one main layer file, no vm or fsm files.

4) Reset statistics, vacuum the table, look at the statistics:

```
postgres=#
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('bgwriter');
vacuum (BUFFER_USAGE_LIMIT 128) test;
select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
SELECT n.nspname, c.relname, count(*) AS buffers, count(b.isdirty) FILTER
(WHERE b.isdirty = true) as buffer_dirty from pg_buffercache b JOIN pg_class c ON
b.relfilenode = pg_relation_filenode(c.oid) AND b.reldatabase IN (0, (SELECT oid
FROM pg_database WHERE datname = current_database())) JOIN pg_namespace n ON
n.oid = c.relnamespace GROUP BY n.nspname, c.relname having relname='test';
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

VACUUM
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
---------------+---------+---+------+----+------+----+----+------+----+----+----+----
client backend | normal | 0 | 0.00 | 0 | 0.00 | 4 | 0 | 828 | 0 | | 0 | 0
client backend | vacuum | 4 | 0.03 | 0 | 0.00 | 0 | 0 | 0 | 0 | 4 | |
(2 rows)
nspname | relname | buffers | buffer_dirty
---------+---------+---------+--------------
 public | test | 20 | 20
(1 row )
```

When using the BUFFER_USAGE_LIMIT parameter The buffer ring is used even with small tables (less than 1/4 of the buffer cache size). Without the BUFFER_USAGE_LIMIT parameter , the buffer ring would not be used because the table is small. If BUFFER_USAGE_LIMIT = 0 , the buffer ring will not be used with tables of any size.

extends = 4 means that 4 blocks were added to some files by the **vacuum command** , or perhaps the files did not even exist and they were created.

context = vacuum means that the **vacuum command** used the buffer ring.

reuses= 4 means that 4 blocks were added to the ring.

writes = 4 means that 4 blocks were sent for writing.

hits =828 means that the **vacuum command** accessed 828 buffers in the buffer cache. These are blocks of system catalog objects. It is possible that some of the buffers were accessed more than once.

20 were read into the buffer cache table blocks and they were changed. Loading blocks into the buffer cache (" hits " present, " miss " absent) , as well as block contamination (" reads " is present, " dirtiness " is absent) in the pg_stat_io view is not reflected.

This is something to take into account when testing and analyzing test results.

5) Why 4 blocks, there are 20 blocks in the table? Look how many blocks there are in all the files in the table:

```
postgres=#
select pg_table_size('test')/8192 blocks;
select pg_relation_size('test')/8192 blocks;
select pg_relation_size('test',' main ')/8192 blocks;
select pg_relation_size('test','vm')/8192 blocks;
select pg_relation_size('test','fsm')/8192 blocks;
blocks
--------
24
(1 row)

blocks
--------
      20
(1 row)

blocks
--------
      20
(1 row)

blocks
--------
1
(1 row)

blocks
--------
3
(1 row)
```

After the **vacuum command** the size of the table files became 24 blocks. 4 blocks refer to the files of the vm and fsm layers. The **vacuum command** created these files.

What is the difference between `evictions` and `reused` when working with rings?

If the block is in a buffer belonging to a ring and the block is replaced by another block in the buffer of the same ring, then the block replacement is taken into account in the `reused statistics` of this type (`context`) of the ring. In this case, unoccupied blocks were added to the ring.

If a table block were pinned by another server process, then pinning the buffer to the ring would increment `the evictions statistic` , not `the reused statistic` . Also, replacing a block in a ring buffer with another block of the same table would count as `evictions` in the row that pertains to the ring (`context = vacuum`).

6) Delete table :

```
postgres=# drop table test;
DROP TABLE
```

7) Perform commands :

```
postgres=# checkpoint;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
select * from pg_stat_bgwriter\gx

CHECKPOINT
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+----+------+----+------+----+----+------+----+----+----+-----
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 4906 | 0 | | 0 | 0
client backend | normal | 0 | 0.00 | 0 | 0.00 | 4 | 0 | 919 | 0 | | 0 | 0
client backend | vacuum | 4 | 0.05 | 0 | 0.00 | 0 | 0 | 0 | 0 | 4 | |
checkpointer | normal | 39 | 0.44 | 39 | 1.12 | | | | | | 17 | 41
(4 rows)

-[ RECORD 1 ]---------+---
checkpoints_timed | 0
checkpoints_req | 1
checkpoint_write_time | 9
checkpoint_sync_time | 45
buffers_checkpoint | 39
buffers_clean | 0
maxwritten_clean | 0
buffers_backend | 40
buffers_backend_fsync | 0
buffers_alloc | 20
```

Values of statistics in pg_stat_bgwriter differ from the values in pg_stat_io .

Part 6. bgwriter operation and statistics comparison in pg_stat_bgwriter and pg_st_at_io views

1) Create files for a more complex test, closer to real loads:

```
postgres@tantor:~$ mcedit initcre.sql

DO
$$
begin
for i in 1..10000 by 100 loop
for j in 0..99 loop
execute concat('drop table if exists test.film_summary',i+j);
end loop;
commit;
end loop;
end;
$$
LANGUAGE plpgsql;
drop schema if exists test cascade;
create schema test;
select format('create table test.film_summary%s (film_id int, title int,
release_year int) with (autovacuum_enabled=off);', g.id) from generate_series(1,
10) as g(id)
```

```
\ gexec
```

The script deletes 100,000 tables from previous tests used in practices, if any, and creates 10 tables.

2) Create the **initdeleven .sql script file** :

```
postgres@tantor:~$ mcedit initdeleven.sql

select format('delete from test.film_summary%s where int8 mod (( ctid
::text::point)[ 0 ]::bigint ,2 ) = 1 ', g.id) from generate_series(1, 10) as
g(id);
\ gexec
```

The script deletes all lines in odd numbers blocks . 5 million rows in 10 tables ~3 minutes.

3) Create the **initdelodd .sql script file** :

```
postgres@tantor:~$ mcedit initdelodd.sql

select format('delete from test.film_summary%s where int8 mod (( ctid
::text::point)[ 0 ]::bigint ,2 ) = 0 ;', g.id) from generate_series(1, 10) as
g(id);
\ gexec
```

The script deletes all lines in even numbers blocks .

4) Create the **initins .sql script file** :

```
postgres@tantor:~$ mcedit initins.sql

select format('insert into test.film_summary%s select i, i , i from
generate_series(0, 10000000-1) as i;', g.id) from generate_series(1, 10) as g(id)
\ gexec
```

The script inserts rows into tables.

5) Create the **initsel .sql script file** :

```
postgres@tantor:~$ mcedit initsel.sql

select format('select * from test.film_summary%s where film_id = 0;', g.id) from
generate_series(1, 10) as g(id)
\ gexec
```

psql in the first terminal and the commands:

```
postgres=#
select backend_type, context, writes w, round(write_time::numeric) wt,
writebacks wb, round(writeback_time::numeric) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
\ watch 1
```

print statistics from the **pg_stat_io view once per** second:

```
  backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------------+---------+---+----+----+----+----+----+-----+----+----+----+----
 client backend | normal | 0 | 0 | 0 | 0 | 0 | 0 | 16 | 0 | | 0 | 0
(1 row )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

7) Run the commands:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c " select pg_stat_reset_shared('bgwriter');"
psql -f initcre.sql 2> /dev/null > /dev/null
time psql -f initins.sql > /dev/null
psql -c "select count(*) from test.film_summary1"
time psql -f initsel.sql > /dev/null
```

```
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

real 3m58 .679s
user 0m0.010s
sys 0m0.000s
count
----------
10000000
(1 row)

real 1m27 .860s
user 0m0.011s
sys 0 m 0.000 s
```

While the script is running, watch the work in the first terminal background writer :

```
  backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
-----------------+---------+--------+---------+--------+-------+--------+------+-----------+--------+----+----+------
autovacuum worker | normal | 3278 | 32.71 | 0 | 0 | 0 | 0 | 6524 | 3300 | | 0 | 0
client backend | normal | 405512 | 4009.88 | 0 | 0 | 540710 | 6994 | 101113340 | 525526 | | 0 | 0
background writer | normal | 112490 | 1289.38 | 112512 | 17299 | | | | | | 0 | 0
checkpointer | normal | 4387 | 38.64 | 4392 | 143 | | | | | | 92 | 17413
(4 rows )
```

bgwriter is quite active. Most of the writes and writebacks are done when running the initcre.sql and **initins .sql scripts .** is performed by this process.

8) Run the command in the second terminal:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c " select pg_stat_reset_shared('bgwriter');"
time psql -f initdeleven.sql > /dev/null
```

```
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

real 3m17 .525s
user 0m0.010s
sys 0 m 0.000 s
```

The script deletes all rows in even blocks of table files.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

9) While the script for deleting even lines is running, the first terminal will display the following results:

```
backend_type | context | w | wt | wb | wbt| ex | et | hits | ev | ru | fs | fst
----------+----------+-------+-----+-------+------+----+-----+----------+-------+-------+----+------
autovacuum worker | normal | 151 | 1 | 0 | 0 | 0 | 0 | 4113 | 799 | | 0 | 0
client backend | bulkread | 32438 | 380 | 0 | 0 | | | 0 | 53955 | 54155 | |
client backend | normal | 0 | 0 | 0 | 0 | 0 | 0 | 10000278 | 81 | | 0 | 0
background writer | normal | 22679 | 230 | 22656 | 3490 | | | | | | 0 | 0
checkpointer | normal | 600 | 5 | 576 | 29 | | | | | | 5 | 2458
(5 rows)
```

The number of fsyncs per file is small.

10) The checkpoint process wrote few buffers. The table size is more than 1/4 of the buffer cache:

```
postgres@tantor:~$ psql -c "select pg_size_pretty(pg_relation_size('test.film_summary9'));"
pg_size_pretty
----------------
422 MB
(1 row)
```

The buffer ring is used for fetching. If rows are changing, the BULKREAD ring is not used, and the BULKWRITE ring is used with bulk insert commands.

From the documentation for the pg_stat_io.fsync s column : " The number of fsync calls . **Calls are tracked only in the normal context." That is, fsyncs** when working with buffer **rings are counted and accounted for in the normal** context of the pg_stat_io view . Reason :

"the number of backend fsyncs doesn't say anything about the efficiency of the BufferAccessStrategy. And counting both fsyncs done in IOCONTEXT_NORMAL and IOCONTEXT_[BULKREAD, BULKWRITE, VACUUM] under IOCONTEXT_NORMAL is likely clearer when investigating the number of backend fsyncs."

11) Run queries on the views:

```
postgres@tantor:~$ psql -c "select backend_type, context, writes w, round(write_time::numeric) wt, writebacks wb, round(writeback_time::numeric) wbt, extends ex, round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs, round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or evictions>0 or reuses>0 or hits>0;"
psql -c "select checkpoint_write_time ckpt_write_t, checkpoint_sync_time ckpt_sync_t, buffers_checkpoint ckpt_buffers,buffers_clean bgwr_buffers, maxwritten_clean, buffers_backend, buffers_alloc from pg_stat_bgwriter;"
```

```
backend_type |context | w | wt | wb | wbt |ex|et| hits | ev | ru | fs | fst
----------------+--------+------+----+-------+-------+--+--+----------+--------+--------+----+-------
autovacuum worker|normal | 2416| 24| 0 | 0 | 0| 0| 40928 | 3280 | | 0 | 0
client backend |bulkread| 177573 |2226| 0 | 0 | | | 0 | 269768 | 270783 | |
client backend |normal | 48 | 1| 0 | 0 | 0| 0| 50001666 | 227 | | 0 | 0
background writer|normal | 94562 |1053| 94528 | 29283 | | | | | | 0 | 0
checkpointer |normal | 5056| 75| 5163 | 725 | | | | | | 29 | 16335
(5 rows)

ckpt_write_t |ckpt_sync_t |ckpt_buffers |bgwr_buffers |maxwritten_clean|buffers_backend |buffers_alloc
------------+-----------+------------+-------------+----------------+----------------+-------------
495800 | 16408 | 5163 | 94562 | 851 | 180037 | 273312
(1 row )
```

Statistics that are present in both views are highlighted in color.

maxwritten_clean - How many times background writer **paused** flushing dirty pages to disk because it had written too many buffers. No such statistics **are available** in **pg_stat_io .**

If there are no free buffers in the buffer cache (this is normal), then buffers_alloc corresponds to evictions . If there are free buffers, then they are taken from the list of free ones, statistics increases, and evictions does not increase.

This practice point is useful if you have **`used the pg_stat_bgwriter view .`** for monitoring and want to understand how to monitor statistics starting from version 17.

7) Run a test that reads all table blocks, updates the committed bit on rows, and thus dirtys the blocks:

```
postgres@tantor:~$ time psql -f initsel.sql > /dev/null

real 1m3 .605s
user 0m0.011s
sys 0m0.000s
```

```
backend_type |context | w | wt | wb | wbt |ex|et| hits | ev | ru | fs | fst
--------+--------+--------+------+--------+-------+--+--+---------+--------+--------+----+------
autovacuum worker|normal | 3961 | 38 | 0 | 0 | 0| 0| 19684 | 4876 | | 0 | 0
client backend |bulkread| 256022 | 3211 | 0 | 0 | | | 0 | 360295 | 362282 | |
client backend |normal | 44 | 1 | 0 | 0 | 0| 0|50001250 | 319 | | 0 | 0
background worker | bulkread | 157499 | 2195 | 0 | 0 | | | 0 | 178032 | 180486 | |
background worker|normal | 174 | 3 | 0 | 0 | 0| 0| 3112 | 203 | | 0 | 0
background writer|normal | 121776 | 1439 | 121792 | 52859 | | | | | | 0 | 0
checkpointer |normal | 4621 | 50 | 4645 | 458 | | | | | | 40 | 23091
(7 rows)
```

8) Try it on one's own answer on questions :
How were engaged processes **`background worker`** V `context=` **`bulkread`** ?
How much ? processes **`background worker`** worked V `context=` **`bulkread`** ?
You can answer this yourself. The effort you put into finding the answer will most effectively help you learn to diagnose problems. The time you spend looking for the answer is not important. The effectiveness of your work does not depend on how quickly you find the answer.

Part 7. Running bgwriter on 128MB and 1GB buffer cache

`buffer cache size to 1GB will affect the pg_stat_io indicators . Then the` buffer pool will be increased and the test will be repeated. Run the commands:

```
postgres@tantor:~$
psql -c "alter system set shared_buffers='1GB'"
sudo restart
```

2) In the first terminal the connection will be closed. Restart psql and run the commands:

```
FATAL: terminating connection due to administrator command
server closed the connection unexpectedly
This probably means the server terminated abnormally
before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.

!?> \q
postgres@tantor:~$ psql

postgres=# select backend_type, context, writes w, round(write_time::numeric)
wt, writebacks wb, round(writeback_time::numeric) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
 \ watch 1
```

3) In the second terminal, execute the commands:

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c " select pg_stat_reset_shared('bgwriter');"
psql -f initcre.sql 2> /dev/null > /dev/null
time psql -f initins.sql > /dev/null
psql -c "select count(*) from test.film_summary1"
time psql -f initsel.sql > /dev/null
```

```
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

real 3m31 .435s
user 0m0.010s
sys 0m0.000s

count
----------
10000000
(1 row)

real 1m21 .983s
user 0m0.010s
sys 0 m 0.000 s
```

The time has decreased slightly. For single sequential inserts and reads, the cache size does not matter.

4) While the script is running, observe in the first terminal which processes are performing input/output:

```
backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+---------+--------+-----+--------+-------+--------+------+-----------+--------+---+---+------
autovacuum worker | normal | 0 | 0 | 0 | 0 | 0 | 0 | 6566 | 2468 | | 0 | 0
client backend | normal | 0 | 0 | 0 | 0 | 540710 | 8056 | 101082280 | 410689 | | 0 | 0
checkpointer | normal | 535651 | 5273 | 535641 | 45090 | | | | | | 82 | 8434
(3 rows)
```

Unlike the previous part of the practice, where the same scripts were executed on a buffer cache of 128 MB, now there is no bgwriter activity. All writes and writebacks when running the `initcre.sql` **and** initins `.sql scripts` executed by `the checkpointer process .`

5) Do it commands :

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c " select pg_stat_reset_shared('bgwriter');"
time psql -f initdeleven.sql > /dev/null
```

```
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

real 3m19 .089s
user 0m0.011s
sys 0 m 0.000 s
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

There is no difference in execution time compared to the previous part (in which the buffer cache size was 128 MB).

6) While the script for deleting even lines is running, the first terminal will display the following results:

```
backend_type |context | w | wt | wb | wbt |ex|et| hits | ev | ru | fs | fst
--------+--------+-------+------+--------+-------+--+--+---------+--------+--------+----+----
autovacuum worker|normal | 0 | 0 | 0 | 0 | 0| 0| 12280 | 0 | | 0 | 0
client backend |bulkread| 0 | 0 | 0 | 0 | | | 0 | 270380 | 270170 | |
client backend |normal | 0 | 0 | 0 | 0 | 0| 0|50000504 | 56 | | 0 | 0
checkpointer |normal |304518 | 3291 | 304523 | 40747 | | | | | | 25 | 400
(4 rows)
```

The number of fsyncs per file is also small, just like in the previous part.
There is no activity in the bgwriter process, all writing was done by checkpointer.

7) Run a test that reads all table blocks, updates the committed bit, and thus dirtys the buffers:

```
postgres@tantor:~$
psql -c "select pg_stat_reset_shared('io');"
psql -c " select pg_stat_reset_shared('bgwriter');"
time psql -f initsel.sql > /dev/null

real 1m8 .394s
user 0m0.010s
sys 0m0.002s
```

```
backend_type | context | w | wt | wb | wbt |ex |et | hits | ev | ru | fs | fst
-------------------+----------+--------+------+--------+-------+---+---+------+--------+--------+----+----
autovacuum worker | normal | 0 | 0 | 0 | 0 | 0 | 0 | 2456 | 0 | | 0 | 0
client backend | bulkread | 118 | 2 | 0 | 0 | | | 0 | 91549 | 91565 | |
client backend | normal | 0 | 0 | 0 | 0 | 0 | 0 | 441 | 18 | | 0 | 0
background worker | bulkread | 179 | 5 | 0 | 0 | | | 0 | 178752 | 178684 | |
background worker | normal | 0 | 0 | 0 | 0 | 0 | 0 | 3302 | 0 | | 0 | 0
background writer | normal | 100 | 1 | 64 | 41 | | | | | | 0 | 0
checkpointer | normal | 230549 | 3002 | 230531 | 36715 | | | | | | 14 | 369
(7 rows )
```

In none of the tests was there any bgwriter activity , unlike in the previous part of the practice.

8) You are familiar with the architecture of the instance. Try to answer the questions yourself:
a) why bgwriter didn't show any activity?
b) what configuration parameter should be changed so that bgwriter works as actively as on a 128MB buffer cache?
Putting effort into finding answers to these questions will help you find simple solutions that don't create side effects.

9) Run the commands to return the configuration parameters:

```
postgres@tantor:~$
psql -c "alter system reset shared_buffers";
sudo restart
```

10) Look at the statistics in the tables:

```
postgres=# select schemaname||'.'||relname name, heap_blks_read tabread,
heap_blks_hit tabhit, idx_blks_read idxread, idx_blks_hit idxhit, toast_blks_read
toastread, toast_blks_hit toasthit from pg_statio_all_tables order by 2 desc
limit 15;
 name | tabread | tabhit | idxread | idxhit | toastread | toasthit
-------------------------+---------+----------+---------+----------+-----------+----------
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
test.film_summary1 | 162165 | 15162143 | | | |
test.film_summary2 | 132926 | 15108088 | | | |
test.film_summary8 | 108110 | 15108088 | | | |
test.film_summary9 | 108110 | 15108088 | | | |
test.film_summary6 | 108110 | 15108088 | | | |
test.film_summary5 | 108110 | 15108088 | | | |
test.film_summary3 | 108110 | 15108088 | | | |
test.film_summary7 | 108110 | 15108088 | | | |
test.film_summary4 | 108110 | 15108088 | | | |
test.film_summary10 | 89867 | 15126331 | | | |
pg_catalog.pg_attribute | 50048 | 7792567 | 30459 | 15974323 | |
pg_catalog.pg_class | 38505 | 25246260 | 12697 | 12929351 | |
pg_catalog.pg_statistic | 6777 | 136160 | 684 | 1710024 | 52 | 346
pg_catalog.pg_type | 4324 | 1035423 | 6194 | 3487848 | 0 | 0
pg_catalog.pg_depend | 3654 | 3894142 | 13502 | 12385733 | |
(15 rows)
```

11) Request to view statistics on indexes:

```
postgres=# select schemaname||'.'||relname table, indexrelname index, idx_blks_read idxread, idx_blks_hit idxhit from pg_statio_all_indexes order by 4 desc limit 5;
table | index | idxread | idxhit
--------------------------+---------------------------------+---------+---------
pg_catalog.pg_attribute | pg_attribute_relid_attnum_index | 14157 | 12490534
pg_catalog.pg_class | pg_class_oid_index | 3220 | 7968689
pg_catalog.pg_depend | pg_depend_depender_index | 7466 | 7349793
pg_catalog.pg_depend | pg_depend_reference_index | 6036 | 5035940
pg_catalog.pg_class | pg_class_relname_nsp_index | 7815 | 3985553
(5 rows)
```

" **idx _ blks _ hit** " block was in the buffer cache.
" **idx _ blks _ read** " block was not in the buffer cache. The process had to clear the buffer (if the free block list is empty), load the block into the cleared buffer, and only then read its contents.

Statistics on tables and indexes are useful because they allow you to identify "hot" objects - those that are accessed most often.

12) Remove the test schema with test tables to free up disk space:

```
postgres=# drop schema test cascade;
NOTICE: drop cascades to 10 other objects
DETAIL: drop cascades to table test.film_summary1
drop cascades to table test.film_summary2
drop cascades to table test.film_summary3
drop cascades to table test.film_summary4
drop cascades to table test.film_summary5
drop cascades to table test.film_summary6
drop cascades to table test.film_summary7
drop cascades to table test.film_summary8
drop cascades to table test.film_summary9
drop cascades to table test.film_summary10
DROP SCHEMA
```

16) Look at examples of queries to table statistics :

```
postgres=# select schemaname||'.'||relname name, seq_scan, idx_scan, idx_tup_fetch, autovacuum_count, autoanalyze_count from pg_stat_all_tables where idx_scan is not null order by 3 desc limit 5;
name | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
--------------------------+----------+----------+---------------+------------------+-------------------
pg_catalog.pg_class | 31253 | 4330223 | 3274290 | 83 | 106
pg_catalog.pg_attribute | 1565 | 3036625 | 9054711 | 90 | 105
pg_catalog.pg_depend | 3 | 2389005 | 2722183 | 83 | 103
pg_catalog.pg_type | 3 | 1093705 | 326604 | 80 | 104
pg_catalog.pg_statistic | 2 | 823316 | 43138 | 9 | 0
(5 rows)
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres=# select relname name, n_tup_ins ins, n_tup_upd upd, n_tup_del del,
n_tup_hot_upd hot_upd, n_tup_newpage_upd newblock, n_live_tup live, n_dead_tup
dead, n_ins_since_vacuum sv, n_mod_since_analyze sa from pg_stat_all_tables where
idx_scan is not null order by 3 desc limit 5;
 name | ins | upd | del | hot_upd | newblock | live | dead | sv | sa
--------------+---------+--------+---------+---------+----------+------+------+-----+-------
 pg_class | 459654 | 156877 | 476171 | 144372 | 9275 | 447 | 40 | 30 | 70
 pg_statistic | 30034 | 12473 | 30032 | 12160 | 313 | 424 | 59 | 0 | 72539
 pg_shdepend | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
 pg_attribute | 3065349 | 0 | 3175464 | 0 | 0 | 3340 | 360 | 270 | 90
 pg_type | 306888 | 0 | 317898 | 0 | 0 | 643 | 80 | 60 | 20
 (5 rows)
```

```
postgres=# select schemaname||'.'||relname name, indexrelname, idx_scan,
round(extract('epoch' from clock_timestamp() -
  last_idx_scan)) scan_sec, idx_tup_read, idx_tup_fetch from pg_stat_all_indexes
order by 3 desc limit 5;
 name | indexrelname | idx_scan | scan_sec | idx_tup_read | idx_tup_fetch
------------------------+-------------------------------------+---------+----------+--------------+---------------
 pg_catalog.pg_class | pg_class_oid_index | 3493066 | 3 | 3051762 | 3032453
 pg_catalog.pg_attribute | pg_attribute_relid_attnum_index | 3036645 | 3 | 9054760 | 9054759
 pg_catalog.pg_depend | pg_depend_depender_index | 1591566 | 598 | 1909239 | 1909239
 pg_catalog.pg_class | pg_class_relname_nsp_index | 837469 | 3 | 244094 | 241270
 pg_catalog.pg_statistic | pg_statistic_relid_att_inh_index | 823325 | 130 | 43329 | 43138
 (5 rows)
```

`idx_tup_read` The number of index entries returned by scans of this index. The value is incremented each time an index entry is read.

`idx_tup_fetch` The number of actual table rows fetched by an index record during a simple scan of that index (Index Scan).

13) Search for tables that have not been accessed for a long time:

```
postgres=# select schemaname schema, relname table,
pg_size_pretty(pg_total_relation_size(relid)) size, seq_scan, idx_scan,
last_seq_scan::time seq_scan_t, last_idx_scan::time idx_scan_t from
pg_stat_all_tables where schemaname not in
('pg_catalog','information_schema','pg_toast') order by 4, 5;
 schema | table | size | seq_scan | idx_scan | seq_scan_t | idx_scan_t
----------+---------+------------+----------+----------+-----------------+-----------
 public | pgbench_history | 0 bytes | 0 | | |
 public | pgbench_branches | 56 kB | 1 | 0 | 13:57:07.951584 |
 public | pgbench_tellers | 56 kB | 1 | 0 | 13:57:07.951584 |
 public | shmem_reference | 16 kB | 1 | | 13:02:03.21597 |
 public | test | 192 kB | 1 | | 13:42:12.078525 |
 columnar | options | 8192 bytes | 2 | 0 | 21:32:08.85896 |
 columnar | chunk_group | 8192 bytes | 2 | 0 | 21:32:08.85896 |
 columnar | chunk | 16 kB | 2 | 0 | 21:32:08.85896 |
 public | pgbench_accounts | 15 MB | 2 | 0 | 13:57:07.951584 |
 columnar | stripe | 16 kB | 3 | 0 | 21:32:08.85896 |
 columnar | row_mask | 32 kB | 4 | 0 | 21:32:08.85896 |
 test | film_summary3 | 422 MB | 7 | | 20:12:27.735492 |
 test | film_summary4 | 422 MB | 7 | | 20:12:34.973333 |
 test | film_summary10 | 422 MB | 7 | | 20:13:19.345392 |
 test | film_summary6 | 422 MB | 7 | | 20:12:50.135028 |
 test | film_summary7 | 422 MB | 7 | | 20:12:58.123483 |
 test | film_summary8 | 422 MB | 7 | | 20:13:04.938543 |
 test | film_summary9 | 422 MB | 7 | | 20:13:12.069751 |
 test | film_summary5 | 422 MB | 7 | | 20:12:42.720495 |
 test | film_summary2 | 422 MB | 7 | | 20:12:21.332072 |
 test | film_summary1 | 422 MB | 10 | | 20:12:15.237606 |
 (21 rows)
```

14) Search for long unused indexes:

```
postgres=# select s.schemaname schema, s.relname table, indexrelname index,
  pg_size_pretty(pg_relation_size(s.indexrelid)) size, s.idx_scan scans,
s.last_idx_scan::time last_scan from pg_stat_all_indexes s join pg_index i on
s.indexrelid = i.indexrelid where s.schemaname not in
('information_schema','pg_toast') and not i.indisunique order by 5, 6;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
schema | table | index | size | scans | last_scan
------------+-----------+--------------------------------+-----------+---------+---------------
pg_catalog | pg_publication_rel | pg_publication_rel_prpubid_index | 8192 bytes | 0 |
pg_catalog | pg_constraint | pg_constraint_conparentid_index | 16 kB | 0 |
pg_catalog | pg_inherits | pg_inherits_parent_index | 8192 bytes | 0 |
pg_catalog | pg_trigger | pg_trigger_tgconstraint_index | 8192 bytes | 0 |
pg_catalog | pg_auth_members | pg_auth_members_grantor_index | 16 kB | 0 |
pg_catalog | pg_shdepend | pg_shdepend_reference_index | 8192 bytes | 0 |
pg_catalog | pg_class | pg_class_tblspc_relfilenode_index | 1008 kB | 0 |
pg_catalog | pg_constraint | pg_constraint_contypid_index | 16 kB | 0 |
pg_catalog | pg_constraint | pg_constraint_conname_nsp_index | 16 kB | 6 | 13:57:07.951584
pg_catalog | pg_index | pg_index_indrelid_index | 424 kB | 65381 | 00:42:46.840573
pg_catalog | pg_statistic_ext | pg_statistic_ext_relid_index | 8192 bytes | 186119 | 00:42:31.871778
pg_catalog | pg_shdepend | pg_shdepend_depender_index | 8192 bytes | 794253 | 23:50:14.82469
pg_catalog | pg_depend | pg_depend_reference_index | 3928 kB | 797566 | 23:50:14.846317
pg_catalog | pg_depend | pg_depend_depender_index | 4664 kB | 1591819 | 23:50:14.82469
(14 rows )
```

15) To check the correctness of the answer: the name of the configuration parameter from point 8 has 12 characters.

Part 8. Using the `pg_walinspect extension`

The extension appeared in version 15. The extension functions provide the same data as the `pg_waldump command line utility` . In this part of the practice, you will see whether it is convenient to work with the extension functions and what capabilities they have.

1) Install the extension and view the contents of the extension:

```
postgres=# create extension pg_walinspect;
\dx+ pg_walinspect
CREATE EXTENSION
Objects in extension "pg_walinspect"
Object description
-------------------------------------------------------------
function pg_get_wal_block_info(pg_lsn,pg_lsn,boolean)
function pg_get_wal_record_info(pg_lsn)
function pg_get_wal_records_info(pg_lsn,pg_lsn)
function pg_get_wal_stats (pg_lsn,pg_lsn,boolean)
(4 rows )
```

The extension consists of four functions.

2) Do it request :

```
postgres=# select "resource_manager/record_type" type, count,
round(count_percentage) "count%", record_size, round(record_size_percentage)
"size%", fpi_size, round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats ((select
'0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number * size from
pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF', false ) tantor
where count<>0 order by 8 desc;
type | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
-------------+------+--------+-----------+-------+---------+------+-------+--------
XLOG | 1814 | 48 | 92786 | 10 | 14798492 | 99 | 14891278 | 94
Heap2 | 1821 | 49 | 780143 | 88 | 7300 | 0 | 787443 | 5
Heap | 45 | 1 | 7101 | 1 | 55904 | 0 | 63005 | 0
Btree | 55 | 1 | 4264 | 0 | 36976 | 0 | 41240 | 0
Transaction | 1 | 0 | 887 | 0 | 0 | 0 | 887 | 0
Standby | 7 | 0 | 476 | 0 | 0 | 0 | 476 | 0
(6 rows)
```

Function `pg_get_wal_stats(start_lsn,end_lsn, per_record boolean)` shows statistics By magazine records between two LSNs.

`end_lsn` can be specified with a **maximum** of 'FFFFFFFF / FFFFFFFF'.

Full page images fpi occupy a significant place .

3) If the third parameter is true , then statistics are also issued for the journal entry type `record _ type` , and not the total for each `resource _ manager` . Repeat request having indicated `per_record= true` :

```
postgres=# select "resource_manager/record_type" type, count,
round(count_percentage) "count%", record_size, round(record_size_percentage)
"size%", fpi_size, round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats((select
'0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number * size from
pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF', true ) tantor
where count<>0 order by 8 desc;
type | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
-----------------------+-------+--------+-----------+-------+---------+------+-------------+--------
XLOG/ FPI_FOR_HINT | 1810 | 48 | 92310 | 10 | 14798492 | 99 | 14890802 | 94
Heap2/ PRUNE | 1808 | 48 | 779040 | 88 | 0 | 0 | 779040 | 5
Btree/INSERT_LEAF | 55 | 1 | 4264 | 0 | 36976 | 0 | 41240 | 0
Heap/INSERT | 13 | 0 | 3285 | 0 | 36752 | 0 | 40037 | 0
Heap/UPDATE | 3 | 0 | 261 | 0 | 19152 | 0 | 19413 | 0
Heap2/MULTI_INSERT | 13 | 0 | 1103 | 0 | 7300 | 0 | 8403 | 0
Heap/HOT_UPDATE | 15 | 0 | 1861 | 0 | 0 | 0 | 1861 | 0
Transaction/COMMIT | 1 | 0 | 887 | 0 | 0 | 0 | 887 | 0
Heap/UPDATE+INIT | 1 | 0 | 851 | 0 | 0 | 0 | 851 | 0
Heap/DELETE | 8 | 0 | 480 | 0 | 0 | 0 | 480 | 0
Standby/RUNNING_XACTS | 7 | 0 | 476 | 0 | 0 | 0 | 476 | 0
XLOG/CHECKPOINT_ONLINE | 2 | 0 | 296 | 0 | 0 | 0 | 296 | 0
Heap/LOCK | 4 | 0 | 240 | 0 | 0 | 0 | 240 | 0
XLOG/CHECKPOINT_SHUTDOWN | 1 | 0 | 148 | 0 | 0 | 0 | 148 | 0
Heap/INSERT+INIT | 1 | 0 | 123 | 0 | 0 | 0 | 123 | 0
XLOG/NEXTOID | 1 | 0 | 32 | 0 | 0 | 0 | 32 | 0
(16 rows )
```

More lines were output and `record _ type was added to the result` .

4) See the full list of resource managers:

```
postgres=# select * from pg_get_wal_resource_managers();
rm_id | rm_name | rm_builtin
-------+----------+------------
0 | XLOG | t
1 | Transaction | t
2 | Storage | t
3 | CLOG | t
4 | Database | t
5 | Tablespace | t
6 | MultiXact | t
7 | RelMap | t
8 | Standby | t
9 | Heap2 | t
10 | Heap | t
11 | Btree | t
12 | Hash | t
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
13 | Gin | t
14 | Gist | t
15 | Sequence | t
16 | SPGist | t
17 | BRIN | t
18 | CommitTs | t
19 | ReplicationOrigin | t
20 | Generic | t
21 | LogicalMessage | t
22 | Heap3 | t
(23 rows)
```

Extension libraries can register their own resource managers.

5) There is a view with log statistics. The view is not included in the extension, it is standard. See what is output in the view:

```
postgres =# select * from pg_stat_wal \ gx
-[ RECORD 1 ]----+-----------------------------
wal_records | 1304529873
wal_fpi | 9623757
wal_bytes | 166102694837
wal_buffers_full | 8083465
wal_write | 8387587
wal_sync | 293575
wal_write_time | 0
wal_sync_time | 0
stats_reset | 2025-01-01 01:01:33.035882+03
```

6) Statistics are accumulated from the moment the cluster is created or statistics are reset **using the pg_stat_reset_shared (' wal ') function .** Run commands :

```
postgres=#
select pg_stat_reset_shared('wal');
select * from pg_stat_wal\gx \\
pg_stat_reset_shared
----------------------

(1 row)

-[ RECORD 1 ]----+-----------
wal_records | 0
wal_fpi | 0
wal_bytes | 0
wal_buffers_full | 0
wal_write | 0
wal_sync | 0
wal_write_time | 0
wal_sync_time | 0
stats_reset | 2035-01-01 01:01:55.994752+03
```

7) Reduce the number of WAL files:

```
postgres @ tantor :~$
for (( i=0; i <= 60; i+=1 ));
do
psql -c "select pg_switch_wal();" > /dev/null 2> /dev/null
psql -c "checkpoint;" > /dev/null
done
```

8) see how many log files are left:

```
postgres@tantor:~$ ls $PGDATA/pg_wal
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
000000010000002500000D8 000000010000002500000D9 000000010000002500000DA
000000010000002500000DB 000000010000002500000DC archive_status
```

9) List the log files and the LSN of the start of each WAL file:

```
postgres=# select name, '0/0'::pg_lsn +
(pg_split_walfile_name(name(oleg))).segment_number * size min from pg_ls_waldir()
oleg order by 1;
name | min
--------------------------+-------------------
000000010000002500000DC | 00000025/DC000000
000000010000002500000DB | 00000025/DB000000
000000010000002500000DA | 00000025/DA000000
000000010000002500000D9 | 00000025/D9000000
000000010000002500000D8 | 00000025/D8000000
(5 rows)
```

10) Run the query with the `pg_get_wal_stats ( ) function that you ran earlier` :

```
postgres=#
select "resource_manager/record_type" type, count, round(count_percentage)
"count%", record_size, round(record_size_percentage) "size%", fpi_size,
round(fpi_size_percentage(tantor)) "fpi%", combined_size,
round(combined_size_percentage) "total%" from pg_get_wal_stats ((select
'0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number * size from
pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF', true ) tantor
where count<>0 order by 8 desc;
type | count | count% | record_size | size% | fpi_size | fpi% | combined_size | total%
-----------------------+-------+--------+------------+-------+----------+------+--------------+--------
XLOG/CHECKPOINT_ONLINE | 1 | 33 | 148 | 52 | 0 | 0 | 148 | 52
Standby/RUNNING_XACTS | 2 | 67 | 136 | 48 | 0 | 0 | 136 | 48
(2 rows )
```

There are five WAL files, four of which are completely empty. Only records for the last checkpoint are saved.

to see what is in the journal entries themselves. Look text functions `pg_get_wal_record s_info ()` :

```
postgres=# \ sf pg_get_wal_record s_info
CREATE OR REPLACE FUNCTION public.pg_get_wal_records_info( start_lsn pg_lsn,
end_lsn pg_lsn , OUT start_lsn pg_lsn, OUT end_lsn pg_lsn, OUT prev_lsn pg_lsn,
OUT xid xid, OUT resource_manager text, OUT record_type text, OUT record_length
integer, OUT main_data_length integer, OUT fpi_length integer, OUT description
text, OUT block_ref text )
RETURNS SETOF record
LANGUAGE c
PARALLEL SAFE STRICT
AS '$libdir/pg_walinspect', $function$pg_get_wal_records_info$function$
```

The function has two input parameters and many output parameters.

12) Use the `pg_get_wal_records_info function to find` the log `entries for the` end `of the checkpoint` :

```
postgres=# select resource_manager res, record_type, start_lsn, end_lsn,
prev_lsn, record_length len, left(description,20) from pg_get_wal_record s _info
((select '0/0'::pg_lsn + (pg_split_walfile_name(name(oleg))).segment_number *
size from pg_ls_waldir() oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF') tantor
where record_type like 'CHECKPOINT%' order by 3 limit 20;
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
res | record_type | start_lsn | end_lsn | prev_lsn | len | left
------------+-----+------------+-------------+------------+-----+---------------------
XLOG | CHECKPOINT_ONLINE | 25/D8000070 | 25/D8000108 | 25/D8000028 | 148 | redo 25/D8000028; tl
XLOG | CHECKPOINT_ONLINE | 25/D8001FE8 | 25/D8002098 | 25/D8001FA0 | 148 | redo 25/D8001FA0; tl
XLOG | CHECKPOINT_ONLINE | 25/D8007B90 | 25/D8007C28 | 25/D8007B48 | 148 | redo 25/D8007B48; tl
(3 rows)
```

There is no checkpoint start record. The checkpoint end record has an identifier with the letter combination `CHECKPOINT` .

13) Look at everything that the function can output for any journal entry:

```
postgres=# \pset xheader_width 40
Expanded header width is 40.
postgres=# select * from pg_get_wal_record s _info ((select '0/0'::pg_lsn +
(pg_split_walfile_name(name(oleg))).segment_number * size from pg_ls_waldir()
oleg order by name limit 1), 'FFFFFFFF/FFFFFFFF') tantor where record_type like
'CHECKP%' order by 3 limit 1\gx
-[ RECORD 1 ]----+-----------
start_lsn | 25/D8000070
end_lsn | 25/D8000108
prev_lsn | 25/D8000028
xid | 0
resource_manager | XLOG
record_type | CHECKPOINT_ONLINE
record_length | 148
main_data_length | 120
fpi_length | 0
description | redo 25/D8000028; tli 1; prev tli 1; fpw true; xid 361549; oid 942308;
multi 1; offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit
timestamp xid: 0/0; oldest running xid 361549; online
block_ref |
```

14) Look at the issued Recorded by `pg_waldump utility` :

```
postgres=# \! pg_waldump -s 25/D8000070 -e 25/D8000108
rmgr: XLOG len (rec/tot): 148/ 148 , tx: 0 , lsn: 25/D8000070 , prev
25/D8000028 , desc: CHECKPOINT_ONLINE redo 25/D8000028; tli 1; prev tli 1; fpw
true; xid 361549; oid 942308; multi 1; offset 0; oldest xid 723 in DB 1; oldest
multi 1 in DB 5; oldest/newest commit timestamp xid: 0/0; oldest running xid
361549; online
```

The function from the extension produces the same as the **`pg_waldump utility and additionally`** produces calculated data .

15) Print a log record with the pg_get_wal_record_info function , `which prints a single log` record rather than a range and therefore has one `input parameter` :

```
postgres=# select * from pg_get_wal_record_info(' 25/D8000070 ')\gx
-[ RECORD 1 ]----+-----------
start_lsn | 25/D8000070
end_lsn | 25/D8000108
prev_lsn | 25/D8000028
xid | 0
resource_manager | XLOG
record_type | CHECKPOINT_ONLINE
record_length | 148
main_data_length | 120
fpi_length | 0
description | redo 25/D8000028; tli 1; prev tli 1; fpw true; xid 361549; oid 942308;
multi 1; offset 0; oldest xid 723 in DB 1; oldest multi 1 in DB 5; oldest/newest commit
timestamp xid: 0/0; oldest running xid 361549; online
block_ref |
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

Function **pg_get_wal_record_info(in_lsn)** gives out those same the most data that And function **pg_get_wal_record s _info(start_lsn , end_lsn)** , but only By one magazine records .

`pg_walinspect` extension functions and the `pg_waldump` command line utility, depending on what you are more accustomed to using. To filter log entries with the extension, you can use the skills of filtering rows with SQL expressions. With `pg_waldump , you have to use` the grep , egrep, sed utilities to filter entries .

Part 9. Monitoring Blocks

Three sessions will be used. Open three terminals and run psql in them.

1) In the first session, execute the command to lock some table:

```
postgres=# begin;
lock table pgbench_accounts;
BEGIN
LOCK TABLE
postgres =*#
```

The lock command operates in a transaction, so a transaction was opened.

2) In the second session, execute the following queries:

```
postgres=# select pid, left(backend_type, 14) type, left(state,6) state,
wait_event_type wait_type, wait_event, age(backend_xid) age1, age(backend_xmin)
age2, round(extract('epoch' from clock_timestamp() - xact_start)) sec,
left(query,28) query from pg_stat_activity order by 1;
select pl.pid, pl.mode, pl.granted g, pl.relation::regclass, a.wait_event_type
wait, a.wait_event event, round(extract('epoch' from clock_timestamp() -
a.query_start)) sec, left(a.query,28) query from pg_locks pl left join
pg_stat_activity a ON pl.pid = a.pid where a.datname = current_database() order
by 1;
 pid | type | state | wait_type | wait_event | age1 | age2 | sec | query
--------+-------+--------+-----------+---------------------+------+------+-----+------------------------------
 699375 | checkpointer | | Activity | CheckpointerMain | | | |
 699376 | background wri | | Activity | BgWriterHibernate | | | |
 699378 | walwriter | | Activity | WalWriterMain | | | |
 699379 | autovacuum lau | | Activity | AutoVacuumMain | | | |
 699380 | logical replica | | Activity | LogicalLauncherMain | | | |
 762650 | client backend | idle i | Client | ClientRead | 1 | | 903 | lock table pgbench_accounts;
 762653 | client backend | active | | | | 1 | 0 | select pid, left(backend_typ
(7 rows)

 pid | mode | g | relation | wait | event | sec | query
--------+--------+---+------------+--------+------------+-----+------------------------------
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
762650 | ExclusiveLock | t | | Client | ClientRead | 903 | lock table pgbench_accounts;
762650 | ExclusiveLock | t | | Client | ClientRead | 903 | lock table pgbench_accounts;
762650 | AccessExclusiveLock | t | pgbench_accounts | Client | ClientRead | 903 | lock table pgbench_accounts;
762653 | AccessShareLock | t | pg_authid_oid_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database_oid_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid_rolname_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database_datname_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_stat_activity | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_locks | | | 0 | select pl.pid, pl.mode, pl.g
762653 | ExclusiveLock | t | | | | 0 | select pl.pid, pl.mode, pl.g
(12 rows )
```

The views have many columns and they don't fit on the page, so two queries are used. The second query fetches the values by joining the two views.

To monitor locks, use `pg_locks` . To find out in detail what command the process is **executing , use the pg_stat_activity view** .

The sec columns show the number of seconds from the moment the query was executed. If the queries are executed at different times, the values will be different.

3) Let's create a situation where the third session is waiting for the lock set by the first session to be released. In the third session, execute a query to the locked table:

```
postgres =# select count (*) from pgbench_accounts ;
```

The session is blocked and does not return a prompt.

4) In the second session, repeat the two requests:

```
postgres=# select pid, left(backend_type, 14) type, left(state,6) state,
wait_event_type wait_type, wait_event, age(backend_xid) age1, age(backend_xmin)
age2, round(extract('epoch' from clock_timestamp() - xact_start)) sec,
left(query,28) query from pg_stat_activity order by 1;
select pl.pid, pl.mode, pl.granted g, pl.relation::regclass, a.wait_event_type
wait, a.wait_event event, round(extract('epoch' from clock_timestamp() -
a.query_start)) sec, left(a.query,28) query from pg_locks pl left join
pg_stat_activity a ON pl.pid = a.pid where a.datname = current_database() order
by 1;
pid | type | state | wait_type | wait_event | age1 | age2 | sec | query
--------+-------+---------+----------+---------------------+------+------+------+-----------------------------
699375 | checkpointer | | Activity | CheckpointerMain | | | |
699376 | background wri | | Activity | BgWriterHibernate | | | |
699378 | walwriter | | Activity | WalWriterMain | | | |
699379 | autovacuum lau | | Activity | AutoVacuumMain | | | |
699380 | logical replica | | Activity | LogicalLauncherMain | | | |
762650 | client backend | idle i | Client | ClientRead | 1 | | 1451 | lock table pgbench_accounts;
762653 | client backend | active | | | 1 | 0 | select pid, left(backend_typ
763886 | client backend | active | Lock | relation | | 1 | 129 | select count(*) from pgbench
(8 rows)

pid | mode | g | relation | wait | event | sec | query
--------+--------+---+------------+--------+------------+------+-----------------------------
762650 | ExclusiveLock | t | | Client | ClientRead | 1451 | lock table pgbench_accounts;
762650 | AccessExclusiveLock | t | pgbench_accounts | Client | ClientRead | 1451 | lock table pgbench_accounts;
762650 | ExclusiveLock | t | | Client | ClientRead | 1451 | lock table pgbench_accounts;
762653 | ExclusiveLock | t | | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid_oid_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database_oid_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_stat_activity | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid_rolname_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database_datname_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_locks | | | 0 | select pl.pid, pl.mode, pl.g
 763886 | AccessShareLock | f | pgbench_accounts | Lock | relation | 129 | select count(*) from pgbench
763886 | ExclusiveLock | t | | Lock | relation | 129 | select count(*) from pgbench
(14 rows )
```

The results have been supplemented with rows. The fact that the session is locked in the second query is indicated by `pg_locks.granted = f alse` . In the first query, the lock is indicated by `wait_type = ' Lock '` . **The duration of the command execution** , including waits , is given in the **columns pg_stat_activity** `xact_start` and `query_start` . Column values:

xact_start - the start of an open transaction, or null if no transaction is open. This is the same as the start of the first command in the transaction. A non-empty value in xact_start means that the database horizon is being held.

query_start - the start of the currently running query or the duration of the last query (in this case the value of the state<>'active' column). While the query is running, the database horizon is held.

5) To monitor the database horizon, you can use the following queries:

```
postgres=# select datname database, trunc(extract(epoch from max(clock_timestamp()-xact_start))) sec, greatest(max(age(backend_xmin)), max(age(backend_xid))) age1 from pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname order by datname;
select pid, trunc(extract(epoch from (clock_timestamp()-xact_start))) sec, age(backend_xid) age1, age(backend_xmin) age2, datname database, usename, state, wait_event, left(query,20) query from pg_stat_activity where backend_xmin is not null or backend_xid is not null order by greatest(age(backend_xmin), age(backend_xid)) desc;
database | sec | age1
----------+------+------
postgres | 1458 | 1
(1 row)

pid | sec | age1 | age2 | database | usename | state | wait_event | query
--------+------+------+------+----------+----------+---------------------+------------+---------------------
762650 | 1458 | 1 | | postgres | postgres | idle in transaction | ClientRead | lock table pgbench_a
763886 | 871 | | 1 | postgres | postgres | active | relation | select count(*) from
765653 | 0 | | 1 | postgres | postgres | active | | select pid, trunc(ex
(3 rows)
```

The first query returns the horizon retention duration across databases.
The second query returns all processes and the duration of their transactions.
Long horizon holding times significantly reduce productivity.

6) All previous queries are not convenient for finding blocked or blocking processes.
More convenient queries for displaying blocked and blocking processes:

```
postgres=# select pl.pid blocked, pl.mode, a.wait_event_type wait, a.wait_event event, age(a.backend_xmin) age_xmin, round(extract('epoch' from clock_timestamp() - a.query_start)) waitsec, left(a.query,40) blocked_query from pg_locks pl left join pg_stat_activity a ON pl.pid = a.pid where pl.granted='f' and a.datname = current_database();
select a.pid blocked, bl.pid blocker, a.wait_event_type wait, a.wait_event event, age(a.backend_xmin) age, round(extract('epoch' from clock_timestamp() - a.query_start)) waitsec, bl.wait_event_type bl_type, bl.wait_event bl_wait, round(extract('epoch' from clock_timestamp() - bl.query_start)) bl_sec, left(a.query,10) blocked_q, left(bl.query,10) blocker_q from pg_stat_activity a join pg_stat_activity bl on bl.pid = ANY( pg_blocking_pids(a.pid) ) where a.wait_event_type='Lock' and cardinality( pg_blocking_pids(a.pid )>0;

blocked | mode | wait | event | age_xmin | waitsec | blocked_query
--------+---------------+------+----------+----------+---------+----------------------------------------
763886 |AccessShareLock | Lock | relation | 1 | 1436 | select count(*) from pgbench_accounts;
(1 row)

blocked | blocker | wait | event | age | waitsec | bl_type | bl_wait | bl_sec | blocked_q | blocker_q
--------+---------+------+----------+-----+---------+---------+------------+--------+------------+----------
763886 | 762650 | Lock | relation | 1 | 1436 | Client | ClientRead | 2758 | select cou | lock table
(1 row)
```

The first query shows the blocked process and the command it is executing.
The second query shows the ID of the blocking process. The ID is returned by the function , which is expensive to execute because it sets locks. Frequent calls to this function reduce performance. Using this function in periodically executed monitoring scripts is not recommended.

, it is better to use the first query: access to pg_locks . If the query returns processes that have granted =' f ' ., then you can use the pg_blocking_pids (pid) function to find blocking processes .

7) Perform command :

```
postgres=# select pg_cancel_backend( 762650 );
pg_cancel_backend
-------------------
t
(1 row)
```

After executing the function, nothing has changed. The blocking session is idle, and this function interrupts the execution of commands. If the command that set the lock was executed, the function would interrupt the execution of the command and, possibly, the lock would be released. If the lock is not released or the session is idle, then you can use the function that interrupts the session:

8) Complete command :

```
postgres=# select pg_terminate_backend( 762650 );
pg_cancel_backend
-------------------
t
(1 row)
```

In the second session, the command received a lock and executed successfully:

```
 count
--------
100000
(1 row)
```

In the first session, which is idle, there are no changes.

9) In the second session, repeat the requests:

In the second session, execute the following queries:

```
postgres=# select pid, left(backend_type, 14) type, left(state,6) state, wait_event_type wait_type, wait_event, age(backend_xid) age1, age(backend_xmin) age2, round(extract('epoch' from clock_timestamp() - xact_start)) sec, left(query,28) query from pg_stat_activity order by 1;
select pl.pid, pl.mode, pl.granted g, pl.relation::regclass, a.wait_event_type wait, a.wait_event event, round(extract('epoch' from clock_timestamp() - a.query_start)) sec, left(a.query,28) query from pg_locks pl left join pg_stat_activity a ON pl.pid = a.pid where a.datname = current_database() order by 1;
pid | type | state | wait_type | wait_event | age1 | age2 | sec | query
--------+-------+--------+-----------+--------------------+------+------+-----+------------------------------
699375 | checkpointer | | Activity | CheckpointerMain | | | |
699376 | background wri | | Activity | BgWriterHibernate | | | |
699378 | walwriter | | Activity | WalWriterMain | | | |
699379 | autovacuum lau | | Activity | AutoVacuumMain | | | |
699380 | logical replica | | Activity | LogicalLauncherMain | | | |
762653 | client backend | active | | | | 0 | 0 | select pid, left(backend_typ
763886 | client backend | idle | Client | ClientRead | | | | select count(*) from pgbench
(7 rows)

pid | mode | g | relation | wait | event | sec | query
--------+-----------------+---+-----------+------+-------+-----+--------------------------
762653 | AccessShareLock | t | pg_stat_activity | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_locks | | | 0 | select pl.pid, pl.mode, pl.g
762653 | ExclusiveLock | t | | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid_oid_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database_oid_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid_rolname_index | | | 0 | select pl.pid, pl.mode, pl.g
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
762653 | AccessShareLock | t | pg_database_datname_index | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_authid | | | 0 | select pl.pid, pl.mode, pl.g
762653 | AccessShareLock | t | pg_database | | | 0 | select pl.pid, pl.mode, pl.g
(9 rows )
```

Server process **762650** has terminated and is not listed in the operating system processes or in query results.

10) In the first session you can execute any query and make sure that the session is completed :

```
postgres=*# select 1;
FATAL: terminating connection due to administrator command
server closed the connection unexpectedly
This probably means the server terminated abnormally
before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded .
postgres=#
```

The session has connected to a new server process.

11) You can close unnecessary sessions and terminals.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Practice for Chapter 15

Part 1. Installing the pg_stat_kcache extension

The statistics collected by the instance in the standard pg_stat_io views do not allow to distinguish whether a block was in the Linux page cache or was read from disk. The pg_stat_kcache extension allows to distinguish this. The extension collects Linux statistics by executing the getrusage system call after each command.

Unlike operating system utilities, the extension collects statistics down to the SQL command.

If the extension is missing, you need to install it. The extension is written in C and includes a shared library. The extension does not require modifications to the PostgreSQL core.

The pg_stat_kcache extension is standardly supplied with **the** Tantor DBMS . SE starting from version 16.6.

1) Switch to the root operating system user terminal:

```
postgres @ tantor :~$ su -
Password : root
```

2) Install the language compiler c version 13:

```
root@tantor:~# apt install clang-13
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
clang-13 is already the newest version (1:13.0.1-11+b1).
The following packages were automatically installed and are no longer required:
libintl-perl libintl-xs-perl libmodule-find-perl libmodule-scandeps-perl libproc-processtable-perl libsort-naturally-perl libterm-readkey-perl
Use 'apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 607 not upgraded.
```

The compiler has been installed.

3) Download the extension source code:

```
root@tantor:~# wget https://github.com/powa-team/pg_stat_kcache/archive/REL2_3_0.tar.gz
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/powa-team/pg_stat_kcache/tar.gz/refs/tags/REL2_3_0 [following]
Length: unspecified [application/x-gzip]
Saving to: 'REL2_3_0.tar.gz.1'
'REL2_3_0.tar.gz.1' saved [24477]
```

4) Unzip file REL2_3_0.tar.gz:

```
root@tantor:~# tar xzf ./REL2_3_0.tar.gz
```

5) Go to the directory created when unpacking the archive:

```
root@tantor:~# cd pg_stat_kcache-REL2_3_0
```

6) Build the extension:

```
root @ tantor :~/ pg _ stat _ kcache - REL 2_3_0# make
gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Werror=vla -Wendif-labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -Wcast-function-type -Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv -fexcess-precision=standard -Wno-format-truncation -Wno-stringop-truncation -O2 -pipe -Wno-missing-braces -DTANTOR_SE -fPIC -fvisibility=hidden -I. -I./ -I/opt/tantor/db/16/include/postgresql/server -I/opt/tantor/db/16/include/postgresql/internal -O2 -pipe -Wno-missing-braces -DTANTOR_SE -
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
D_GNU_SOURCE -I/usr/include/libxml2 -I/usr/local/include/zstd -c -o pg_stat_kcache.o
pg_stat_kcache.c
   gcc -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-statement -Werror=vla -Wendif-
labels -Wmissing-format-attribute -Wimplicit-fallthrough=3 -Wcast-function-type -
Wshadow=compatible-local -Wformat-security -fno-strict-aliasing -fwrapv -fexcess-precision=standard
-Wno-format-truncation -Wno-stringop-truncation -O2 -pipe -Wno-missing-braces -DTANTOR_SE -fPIC -
fvisibility=hidden -shared -o pg_stat_kcache.so pg_stat_kcache.o -L/opt/tantor/db/16/lib -
L/usr/lib/llvm-13/lib -L/usr/local/lib/zstd -Wl,--as-needed -Wl,-rpath,'/opt/tantor/db/16/lib',--
enable-new-dtags -fvisibility=hidden
   /usr/bin/clang-13 -Wno-ignored-attributes -fno-strict-aliasing -fwrapv -Wno-unused-command-line-
argument -Wno-compound-token-split-by-macro -O2 -I. -I./ -
I/opt/tantor/db/16/include/postgresql/server -I/opt/tantor/db/16/include/postgresql/internal -O2 -
pipe -Wno-missing-braces -DTANTOR_SE -D_GNU_SOURCE -I/usr/include/libxml2 -I/usr/local/include/zstd
-flto=thin -emit-llvm -c -o pg_stat_kcache.bc pg_stat_kcache.c
```

7) On the Tantor DBMS version SE 16.6 and newer, you do not need to perform this step, since the extension library is already installed.

Install files extensions :

```
root@tantor:~/pg_stat_kcache-REL2_3_0# make install
/usr/bin/mkdir -p '/opt/tantor/db/16/lib/postgresql'
/usr/bin/mkdir -p '/opt/tantor/db/16/share/postgresql/extension'
/usr/bin/mkdir -p '/opt/tantor/db/16/share/postgresql/extension'
/usr/bin/install -c -m 755 pg_stat_kcache.so '/opt/tantor/db/16/lib/postgresql/pg_stat_kcache.so'
/usr/bin/install -c -m 644 .//pg_stat_kcache.control
'/opt/tantor/db/16/share/postgresql/extension/'
/usr/bin/install -c -m 644 .//pg_stat_kcache--2.1.0--2.1.1.sql .//pg_stat_kcache--2.1.0.sql
.//pg_stat_kcache--2.1.1--2.1.2.sql .//pg_stat_kcache--2.1.1.sql .//pg_stat_kcache--2.1.2--
2.1.3.sql .//pg_stat_kcache--2.1.2.sql .//pg_stat_kcache--2.1.3--2.2.0.sql .//pg_stat_kcache--
2.1.3.sql .//pg_stat_kcache--2.2.0--2.2.1.sql .//pg_stat_kcache--2.2.0.sql .//pg_stat_kcache--
2.2.1--2.2.2.sql .//pg_stat_kcache--2.2.1.sql .//pg_stat_kcache--2.2.2--2.2.3.sql
.//pg_stat_kcache--2.2.2.sql .//pg_stat_kcache--2.2.3--2.3.0.sql .//pg_stat_kcache--2.2.3.sql
.//pg_stat_kcache--2.3.0.sql '/opt/tantor/db/16/share/postgresql/extension/'
/usr/bin/mkdir -p '/opt/tantor/db/16/lib/postgresql/bitcode/pg_stat_kcache'
/usr/bin/mkdir -p '/opt/tantor/db/16/lib/postgresql/bitcode'/pg_stat_kcache/
/usr/bin/install -c -m 644 pg_stat_kcache.bc
'/opt/tantor/db/16/lib/postgresql/bitcode'/pg_stat_kcache/./
cd '/opt/tantor/db/16/lib/postgresql/bitcode' && /usr/lib/llvm-13/bin/llvm-lto -thinlto -thinlto-
action=thinlink -o pg_stat_kcache.index.bc pg_stat_kcache/pg_stat_kcache.bc
```

8) Go back or open the terminal of the postgres operating system user :

```
root@tantor:~/pg_stat_kcache-REL2_3_0# <ctrl+d>
logout
```

9) Connect the extension libraries that will be used further in the practices:

```
postgres@tantor:~$ psql -c "alter system set shared_preload_libraries =
pg_stat_statements, pg_wait_sampling, pg_stat_kcache;"
ALTER SYSTEM
```

10) Restart instance :

```
postgres@tantor:~$ sudo restart
```

11) Run psql and install the extension:

```
postgres@tantor:~$ psql

postgres=# create extension pg_stat_kcache;
ERROR: required extension "pg_stat_statements" is not installed
HINT: Use CREATE EXTENSION ... CASCADE to install required extensions too.
postgres=# create extension pg_stat_kcache cascade ;
NOTICE: installing required extension "pg_stat_statements"
CREATE EXTENSION
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

The kcache extension uses pg_stat_statements extension . It could be installed with a separate command, but the cascade option was used for installation.

12) See what objects are included in the extensions:

```
postgres=# \dx+ pg_stat_statements
Objects in extension "pg_stat_statements"
Object description
-----------------------------------------------------------
function pg_stat_statements(boolean)
function pg_stat_statements_info()
function pg_stat_statements_reset(oid,oid,bigint,boolean)
view pg_stat_statements
view pg_stat_statements_info
(5 rows)

postgres=# \dx+ pg_stat_kcache
Objects in extension "pg_stat_kcache"
Object description
---------------------------------
function pg_stat_kcache()
function pg_stat_kcache_reset()
view pg_stat_kcache
view pg_stat_kcache_detail
(4 rows)
```

Extensions contain views and functions.

`pg_stat_kcache_detail` view has columns: `query, top, rolname` and gives data with command precision. Statistics are given from 14 columns for planning and 14 columns for command execution.

The pg_stat_kcache view contains summary statistics from pg_stat_kcache_detail, grouped by database:

```
CREATE VIEW pg_stat_kcache AS SELECT datname, SUM( columns ) FROM
pg_stat_kcache_detail WHERE top IS TRUE GROUP BY datname;
```

Statistics in both views:

```
exec_reads reads, in bytes
exec_writes writes, in bytes
exec_reads_blks reads, in 8K-blocks
exec_writes_blks writes, in 8K-blocks
exec_user_time user CPU time used
exec_system_time system CPU time used
exec_minflts page reclaims (soft page faults)
exec_majflts page faults (hard page faults)
exec_nswaps swaps
exec_msgsnds IPC messages sent
exec_msgrcvs IPC messages received
exec_nsignals signals received
exec_nvcsws voluntary context switches
exec_nivcsws involuntary context switches
```

The data corresponds to the rusage structure, its contents can be viewed with the `man getrusage command` .

If `exec_majflts` is insignificant compared to `exec_minflts , it means that there` is enough RAM. Other than this statement, `the exec_majflts statistic is meaningless .`

`The` ratio of `exec_user_time and` exec_system_time `allows you` to determine whether there is a bias towards the kernel code or the PostgreSQL `code and extension and application` functions.

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

The presence of `exec_nivcsws ( forced execution` context switches by the operating system scheduler) indicates that the request was heavily CPU-intensive and the CPU was the bottleneck.

13) Look at the parameters of the extension:

```
postgres=# \dconfig pg_stat_k*
List of configuration parameters
Parameter | Value
--------------------------------+--------
pg_stat_kcache.linux_hz | 250000
pg_stat_kcache.track | top
pg_stat_kcache.track_planning | off
(3 rows)
```

`pg_stat_kcache.linux_hz` (default -1) is automatically set to the value of the linux CONFIG_HZ parameter and is used to compensate for sampling errors. No need to change.

`pg_stat_kcache.track (` default `top )` three values :

a) `top` - track commands sent for execution by the client

b) `all` - track nested queries. For example, queries in called functions

c) `none` - disables collection of pg_stat_kcache statistics

`pg_stat_kcache . track_planning` (default `off`) whether to collect statistics for the planning stage

14) Extensions can allocate structures in shared memory to accumulate data. See what structures memory (buffers) were highlighted extensions :

```
postgres=# select * from (select *,lead(off) over(order by off)-off as true
from pg_shmem_allocations) as a where name like 'pg_%' order by 1;
name | off | size | allocated_size | true
---------------------------+----------+-------+----------------+---------
pg_qualqueryexamples_hash | 152021248 | 2896 | 2944 | 1067136
pg_qualstatements_hash | 151746048 | 2896 | 2944 | 275200
pg_qualstats | 151741696 | 147 | 256 | 4352
 pg_stat_kcache | 150352128 | 992 | 1024 | 1024
pg_stat_kcache hash | 150353152 | 2896 | 2944 | 1388544
pg_stat_statements | 148163456 | 64 | 128 | 128
pg_stat_statements hash | 148163584 | 2896 | 2944 | 2188544
pg_store_plans | 153088384 | 72 | 128 | 128
pg_store_plans hash | 153088512 | 2896 | 2944 | 251136
pg_wait_sampling | 148145920 | 17536 | 17536 | 17536
(10 rows)
```

`The pg_stat_kcache` extension uses two shared `memory buffers .`

`of` statements for which statistics are collected and the size of shared memory structures are determined by the pg_stat_statements.max parameter `( default 5000`), since the `pg_stat_kcache extension` depends on the `pg_stat_statements extension .`

15) Close psql and run the commands:

```
postgres=# \q
postgres@tantor:~$
psql -c "alter system set pg_stat_statements.max= 50000 ;"
sudo restart
psql -c "select * from (select *,lead(off) over(order by off)-off as true from
pg_shmem_allocations) as a where name like 'pg_%' order by 1;"
psql -c "alter system reset pg_stat_statements.max;"
sudo restart
psql
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
ALTER SYSTEM
name | off | size | allocated_size | true
---------------------------+----------+-------+----------------+----------
pg_qualqueryexamples_hash | 183898624 | 2896 | 2944 | 1067136
pg_qualstatements_hash | 183623424 | 2896 | 2944 | 275200
pg_qualstats | 183619072 | 147 | 256 | 4352
 pg_stat_kcache | 169890816 | 992 | 1024 | 1024
pg_stat_kcache hash | 169891840 | 2896 | 2944 | 13727232
pg_stat_statements | 148163456 | 64 | 128 | 128
pg_stat_statements hash | 148163584 | 2896 | 2944 | 21727232
pg_store_plans | 184965760 | 72 | 128 | 128
pg_store_plans hash | 184965888 | 2896 | 2944 | 251136
pg_wait_sampling | 148145920 | 17536 | 17536 | 17536
(10 rows)

ALTER SYSTEM

postgres=#
```

At increase numbers teams , by which is going to statistics 10 times from 5000 to 50000 parameter **pg_stat_statements.max** , size buffers `pg_stat_kcache hash` And `pg_stat_statements hash` increased by ~10 times .

Part 2. Using the pg_stat_kcache extension

1) Example of a request for statistics collected by the extension:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0)
time, s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid order by
total_exec_time desc limit 5;

database | time | calls | recl |faults| reads | writes | user | sys | vsw | isw | query
----------+------+-------+-------+------+---------+---------+------+------+------+-----+-----------------
postgres | 1020 | 19988 | 4323 | 0 | 15 MB | 0 bytes | 0.48 | 0.62 | 1281 | 25 | UPDATE pgbench_
postgres | 488 | 19988 | 7 | 0 | 0 bytes | 0 bytes | 0.38 | 0.47 | 0 | 21 | UPDATE pgbench_
postgres | 459 | 19988 | 9 | 0 | 0 bytes | 0 bytes | 0.37 | 0.47 | 0 | 19 | UPDATE pgbench_
postgres | 369 | 19988 | 0 | 0 | 0 bytes | 0 bytes | 0.34 | 0.41 | 0 | 17 | SELECT abalance
postgres | 306 | 19988 | 272 | 0 | 0 bytes | 1056 kB | 0.29 | 0.38 | 1 | 13 | INSERT INTO pgb
(5 rows)
```

2) Reset statistics :

```
postgres=#
select pg_stat_statements_reset();
select pg_stat_kcache_reset();
pg_stat_statements_reset
-------------------------------
2035-01-01 01:01:01.101508+03
(1 row)

pg_stat_kcache_reset
----------------------

(1 row)
```

Since the request combined data from two extensions, it is necessary to reset the statistics of both extensions.

3) Repeat request :

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0)
time, s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid order by
total_exec_time desc limit 5;
database | time | calls | faults | recl | reads | writes | user | sys | vsw | isw | query
----------+------+-------+--------+------+---------+---------+------+------+-----+-----+-----------------
postgres | 0 | 1 | 1 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_
(1 row )
```

Statistics have been reset.

4) Run the initins .sql test , which involves creating 10 tables of 422MB each:

```
psql -c "select pg_stat_reset_shared('io');"
psql -c " select pg_stat_reset_shared('bgwriter');"
psql -f initcre.sql 2> /dev/null > /dev/null
time psql -f initins.sql > /dev/null
pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

real 4m0 .245s
user 0m0.010s
sys 0 m 0.000 s
```

Adding libraries with the shared_preload_libraries parameter **did not affect the script** execution time: in point 7 of part 6 of practice 14 it was 3 m 58 . A slight increase in execution time due to the `track_io_timing=on parameter` .

5) While the commands are running in the second window, run the commands:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0) time,
s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid order by
total_exec_time desc;
\watch 3

database | time | calls | recl |faults| reads | writes | user | sys | vsw | isw | query
----------+--------+-------+--------+------+---------+---------+-------+------+-----+-----+-----
postgres | 24959 | 1 | 938 | 0 | 0 bytes | 1402 MB | 11.50 | 3.66 | 276 | 265 | insert into test.f
postgres | 24891 | 1 | 936 | 0 | 0 bytes | 1397 MB | 11.96 | 3.28 | 244 | 254 | insert into test.f
postgres | 24271 | 1 | 945 | 0 | 0 bytes | 1295 MB | 12.04 | 3.25 | 234 | 277 | insert into test.f
postgres | 24265 | 1 | 956 | 0 | 0 bytes | 1260 MB | 12.01 | 3.24 | 257 | 239 | insert into test.f
postgres | 23717 | 1 | 934 | 0 | 0 bytes | 1305 MB | 11.94 | 3.35 | 231 | 241 | insert into test.f
postgres | 22869 | 1 | 1205 | 0 | 0 bytes | 1317 MB | 12.46 | 2.70 | 337 | 269 | insert into test.f
postgres | 19388 | 1 | 37420 | 79 | 3544 kB | 1253 MB | 12.34 | 2.82 | 276 | 191 | insert into test.f
postgres | 5 | 16 | 222 | 0 | 0 bytes | 32 kB | 0.01 | 0.00 | 0 | 0 | select d.datname d
postgres | 0 | 1 | 75 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select d.datname d
postgres | 0 | 2 | 12 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_res
postgres | 0 | 2 | 8 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select format($1,
postgres | 0 | 1 | 1 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_kca
(12 rows )
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Command lines `insert` will appear after these commands are completed.

6) After the script has finished executing, view the general statistics using the query:

```
postgres=# select datname database, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(exec_reads) reads,
pg_size_pretty(exec_writes) writes, round(exec_system_time::numeric,0) sys,
round(exec_user_time::numeric,0) usr, exec_nvcsws vsw, exec_nivcsws isw from
pg_stat_kcache;
database | recl |faults| reads | writes | sys | usr | vsw | isw
----------+--------+------+---------+--------+-----+-----+------+------
postgres | 46586 | 79 | 3544 kB | 13 GB | 33 | 119 | 2624 | 2737
(1 row)
```

7) Perform command :

```
postgres@tantor:~$ time psql -f initsel.sql > /dev/null

real 1m13 .586s
user 0m0.012s
sys 0m0.000s
```

The test took a long time to run because the first select on the tables made the blocks dirty.

8) in the second stop `\watch 1` with the key combination < `ctrl + c` > and run the commands:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0) time,
s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time:: numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws
isw, left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k
using (userid , dbid, queryid) join pg_database d on s.dbid = d.oid where query
like 'select%' order by 12 desc limit 11;
\watch 3

database | time | calls | recl |faults| reads | writes | user | sys | vsw | isw | query
----------+-------+-------+---------+------+---------+--------+------+------+-------+-----+-----
postgres | 10366 | 1 | 7935 | 0 | 422 MB | 731 MB | 2.70 | 2.05 | 11494 | 688 | select * from test
postgres | 10122 | 1 | 7428 | 0 | 422 MB | 743 MB | 2.83 | 1.90 | 11251 | 412 | select * from test
postgres | 9087 | 1 | 7218 | 0 | 424 MB | 744 MB | 2.84 | 1.88 | 11312 | 227 | select * from test
postgres | 7779 | 1 | 7645 | 0 | 422 MB | 778 MB | 2.97 | 2.05 | 11814 | 539 | select * from test
postgres | 7758 | 1 | 7383 | 0 | 422 MB | 752 MB | 2.93 | 1.73 | 12135 | 157 | select * from test
postgres | 7121 | 1 | 7387 | 0 | 425 MB | 772 MB | 2.93 | 1.99 | 12238 | 185 | select * from test
postgres | 7112 | 1 | 7000 | 0 | 422 MB | 770 MB | 2.84 | 2.02 | 11049 | 465 | select * from test
postgres | 6954 | 1 | 8293 | 0 | 422 MB | 749 MB | 2.96 | 1.96 | 10747 | 352 | select * from test
postgres | 5870 | 1 | 7640 | 0 | 426 MB | 761 MB | 2.96 | 1.98 | 11434 | 404 | select * from test
postgres | 4765 | 1 | 38121 | 1 | 422 MB | 675 MB | 2.75 | 1.95 | 12040 | 131 | select * from test
postgres | 7 | 20 | 260 | 0 | 0 bytes | 32 kB | 0.01 | 0.00 | 0 | 0 | select d.dataname d
(11 rows )
```

9) Repeat the commands several times:

```
postgres@tantor:~$
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null

real 0m17 .399s
user 0m0.010s
sys 0m0.000s

real 0m18 .726s
user 0m0.005s
```

```
sys 0m0.005s

real 0m13 .751s
user 0m0.013s
sys 0m0.000s

real 0m15 .422s
user 0m0.005s
sys 0m0.005s

real 0m15 .717s
user 0m0.012s
sys 0 m 0.000 s
```

The select commands in the test did not dirty the blocks.
The first commands were executed for ~18 seconds - the Linux cache was filled, the next ~15 seconds - the Linux cache was filled.

See what values increase when running the **initsel .sql test** :

```
 database | time | calls | recl | faults | reads | writes | user | sys | vsw | isw | query
----------+-------+-------+--------+------+---------+---------+------+------+-------+------+-----
postgres | 14786 | 6 | 14402 | 0 | 1957 MB | 749 MB | 8.69 | 8.99 | 36143 | 1209 | select * from test
postgres | 18002 | 6 | 13533 | 0 | 2075 MB | 743 MB | 8.49 | 8.81 | 38478 | 1176 | select * from test
postgres | 18184 | 6 | 13969 | 0 | 2062 MB | 731 MB | 8.38 | 8.95 | 38795 | 1139 | select * from test
postgres | 15365 | 6 | 13685 | 0 | 1931 MB | 778 MB | 8.38 | 8.97 | 36917 | 1120 | select * from test
postgres | 13465 | 6 | 13667 | 0 | 1935 MB | 761 MB | 8.33 | 9.11 | 36669 | 1014 | select * from test
postgres | 14994 | 6 | 13085 | 0 | 2078 MB | 770 MB | 8.31 | 9.03 | 38189 | 881 | select * from test
postgres | 16188 | 6 | 13316 | 0 | 1744 MB | 744 MB | 8.38 | 8.74 | 33297 | 857 | select * from test
postgres | 14761 | 6 | 13462 | 0 | 1962 MB | 772 MB | 8.67 | 8.71 | 37739 | 732 | select * from test
postgres | 14318 | 6 | 34851 | 0 | 1672 MB | 752 MB | 8.99 | 6.99 | 32540 | 719 | select * from test
postgres | 11515 | 6 | 59070 | 1 | 1592 MB | 675 MB | 8.42 | 8.56 | 31446 | 657 | select * from test
postgres | 0 | 8 | 49 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select format($1,
postgres | 7 | 20 | 260 | 0 | 0 bytes | 32 kB | 0.01 | 0.00 | 0 | 0 | select d.dataname d
(12 rows )
```

The values in all numeric columns are increased except `faults` and `writes` .
in the `calls column` .

10) Reset the statistics and re-run one test:

```
postgres=#
psql -c "select pg_stat_statements_reset();"
psql -c "select pg_stat_kcache_reset();"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
time psql -f initsel.sql > /dev/null
...
real 0m18 .596s
user 0m0.010s
sys 0 m 0.002 s
```

11) Look at the statistics for one-time requests:

```
database | time | calls | recl | faults | reads | writes | user | sys | vsw | isw | query
----------+------+-------+--------+------+---------+---------+------+------+------+-----+--------------------
postgres | 1732 | 1 | 1453 | 0 | 366 MB | 0 bytes | 1.15 | 1.41 | 5966 | 291 | select * from test
postgres | 1690 | 1 | 1473 | 0 | 376 MB | 0 bytes | 1.18 | 1.37 | 6138 | 125 | select * from test
postgres | 792 | 1 | 2473 | 0 | 21 MB | 0 bytes | 1.10 | 1.07 | 372 | 122 | select * from test
postgres | 1754 | 1 | 1426 | 0 | 391 MB | 0 bytes | 1.13 | 1.50 | 6404 | 91 | select * from test
postgres | 1704 | 1 | 4431 | 0 | 375 MB | 0 bytes | 1.25 | 1.36 | 6111 | 84 | select * from test
postgres | 1175 | 1 | 1432 | 0 | 171 MB | 0 bytes | 1.11 | 1.21 | 2892 | 62 | select * from test
postgres | 1761 | 1 | 1442 | 0 | 410 MB | 0 bytes | 1.03 | 1.46 | 6782 | 62 | select * from test
postgres | 1744 | 1 | 1449 | 0 | 377 MB | 0 bytes | 1.16 | 1.48 | 6181 | 51 | select * from test
postgres | 1718 | 1 | 1429 | 0 | 395 MB | 0 bytes | 1.15 | 1.41 | 6430 | 50 | select * from test
postgres | 1012 | 1 | 1451 | 0 | 115 MB | 0 bytes | 1.14 | 1.10 | 2050 | 35 | select * from test
postgres | 1 | 4 | 0 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 1 | select d.dataname d
postgres | 0 | 1 | 13 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_kca
(12 rows)
```

All reads were performed from the linux cache.

When reading one table from the Linux cache, ~1400 were read blocks. The two tables have several times more .

The size of each table is 422 MB:

```
postgres =# \ dt + test .*
                                   List of relations
Schema | Name | Type | Owner | Persistence | Access method | Size | Description
--------+-------+-------+----------+-------------+---------------+--------+-------------
test | film_summary1 | table | postgres | permanent | heap | 422 MB |
test | film_summary10 | table | postgres | permanent | heap | 422 MB |
test | film_summary2 | table | postgres | permanent | heap | 422 MB |
test | film_summary3 | table | postgres | permanent | heap | 422 MB |
test | film_summary4 | table | postgres | permanent | heap | 422 MB |
test | film_summary5 | table | postgres | permanent | heap | 422 MB |
test | film_summary6 | table | postgres | permanent | heap | 422 MB |
test | film_summary7 | table | postgres | permanent | heap | 422 MB |
test | film_summary8 | table | postgres | permanent | heap | 422 MB |
test | film_summary9 | table | postgres | permanent | heap | 422 MB |
(10 rows)
```

12) Buffer cache size 128MB:

```
postgres=# show shared_buffers;
shared_buffers
----------------
128MB
(1 row)
```

13) When executing select buffer rings were used and the buffers in the rings were reused for table blocks:

```
postgres=# select backend_type, context, writes w, round(write_time::numeric,2)
wt, writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev, reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
   backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
----------+----------+---+------+----+------+----+----+------+-----+--------+----+-----
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 2453 | 0 | | 0 | 0
client backend | bulkread | 0 | 0.00 | 0 | 0.00 | | | 4817 | 320 | 179947 | |
client backend | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 463 | 0 | | 0 | 0
background worker | bulkread | 0 | 0.00 | 0 | 0.00 | | | 9680 | 640 | 345146 | |
background worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 3240 | 0 | | 0 | 0
(5 rows)
```

14) Check how will affect on indicators increase size buffer cache to 2 Gb . Then the buffer pool will be increased and the test will be repeated. Run the commands:

```
postgres@tantor:~$
psql -c "alter system set shared_buffers=' 2GB '"
psql -c "select pg_stat_statements_reset();"
psql -c "select pg_stat_kcache_reset();"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
sudo restart
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
...
real 0m18 .908s
user 0m0.010s
sys 0m0.000s

real 0m19 .400s
user 0m0.009s
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
sys 0m0.001s

real 0m19 .122s
user 0m0.009s
sys 0m0.004s

real 0m19 .556s
user 0m0.002s
sys 0m0.012s

real 0m19 .197s
user 0m0.014s
sys 0 m 0.000 s
```

15) Statistics after executing commands:

```
postgres=# select d.datname database, round(s.total_exec_time::numeric, 0) time,
s.calls, pg_size_pretty(exec_minflts*4096) recl ,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid where query like
'select%' order by 12 desc limit 11;

select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev , reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
database | time | calls | recl |faults| reads | writes | user | sys | vsw | isw | query
----------+------+-------+--------+------+---------+---------+------+------+-------+-----+-----
postgres | 0 | 2 | 10 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_res
postgres | 0 | 1 | 13 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_kca
postgres | 9140 | 5 | 486570 | 0 | 2097 MB | 0 bytes | 6.27 | 7.53 | 34311 | 249 | select * from test
postgres | 8991 | 5 | 550942 | 0 | 1949 MB | 0 bytes | 6.38 | 7.49 | 32074 | 325 | select * from test
postgres | 9219 | 5 | 541808 | 0 | 2095 MB | 0 bytes | 6.48 | 7.38 | 34087 | 238 | select * from test
postgres | 9206 | 5 | 551916 | 0 | 2022 MB | 0 bytes | 6.45 | 7.21 | 33507 | 206 | select * from test
postgres | 9175 | 5 | 474780 | 0 | 2116 MB | 0 bytes | 6.20 | 7.38 | 35075 | 266 | select * from test
postgres | 9178 | 5 | 487968 | 0 | 2095 MB | 0 bytes | 6.35 | 7.34 | 34234 | 211 | select * from test
postgres | 9241 | 5 | 485323 | 0 | 2111 MB | 0 bytes | 6.18 | 7.48 | 34724 | 251 | select * from test
postgres | 8598 | 5 | 550936 | 0 | 1828 MB | 0 bytes | 6.41 | 7.26 | 30014 | 320 | select * from test
postgres | 8395 | 5 | 566218 | 0 | 1708 MB | 0 bytes | 6.36 | 7.33 | 28196 | 499 | select * from test
(11 rows)

backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
---------------------+---------+---+------+----+------+----+----+-------+---------+----+----+-----
autovacuum launcher | normal | 2 | 0.07 | 0 | 0.00 | | | 61 | 5 | | 0 | 0
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 5775 | 1584 | | 0 | 0
client backend | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 3913 | 832895 | | 0 | 0
background worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 16191 | 1608110 | | 0 | 0
(4 rows )
```

Buffer rings were not used. The total size of the tables is 4.4 GB, which exceeds the size of the buffer cache. The displacement of occupied blocks and their replacement with other blocks (**evictions**) increased. Due to the increase in instance memory, there was little memory left for the Linux page cache and its efficiency decreased. The test execution time increased to ~19 seconds . If you reset the Linux cache, the result will be the same:

```
root@tantor:~# echo 1 > /proc/sys/vm/drop_caches
real 0m19 .380s
real 0m18 .777s
```

The number of five readings of each table ~ 550942 has increased disproportionately compared to the values from item 9 14402 .

When using a 128MB buffer cache, the test execution time was ~15 seconds .

exec_minflts statistics does not show reading pages from disk that are not in the page cache.

Part 3. Performance when using direct i/o

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

1) Enable direct I/O (direct i /o) , perform the test and see the test result:

```
postgres@tantor:~$
psql -c "alter system set debug_io_direct = data ;"
psql -c "alter system reset shared_buffers;"
psql -c "select pg_stat_statements_reset();"
psql -c "select pg_stat_kcache_reset();"
psql -c "select pg_stat_reset_shared('io');"
psql -c "select pg_stat_reset_shared('bgwriter');"
sudo restart
time psql -f initsel.sql > /dev/null
time psql -f initsel.sql > /dev/null
psql -c "select d.datname database, round(s.total_exec_time::numeric, 0) time,
s.calls, pg_size_pretty(exec_minflts*4096) recl,
pg_size_pretty(exec_majflts*4096) faults, pg_size_pretty(k.exec_reads) reads,
pg_size_pretty(k.exec_writes) writes, round(k.exec_user_time::numeric, 2) user,
round(k.exec_system_time::numeric, 2) sys, k.exec_nvcsws vsw, k.exec_nivcsws isw,
left(s.query, 18) query from pg_stat_statements s join pg_stat_kcache() k using
(userid, dbid, queryid) join pg_database d on s.dbid = d.oid where query like
'select%' order by 12 desc limit 11;"
psql -c "select backend_type, context, writes w, round(write_time::numeric,2) wt,
writebacks wb, round(writeback_time::numeric,2) wbt, extends ex,
round(extend_time::numeric) et, hits, evictions ev , reuses ru, fsyncs fs,
round(fsync_time::numeric) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;"
psql -c "alter system reset debug_io_direct;"
sudo restart
```

```
real 0m56 .905s
user 0m0.010s
sys 0m0.000s

real 0m56 .558s
user 0m0.009s
sys 0m0.000s

database | time | calls | recl |faults| reads | writes | user | sys | vsw | isw | query
----------+-------+-------+--------+------+---------+---------+------+------+--------+-----+-----
postgres | 0 | 2 | 10 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_res
postgres | 0 | 1 | 13 | 0 | 0 bytes | 0 bytes | 0.00 | 0.00 | 0 | 0 | select pg_stat_kca
postgres | 11673 | 2 | 2316 | 0 | 844 MB | 0 bytes | 2.47 | 5.66 | 108018 | 21 | select * from test
postgres | 11648 | 2 | 2301 | 0 | 844 MB | 0 bytes | 2.14 | 5.49 | 108023 | 32 | select * from test
postgres | 11819 | 2 | 2343 | 0 | 844 MB | 0 bytes | 2.02 | 5.77 | 108014 | 30 | select * from test
postgres | 11401 | 2 | 2345 | 0 | 844 MB | 0 bytes | 2.05 | 5.74 | 108019 | 24 | select * from test
postgres | 11391 | 2 | 2311 | 0 | 844 MB | 0 bytes | 1.98 | 5.79 | 108015 | 31 | select * from test
postgres | 11754 | 2 | 2313 | 0 | 844 MB | 0 bytes | 2.13 | 5.84 | 108014 | 42 | select * from test
postgres | 11825 | 2 | 8232 | 0 | 844 MB | 0 bytes | 2.49 | 5.60 | 108018 | 86 | select * from test
postgres | 11425 | 2 | 2344 | 0 | 844 MB | 0 bytes | 2.70 | 5.52 | 108019 | 84 | select * from test
postgres | 11444 | 2 | 2340 | 0 | 844 MB | 0 bytes | 2.42 | 5.71 | 108018 | 32 | select * from test
(11 rows)

backend_type | context | w | wt | wb | wbt | ex | et | hits | ev | ru | fs | fst
---------------------+----------+---+------+----+------+----+----+------+----+--------+----+-----
autovacuum launcher | normal | 0 | 0.00 | 0 | 0.00 | | | 10 | 0 | | 0 | 0
autovacuum worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 1663 | 0 | | 0 | 0
client backend | bulkread | 0 | 0.00 | 0 | 0.00 | | | 385 | 0 | 361427 | |
client backend | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 3091 | 2 | | 0 | 0
background worker | bulkread | 0 | 0.00 | 0 | 0.00 | | | 575 | 0 | 716793 | |
background worker | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 6471 | 0 | | 0 | 0
(6 rows )
```

Command execution time at startup **direct i / o** significantly increased - up to `0 m 56` . Test result: do not turn on **direct i / o** using the experimental `debug_io_direct option` , `as using it` will significantly reduce performance.

`Faults` statistics are zero. `Reclaims` statistics have become small.

`writes` statistics are zero because the select commands did not dirty the blocks and no blocks were written.

Practice for Chapter 16

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

Part 1. Using pg_wait_sampling extension

1) Check if the following libraries are present in the directory so that the libraries can be loaded:

```
postgres@tantor:~$ ls ` pg_config --pkglibdir ` | egrep
"statements|sampling|qualstats|plans|prewarm|kcache"
pg_prewarm.so
pg_qualstats.so
pg_stat_kcache.so
pg_stat_statements.so
pg_store_plans.so
pg_wait_sampling.so
```

The command uses backtick characters rather than apostrophes.

2) Remove libraries that are not in the directory from the following command and load those that are:

```
postgres@tantor:~$ psql -c "alter system set shared_preload_libraries =
pg_stat_statements, pg_wait_sampling, pg_qualstats, pg_store_plans, pg_prewarm,
pg_stat_kcache ;"
ALTER SYSTEM
```

If any of the library files are missing, the instance will not start. The alter system command will fail because it is executed on a running instance. To change the value of the shared_preload_libraries parameter, `you will have to edit` a text file in the file system. You should not stop the instance without access to the file system, because it may not start.

The command specifies 6 libraries.

in which libraries are listed in the shared_preload_libraries parameter can be important. For example, the pg_wait_sampling library should **be listed after** pg_stat_statements to **prevent** pg_wait_sampling **from** overwriting `queryid` s `used` by `pg_wait_sampling .`

Libraries may depend on each other. For example, the pg_stat_kcache library **requires the pg_stat_statements** library **, otherwise the instance will not start :**

```
postgres@tantor:~$ psql -c "alter system set shared_preload_libraries =
pg_wait_sampling, pg_qualstats, pg_store_plans, pg_prewarm, pg_stat_kcache;"
ALTER SYSTEM
postgres@tantor:~$ pg_ctl stop
waiting for server to shut down.... done
server stopped
postgres@tantor:~$ pg_ctl start
waiting for server to start....
[552186] LOG: Auto detecting pg_stat_kcache.linux_hz parameter...
[552186] LOG: pg_stat_kcache.linux_hz is set to 62500
[552186] FATAL: unrecognized configuration parameter "pg_stat_statements.max"
[552186] HINT: make sure pg_stat_statements is loaded and make sure pg_stat_kcache is
present after pg_stat_statements in the shared_preload_libraries setting
[552186] LOG: database system is shut down
stopped waiting
pg_ctl: could not start server
```

3) Restart the instance so that the libraries are loaded:

```
postgres@tantor:~$ sudo systemctl restart tantor-se-server-16
```

Restarting an instance is equivalent to stopping the instance with a final checkpoint and starting the instance. On instances with a large buffer cache (large number of dirty blocks), the final checkpoint may take a long time to complete. To reduce downtime, perform a checkpoint before restarting the instance, or **wait for a timed checkpoint to complete and stop the instance immediately after the checkpoint completes** .

Example messages from the diagnostic log:

```
[552332] LOG: received fast shutdown request
[552332] LOG: aborting any active transactions
[552340] LOG: pg_wait_sampling collector shutting down
[552332] LOG: background worker "logical replication launcher" (PID 552341) exited with exit code 1
[552334] LOG: shutting down
[552334] LOG: checkpoint starting: shutdown immediate
[552334] LOG: checkpoint complete : wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled;
write=0.016 s, sync=0.005 s, total=0.057 s; sync files=2, longest=0.003 s, average=0.003 s; distance=0 kB,
estimate=0 kB; lsn=44/1A8598B0, redo lsn=44/1A8598B0
LOG: database system is shut down
[552367] LOG: starting Tantor Special Edition 16.6.1 a74db619 on x86_64-pc-linux-gnu, compiled by gcc (Astra
12.2.0-14.astra3+b1) 12.2.0, 64-bit
[552367] LOG: listening on IPv4 address "127.0.0.1", port 5432
[552367] LOG: listening on IPv6 address "::1", port 5432
[552367] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
[552371] LOG: database system was shut down at
[552367] LOG: database system is ready to accept connections
[552375] LOG: pg_wait_sampling collector started
[552374] LOG: autoprewarm successfully prewarmed 947 of 947 previously-loaded blocks
```

When the instance was restarted , active transactions were first terminated , new sessions could not be created, and client application downtime began. The final checkpoint began . After the checkpoint completed , the instance was restarted and the downtime ceased.

4) Install the extension in the postgres database, see what objects are included in the extension, what configuration parameters the extension has:

```
postgres=# create extension pg_wait_sampling;
CREATE EXTENSION
postgres=# \dx+ pg_wait_sampling
Objects in extension "pg_wait_sampling"
Object description
------------------------------------------------
function pg_wait_sampling_get_current(integer)
function pg_wait_sampling_get_history()
function pg_wait_sampling_get_profile()
function pg_wait_sampling_reset_profile ( )
view pg_wait_sampling_current
view pg_wait_sampling_history
view pg_wait_sampling_profile
(7 rows)
postgres=# select name, setting, context, min_val, max_val, boot_val,
pending_restart r from pg_settings where name like '%pg_wait_sampling%';
name | setting | context | min_val | max_val | boot_val | r
---------------------------------+---------+-----------+---------+------------+----------+--
pg_wait_sampling. history_period | 10 | superuser | 1 | 2147483647 | 10 | f
pg_wait_sampling. history_size | 5000 | superuser | 100 | 2147483647 | 5000 | f
pg_wait_sampling. profile_period | 10 | superuser | 1 | 2147483647 | 10 | f
pg_wait_sampling. profile_pid | on | superuser | | | on | f
pg_wait_sampling. profile_queries | top | superuser | | | top | f
pg_wait_sampling.sample_cpu | on | superuser | | | on | f
(6 rows)
```

The extension has a history and a profile of waiting events. These functions are independent of each other.

Wait events may be caught in the profile, but not in history, and vice versa. The probability of catching a wait event is determined only by two parameters: `pg_wait_sampling.profile_period` and `pg_wait_sampling.history_period` . The lower the value, the higher the probability of

catching a rare wait event. The profile does not take data from history, and the pg_wait_sampling.profile_period parameter does not depend on pg_wait_sampling.history_period and pg_wait_sampling.history_size .

5) See how much shared memory the extension uses:

```
postgres=# select * from (select *, lead(off) over(order by off)-off as true
from pg_shmem_allocations) as a where name like '%wait%';
name | off | size | allocated_size | true
-------------------+-----------+-------+----------------+-------
pg_wait_sampling | 148145920 | 17568 | 17664 | 17664
(1 row)
```

Three extension structures use 17664 bytes . Most of it is occupied by a queue (MessageQueue) of a fixed size of 16K, memory for the list of PIDs, memory for identifiers of commands (queryid) executed by processes. The size of the structure for storing the list of PIDs of processes is determined by the maximum number of processes of the instance. The maximum number of processes of the instance depends on the values of the parameters: max_connections, autovacuum_max_workers, max_worker_processes, max_wal_senders .

6) Perform commands :

```
postgres=# select * from pg_wait_sampling_history limit 10;
select * from pg_wait_sampling_history where queryid<>0;
select pg_backend_pid();
pid | ts | event_type | event | queryid
--------+-----------------------------+-----------+---------------------------+---------
552370 | 2025-07-10 13:17:39.562626+03 | Activity | BgWriterHibernate | 0
552372 | 2025-07-10 13:17:39.562626+03 | Activity | WalWriterMain | 0
553742 | 2025-07-10 13:17:39.57326+03 | Client | ClientRead | 0
552373 | 2025-07-10 13:17:39.57326+03 | Activity | AutoVacuumMain | 0
552376 | 2025-07-10 13:17:39.57326+03 | Activity | LogicalLauncherMain | 0
552374 | 2025-07-10 13:17:39.57326+03 | Extension | Extension | 0
552369 | 2025-07-10 13:17:39.57326+03 | Activity | CheckpointerMain | 0
552370 | 2025-07-10 13:17:39.57326+03 | Activity | BgWriterHibernate | 0
552372 | 2025-07-10 13:17:39.57326+03 | Activity | WalWriterMain | 0
553742 | 2025-07-10 13:17:39.583412+03 | Client | ClientRead | 0
(10 rows)

pid | ts | event_type | event | queryid
--------+-----------------------+------------+-------+---------------------
 553742 | 2025-07-10 13:17:43.228602+03 | | | 7356378217053506569
(1 row)

pg_backend_pid
----------------
          553742
(1 row)
```

The second command usually produces zero lines, but if you run it several times, there is a good chance it will produce a line that corresponds to the server process that is servicing the current session.

The extension uses a background process pg_wait_sampling collector, which polls the state of all instance processes at a frequency specified by the pg_wait_sampling.history_period or pg_wait_sampling.profile_period parameter (default 10 milliseconds) and stores 5000 (default pg_wait_sampling.history_size) events .

On an instance with many active sessions, a history of 5000 events can be overwritten in a fraction of a second. The history stores wait events for all processes. If server processes do not encounter locks, then 99.98% of wait events will be filled by background processes and are not related to requests.

7) Make sure there are exactly 5000 events in the history:

```
postgres=# select count(*), max(ts)-min(ts) duration from
pg_wait_sampling_history;
count | duration
-------+-----------------
5000 | 00:00 : 07.572279
(1 row )
```

On an idle instance, history stores only 7 seconds of data .

8) Since the history stores little data, you can use the data from the profile:

```
postgres=# select * from pg_wait_sampling_profile where queryid>0;
pid | event_type | event | queryid | count
--------+-----------+-------------+---------------------+-------
553742 | | | 6003029174018016234 | 2
553742 | IO | DataFileRead | 2425453347952179914 | 1
553742 | | | 7356378217053506569 | 2
553742 | | | 153783292416311340 | 1
553742 | IO | DataFileRead | 6003029174018016234 | 2
(5 rows)
```

The background process accumulates (groups, aggregates) wait events and counts their number (count). This is called the "wait profile" - the number of wait events grouped by:
1) types and kinds of events (event_type, event)
2) processes (pid) if pg_wait_sampling.profile_pid=true
3) types commands (queryid), if pg_wait_sampling.profile_queries=true.

Data is saved for all processes, including server processes of terminated sessions that are no longer present in the operating system. If `pg_wait_sampling.profile_pid=on` , then the number of rows in the view only increases.

9) Run the load in the second terminal:

```
postgres@tantor:~$ pgbench - C -c 90 -T 900 -P 10
pgbench (16.6)
starting vacuum...end.
progress: 10.0 s, 188.2 tps, lat 429.108 ms stddev 469.398, 0 failed
progress: 20.0 s, 196.1 tps, lat 468.844 ms stddev 638.098, 0 failed
```

The -C option generates a large number of sessions.

10) Check that the number of rows in the view only increases:

```
postgres=# select count(*) from pg_wait_sampling_profile;
\watch 10

count
-------
50453
(1 row)

count
-------
55256
(1 row)

^ C
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2025 www.tantorlabs.ru

```
postgres =#
```

The strings are in a memory structure and if the number of strings is large, a lot of memory may be occupied. The extension does not clean up the memory used by this structure.

You need to periodically free up memory by calling `the pg_wait_sampling_reset_profile() function` , or set `pg_wait_sampling.profile_pid=false` .

11) Before performing comparative tests, it is sometimes convenient to reset statistics so that the data accumulated from the previous test does not get into the result. To do this, use the statistics reset functions. Reset all statistics:

```
postgres=#
select pg_stat_reset();
select pg_stat_reset_shared(null);
select pg_stat_reset_shared('bgwriter');
select pg_stat_reset_shared('archiver');
select pg_stat_reset_shared('io');
select pg_stat_reset_shared('wal');
select pg_stat_reset_shared('recovery_prefetch');
select pg_stat_reset_slru(null);
select pg_stat_statements_reset();
select pg_stat_kcache_reset();
select pg_wait_sampling_reset_profile();
select pg_qualstats_reset();

pg_stat_reset
---------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_shared
----------------------

(1 row)

pg_stat_reset_slru
--------------------

(1 row)

pg_stat_statements_reset
------------------------------
2025-02-10 13:53:27.95483+03
(1 row)

pg_stat_kcache_reset
----------------------
```

"Tantor: PostgreSQL 16 Performance Tuning" Astra Group, Tantor Labs LLC © 2024 www.tantorlabs.ru

```
(1 row)

pg_wait_sampling_reset_profile
--------------------------------

(1 row)

ERROR: function pg_qualstats_reset() does not exist
LINE 1: select pg_qualstats_reset();
```

The last function is missing because the `pg_qualstats extension` was not installed.