6-MAR-25

Tantor: PostgreSQL 16 Performance tuning

tantor

Oleg Ivanov

Table of contents

Copyright

Created by : Oleg Ivanov

Created: **6 March 2025**

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

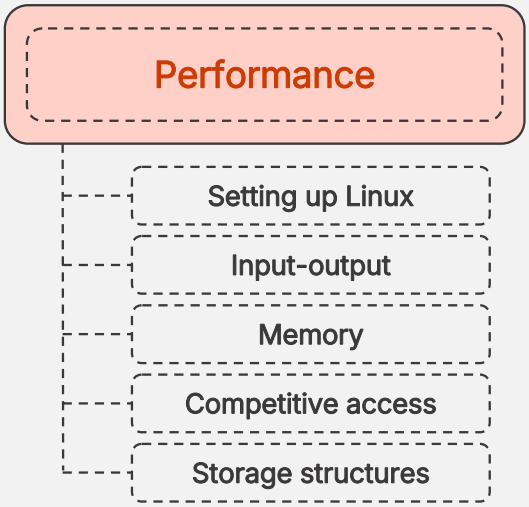
1

Introduction

Performance tuning

Review

- The course covers:
  - Linux operating system
  - Configuring an instance serving a database cluster
  - optimization of storage structures
- the course does not cover:
  - SQL code setup



Review

Well intended for administrators and developers whose tasks include tuning the performance of applications running on the PostgreSQL DBMS .

PostgreSQL database cluster consists of a set of files stored in a file system in a directory (the directory is specified by `the PGDATA environment variable`) . Clients (users) are served by a computer program called a DBMS. A running program is called a PostgreSQL instance . An instance serving a database cluster is a set of processes in the operating system and the RAM used by the processes. The types of hardware resources used by an instance are:

1) " input-output " (disk subsystem, disk, storage system)
2) " memory " (RAM)
3) " processor " (central processor cores)
4) network (network interfaces) .

Resources may be more stressed (scarce) or one resource may become a bottleneck that will determine the performance of the entire instance.

Instance processes share resources and the bottleneck may be resource starvation . Therefore, special attention is paid to blocking and resource access wait events.

The instance serves applications in a client-server mode: applications submit SQL commands , instance processes execute the SQL commands and return the result.

SQL is a declarative language . This means that SQL commands describe what result you want to get (" what "), not how to achieve the result (" how "). There may be several ways to execute commands. For example, a full table scan or an index scan. The methods differ in resource usage. Choosing a method (plan) for executing SQL commands also relates to performance tuning and is called " SQL code tuning " (SQL tuning, query performance tuning). To this topic It is worth moving on after tuning the instance performance and optimizing the data storage structures (tables and indexes), which are covered in this course.

https://en.wikipedia.org/wiki/Starvation_(computer_science)

About Tantor LLC

- since 2016 on the international market
- Tantor DBMS for government and commercial organizations
- development of the Tantor Monitoring Platform and management of the PostgreSQL family of DBMS, as well as Patroni clusters
- many years of experience in operating high-load systems
- is part of the Astra Group of Companies

About Tantor

Since 2016, the Tantor team has worked in the international PostgreSQL DBMS support market and served clients from Europe, North and South America, and the Middle East. The Tantor team developed the Tantor Platform software and subsequently created the Tantor DBMS, based on the program code of the freely distributed PostgreSQL DBMS.

In 2021, the company is concentrated its main activities on the design and development of the Tantor DBMS, as well as the development of the Tantor Platform - a tool for managing and monitoring databases based on PostgreSQL.

The design and development of products is based on many years of accumulated experience in the operation of high-load software systems in the public and private sectors.

In 2002, Tantor LLC joined the Astra Group of Companies.

Tantor DBMS

Tantor BE

New features and improvements compared to PostgreSQL, technical support

Tantor SE

Enterprise-level DBMS, suitable for the most loaded OLTP systems or KHDs up to 100 TB in size

Tantor SE 1C

DBMS for high loads, optimized and approved for working with 1C applications

Tantor PipelineDB

An extension that allows continuous data processing

As part of Tantor xData

Maximum version of DBMS, optimized for working with 1C

Tantor DBMS

Tantor DBMS is a relational database of the PostgreSQL family with increased performance and stability. It is released in several editions (assemblies): BE, SE, SE1C, Certified . Edition Special Edition for the most loaded OLTP systems and data warehouses up to 100 TB in size. Special Edition 1C for "1C" ERP applications .

Technical support, assistance in building architectural solutions, migration from DBMS of other manufacturers (import substitution) are available for all editions . When purchasing the Tantor DBMS, a license for the Tantor Platform is provided for managing the acquired DBMS.

Tantor xData

- hardware and software complex with high performance, fault tolerance, security
- DBaaS capabilities in data centers enterprises
- Improved automation and backup
- high performance and scalability
- reducing infrastructure and administration costs
- includes Tantor DBMS and Tantor Platform

Tantor xData

The Tantor XData hardware and software suite (HSE) delivers large-scale, mission-critical workloads with high performance and availability. Consolidation of diverse Tantor Special workloads Edition on XData database machines in enterprise data centers helps organizations improve operational efficiency, reduce IT administration , and lower costs.

The hardware and software complex (HSC) Tantor XData is designed for migration from foreign manufacturers' complexes and provides similar load capacity. It is a replacement for highly loaded DBMSs up to ~50 TB per instance, servicing OLTP -type loads , running on hardware and software complexes from foreign manufacturers. For DBMSs servicing data warehouses up to ~120 TB per instance.

It is a replacement for heavy ERP from 1C when migrating from DBMS of foreign manufacturers. Allows you to consolidate several DBMS in one PAC. Can be used when migrating from SAP to 1C: ERP.

Designed for creating cloud platforms.

An advantage of using xData is the presence of a convenient graphical system for monitoring the operation of the DBMS: the Tantor Platform.

Tantor PipelineDB

- an extension for the open source Tantor and PostgreSQL DBMS for continuous execution of SQL queries on data streams with incremental storage of results in regular tables
- high performance time series aggregation
- Allows you to connect streaming data with historical data for real-time comparison
- can be used in applications where immediate response is required
- example of a continuous view to display the daily traffic used by the top ten IP addresses :

```
CREATE VIEW heavy_hitters AS
 SELECT day( arrival_timestamp ) , topk_agg ( ip , 10 , response_size )
 FROM requests_stream GROUP BY day
```


Tantor PipelineDB

Tantor PipelineDB is an open source extension for Tantor or PostgreSQL released in 2024. It allows continuous processing of streaming data with incremental storage of results in tables. Data is processed in real time using only SQL queries. Has a large number of analytical functions that work with constantly updated data. Allows you to connect streaming data with historical data for real-time comparison. Eliminates the need to use traditional ETL (Extract, Transform, Load) logic with CDC (Change Data Capture). The essence of the extension is described below for those familiar with the term "CDC" .

Tantor PipelineDB adds support for continuous views. Continuous views are high-refresh materialized views that are incrementally updated in real-time.

Queries on continuous views **instantly** return **up-to-date** results. This **makes it possible to use TantorPipelineDB in the class of applications where immediate response is important** .

Examples of creating continuous views:

Continuous view for providing analytical data for the last five minutes :

```
CREATE VIEW imps WITH (action=materialize, sw = ' 5 minutes ')
AS SELECT count(*), avg (n), max(n) FROM imps_stream ;
```

By default, `the action=materialize parameter` , so the `action parameter` can be omitted when creating continuous views.

Continuous representation for outputting ninetieth, ninety-fifth, ninety-ninth percentiles response time :

```
CREATE VIEW latency AS
 SELECT percentile_cont (array[90, 95, 99])
 WITHIN GROUP (ORDER BY latency::integer)
 FROM latency_stream ;
```

Continuous view to display daily traffic used by top ten IP addresses :

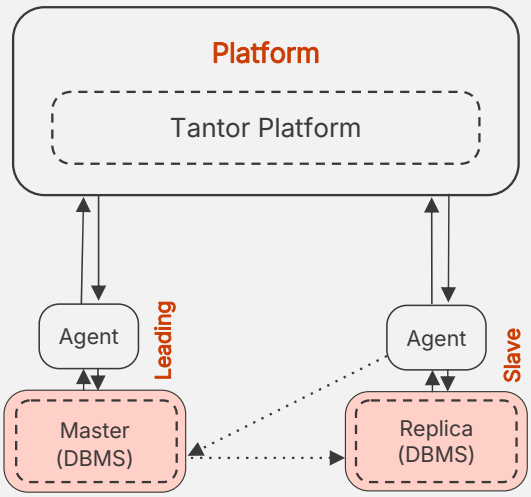
```
CREATE VIEW heavy_hitters AS
 SELECT day( arrival_timestamp ) , topk_agg ( ip , 10 , response_size )
 FROM requests_stream GROUP BY day ;
```

https://tantorlabs.ru/products/pipelinedb

Tantor Platform

- software for managing a large number of DBMS and Patroni clusters
- Tantor DBMS and PostgreSQL forks
- PostgreSQL instance performance metrics , storing and processing metrics, performance tuning recommendations

- integration with mail systems, directory services, messengers



Tantor Platform

Tantor Platform - software for managing Tantor DBMS , forks PostgreSQL , Patroni clusters . Allows convenient management of a large number of DBMS. Refers to the class of software products that includes Oracle Enterprise Manager Cloud Control.

Benefits of using the Tantor Platform : 1. Collection of PostgreSQL instance performance indicators , storage and processing of indicators, recommendations for performance tuning
2. Intuitive and functional graphical interface allows you to focus on PostgreSQL instance performance indicators
3. Automates routine tasks, increasing work efficiency and reducing the likelihood of errors
4. Manages not only the Tantor DBMS, but also other DBMS of the PostgreSQL family
5. Integration with mail systems, directory services, messengers
6. Easy implementation: deploy and put the DBMS under service by the Tantor Platform using Ansible .

Tantor Platform DLH

Tantor Labs also releases the Tantor DLH Platform - software that allows you to organize the process of transforming and loading data using the Extract logic Transformation Load or Extract Load Transform in Tantor DBMS for organizing data warehouses and data marts. Belongs to the class of software products that includes Oracle Data Integrator.

About the course

- In-person or distance learning with an instructor:
  - duration 5 days
  - starts at 10:00
  - lunch break 13:00-14:00
  - end before 17:00 (last day before 15:00)

About the course

The course is designed for full-time or distance learning with an instructor. The course consists of a theoretical part - chapters, practical exercises and breaks. Breaks are combined with practical exercises that students perform independently on a virtual machine prepared for the course.

Approximate schedule:

1) starts at 10:00

2) Lunch break 13:00-14:00. The start of lunch may shift by half an hour in the range from 12:30 to 13:30, as it usually coincides with the break between chapters.

3) the theoretical part ends before 17:00 (on the last day of the course before 15:00).

The course consists of a theoretical part (chapters) and practical assignments. The duration of the chapters is approximately 20-40 minutes. The exact time of the beginning of the chapters and the time for practical assignments is determined by the instructor. The duration of the exercises may vary among different students and this does not affect the effectiveness of assimilation of the course material. You can complete the exercises during breaks between chapters or at the end of each day. The order of the chapters and exercises does not affect the effectiveness of assimilation of the course material. The completion of assignments is not checked. To successfully assimilate the course material, it is enough to:

1) listen to the instructor, looking through the text on the slides and under the slide as the instructor delivers the message

2) ask the instructor questions if internal disagreement arises (questions arise)

3) complete practical tasks and read the text in practical tasks

The course materials include:

1) textbook in `pdf format`

2) practically t tasks in `pdf format`

3) virtual machine image in `ova format`

General concepts of performance tuning

- Performance tuning goals:
  - during commissioning, development, application, selection of equipment
  - migrations from another DBMS
  - during operation when a decrease in target indicators is detected
- Performance tuning includes:
  - targets (metrics) that describe application users' expectations for service quality. Metrics may be specified in a service level agreement
  - procedures for obtaining indicators (monitoring)
  - Performance tuning tools: utilities, extensions, functions, batch files.

General concepts performance settings

The need for performance tuning arises:
1) when putting a program into operation, developing an application, replacing equipment
2) when migrating from a previous version of the DBMS or a DBMS from another manufacturer
3) during operation, when a decrease in target indicators is detected. In this case, there is a " baseline" of performance indicators , when the application works without complaints from users of this application. Baseline indicators define the values of metrics, upon reaching which you can complete the performance tuning. Baseline indicators help to quickly find out what has changed in the configuration of the software system due to which the performance has deteriorated.

Performance tuning types:
1) warning, before performance problems arise (proactive)
2) to fix performance issues (reactive)

Performance tuning includes:
1) Service level agreement (SLA): targets (metrics) that describe application users' expectations regarding the quality of service
2) Daily monitoring of target indicators that determine the quality of service
3) Performance tuning tools: utilities, extensions, functions, batch files.

Source of target indicators:
1) quality of service standards
2) the technical specifications on the basis of which the application was developed or the technical characteristics of the hardware and software systems or the technical requirements for the program
3) an existing SLA that used when migrating an application from a DBMS of another manufacturer.

The target indicators can be formulated broadly. For example: availability 99% of the time ; 90% of requests should be executed in 10 seconds or less. The indicators are not directly related to the DBMS subsystem, whose performance issues prevent the target indicators from being achieved.

Performance Tuning Methodology

- does not depend on the tools used
- includes:
  - assessing the application architecture: how the application interacts with the database
- Errors at the application architecture level create bottlenecks and define the limits within which performance tuning is possible
- If changes can be made to the application architecture to eliminate bottlenecks, then the greatest effect is achieved

Performance Tuning Methodology

The methodology (sequence of actions) does not depend on the tools used. You can use the Tantor Platform, PostgreSQL extensions , SQL commands together .

The methodology includes the following steps:

1) Assess the application architecture: how the application interacts with the database. This step is performed once.

Examples:

a) the application works through caching solutions key-value type (Valkey , Redis), actively uses temporary tables ;

b) the application has tables in which rows are frequently updated. In PostgreSQL DBMS updates (UPDATE command) generate stale rows. Inserting rows (INSERT) does not generate stale rows. An application that primarily inserts rather than updates rows places less load on the DBMS.

c) the application stores data in json format , not in scalar data types ;

d) the application uses asynchronous data processing in the DBMS, and not at the intermediate level (application server). The PostgreSQL DBMS is served by only one instance, and there can be several application servers. At the application server level, it is relatively easy to redistribute the load across several servers.

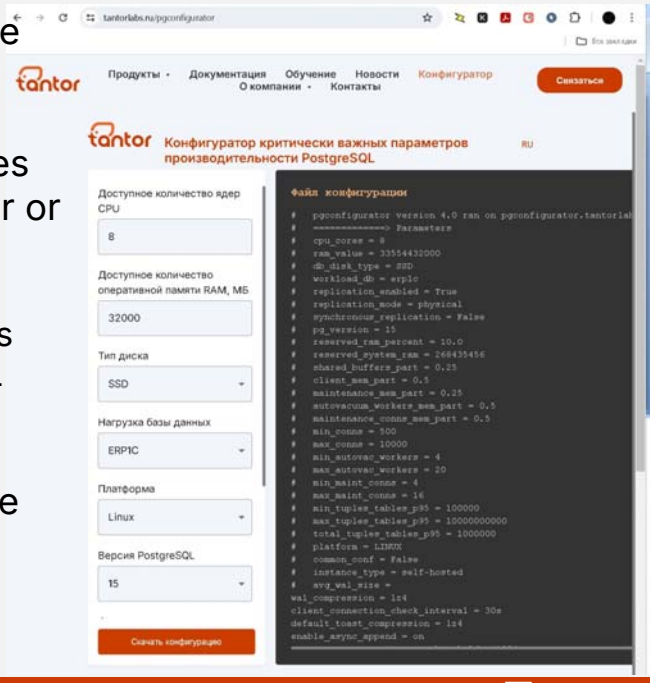
e) using uuid s rather than integer types as primary keys . Primary keys of type int 4 and int8 are filled with a monotonically increasing sequence, and uuids are generated randomly (except for those generated by the uuidv7 () function of the pg_uuidv7 extension available in Tantor 1 6.6) . Inserting records into the primary key index varies in efficiency.

f) a large number of indexes and table sections that were used before migrating to PostgreSQL from DBMSs of other manufacturers. Different DBMSs have features that affect performance, and the optimal number of table sections and the number of indexes per table from a performance point of view may differ in different DBMSs.

Errors at the application architecture level create bottlenecks and define the limits within which performance tuning is possible. If changes can be made to the application architecture to eliminate bottlenecks, then the greatest effect is achieved. For example, transferring long queries to a physical replica significantly unloads the main DBMS (master , primary) .

Sequence of actions for setting up productivity

- Configuring Database Cluster File Storage Parameters
- checking the instance configuration parameters for compliance with the values recommended by the Tantor Configurator or the Tantor Platform
  - Tantor Configurator (http://tantorlabs.ru/pgconfigurator) allows you to perform initial setup of PostgreSQL DBMS configuration parameters
- after a test load or during application operation, check or fine-tune the instance configuration parameters



Performance Tuning Steps

For example, During migration from other types of DBMS, you can reduce the number of sections or transfer large objects (LOB) not in the similar data type " large objects " (using one table per database for storage `pg_largeobject`) , but in files in the file system or columns of `bytea types` or `text` .

2) If you are tuning the performance for the first time or after a significant hardware change, you should check and tune the operating system. The PostgreSQL instance works with hardware resources not at a low level (direct I/O), but through the operating system page cache, and therefore the operation of the instance is significantly affected by the operating system settings. At this step, you check for typical problems that are not related to the DBMS: lack of disk space, use of inappropriate file systems, their mounting parameters .

3) Before setting up the instance, it is worth configure storage structures. If there are storage systems with different characteristics, then place the PGDATA directories in an optimal way. Storage systems usually do not change often, have precise characteristics (volume of space, read and write speed, number of operations per second) and optimization of storage structures is performed once. For example, the `PGDATA/pg_wal directory` can be placed on a separate file system and create a tablespace for temporary objects

4) Perform the initial " instance setup " : check whether the instance configuration parameters are set to values close to optimal. There are hundreds of parameters and they are interconnected: changing the value of one parameter can shift the optimal range of another parameter. Therefore, the setup is iterative. In the first iteration, check that the values are set to the recommended values for the planned load and hardware capabilities. The values can be obtained using the Tantor configurator (https://tantorlabs.ru/pgconfigurator) , recommendations from the PostgreSQL documentation . Example: with a large amount of physical memory, set the buffer cache size to about a quarter of the physical memory.

5) The second iteration of tuning the instance configuration parameters is performed based on the results of a test load or during the operation of the application. In the second iteration, the parameters of the instance subsystems are tuned in an order that reduces the likelihood that after tuning some instance subsystem, you will have to return to what was already tuned.

Sequence of actions for setting up productivity

- defining target indicators, upon reaching which performance tuning is stopped
- selecting the instance subsystem whose tuning efforts will yield the greatest effect
  - the subsystem that serves the most scarce resource is the bottleneck
- measurement of performance indicators related to a subsystem
- making changes to configuration parameters related to a subsystem
  - if the indicators have improved, then the changes in the parameter values were successful
- when the target indicators are reached, the setup is completed

Performance Tuning Steps

The following steps are performed when using the application.

6) The tuning goals and estimated effort are determined (cost to effect analysis). For example, the target indicators are determined upon reaching which it is worth stopping performance tuning. The target indicators can be metrics from the SLA , baseline indicators, parameter values when the application worked with acceptable performance. After eliminating bottlenecks, performance tuning usually has a smaller effect.

Troubleshooting and performance tuning slightly different. When troubleshooting problems that may include performance issues, the degradation in performance is usually abrupt rather than gradual. Typically, the cause is recent changes made to the system: installing updates, changing the topology (for example, reducing the number of physical replicas). When searching for problems, delays on the application server side are excluded, since the cause of the problems may be the application code, not the DBMS.

7) The area (instance subsystem) with the greatest potential for performance tuning is selected. If there is a bottleneck, eliminating it will yield the greatest result from the tuning efforts. The bottleneck is the most scarce resource. For example: CPUs. **CPU load is considered high if CPU load is greater than ~ 90%** (boundary ranges from 85% to 95%). The bottleneck may be locks held longer than the lock type is expected to hold . For example, lightweight locks must be held for several dozen CPU instructions. The bottleneck may be contention for access to buffers occupied by table blocks that are frequently accessed by the application (a " hot " object) .

8) The indicators related to the subsystem are measured ("gather evidence"). The indicators can be compared with the baseline ones. indicators, or with indicators of similar DBMSs in operation.

9) Changes are made to the subsystem parameters. For example, the autovacuum configuration parameters are changed.

10) If the target indicators have improved, then continue to increase or decrease the parameter values. If the indicators have worsened, then stop making changes. If the changes are insignificant, then select another subsystem. When the target indicators are reached, performance tuning stops.

Example why in 6 The paragraph states " When searching for problems, delays on the application server side are excluded, since the cause of the problems may be the application code, and not the DBMS **. "**

web application developer was load testing a PostgreSQL instance to determine the maximum number of queries the instance could handle. The queries were simple: SELECT a single row from an index, similar to the one used by the pgbench utility. With 20,000 " concurrent queries " that the application sent using a pool of 64 database connections, each " query " took between 4 and 10 seconds to complete, with an average of 4.56 seconds. Concurrent queries were defined as queries that were pending from the time the client code sent a request to get a connection from the pool until the row was retrieved and the connection was returned to the pool. To an end user or the web application load testing utility , a " client connection " was the time from when a button was clicked on the web page until the page was rendered . web page (or from the moment a request is sent to the moment a response is received via the REST protocol). The goal of the test was to determine how many " client connections " (" concurrent requests ") the application can handle.

A pool of 8,16,32,64,72,96 connections was tested. The developer called " client connections " " the number of users trying to use the database at the same time ." The developer found that when the number of users trying to use the database at the same time is small, then a smaller number of connections (or just one if there are only a few users) works much better . When reaching 10,000 " simultaneous requests , " a larger number of connections in the connection pool worked better and resulted in better performance. But only up to a certain number of connections (depending on the power of the host). For example, on the developer's MacBook with 32GB of RAM, a pool of 64 sessions gave the best performance. More or less connections resulted in lower performance. The developer found that changing PostgreSQL configuration parameters such as: `shared_memory_size` , `shared_buffers` , `effective_cache_size` , `maintenance_work_mem` , `checkpoint_completion_target` , `wal_buffers` , `random_page_cost` , `work_mem` , `max_wal_size` , `max_worker_processes` , `max_parallel_workers` did not affect the duration of " queries " and PostgreSQL DBMS worked the same as with the default settings . The developer took the list of parameters from " reliable " sources. He did not get to using " hints " . The developer was not an administrator and did not ask the rhetorical question " where is the trace " and did not have time to connect to any processes with a debugger or even delve into LWLock , which saved him time. The developer was not from a large company and did not think that he urgently needed " clustering " , greenplum , stolon , master-master . The developer came to the reasonable conclusion that the only thing that helped to cope with the high load was caching with Redis and other means to reduce the number of requests to the database.

You can't know everything, but you should strive to improve your skills by learning and asking questions. A developer asked a question on the forum and was given the following recommendation. Based on the data provided, for each " request " there are 20,000/64 = 312 requests at the same time. Let's assume that switching connections/contexts in the application server connection pool , sending a request, waiting for the request result, and returning the results takes 10 milliseconds (0.01 seconds). This means that on average, requests wait 0.01 * 312 = 3.12 seconds, which corresponds to an average query execution time (not to the database, but the full query) of 4.56 seconds. This means that the main delays occur on the application server and **the DBMS in its case is not a bottleneck** . The DBMS does not require configuration, which the developer has already seen by changing the configuration parameters. The developer was also recommended to check whether prepared queries are used. Prepared queries allow caching the query execution plan in the server process memory and reducing the planning time. tps=312 is a typical value for simple tests on regular hardware. At 312 queries per second, the planning time is comparable to the query execution time and, perhaps, using prepared queries it will be possible to reduce the planning delay, but the expected improvement is not very large, about 2 times.

1-2

Using the pgbench utility

Benchmarking

- Benchmarking is the process of testing the performance of hardware, software, or an entire system.
  - benchmark - criterion, guideline
- The universal indicator is the number of commands per second (transactions per second , tps), which can be processed by the DBMS
  - if a transaction consists of one command, then this indicator is called the number of queries per second (qps)
  - used to compare performance before and after making changes to the DBMS configuration
- To measure the indicator you need:
  - the set of commands that will be executed in a transaction
  - tables with data and other objects (indexes, integrity constraints, sequences) that are needed to execute commands
  - the number of sessions in which transactions will be executed in parallel

Benchmarking

Benchmarking (benchmark - criterion, guideline) is the process of testing the performance of hardware, software, or the entire system as a whole. In the process of performance tuning, performance indicators are measured. If you are tuning the performance of an instance as a whole, which indicator should you use ?

A simple indicator is the number of commands per second (tps) that the DBMS can process . Conditions can be used when calculating tps . For example, only transactions that were completed in 5 or 20 seconds can be taken into account. Conditions can be absent, in which case commands are sent at maximum speed. Such a test is called a load test.

If a transaction consists of one command, then this indicator is called the number of queries per second (qps) . TPS is used to compare the performance before and after changes to the DBMS configuration. To measure TPS, you need:

1) a set of commands that will be executed in a transaction

2) tables with data and other objects (indexes, integrity constraints, sequences) that are needed to execute commands

3) the number of sessions in which transactions will be executed in parallel.

tps values to be comparable, the above characteristics must be the same.

For a quick check, testing should be fast and the testing utility should be easy to use. PostgreSQL has a command line utility `pgbench` . The utility is extremely easy to use.

To create or recreate tables with data, simply execute the command:

```
pgbench - i
```

Testing for 30 seconds with output of intermediate results every 5 seconds is started by the command:

```
pgbench -T 30 -P 5
```

The result of the work will be given and the main indicator is **tps , number of transactions per second** :

```
latency average = 1.687 ms
stddev latency = 0.225 ms
initial connection time = 3.788 ms
tps = 590 .180430 (without initial connection time)
```

Benchmarking result

- the main thing is this tps in the last line
- standard deviation is derived for latency
- parameter - P allows to output current tps values and latency

```
pgbench -T 30 -P 5
starting vacuum...end.
progress: 5.0 s, 582.0 tps , lat 1.709 ms stddev 0.252, 0 failed
progress: 10.0 s, 597.0 tps , lat 1.667 ms stddev 0.199, 0 failed
progress: 15.0 s, 596.0 tps , lat 1.670 ms stddev 0.274, 0 failed
progress: 20.0 s, 581.8 tps , lat 1.712 ms stddev 0.186, 0 failed
progress: 25.0 s, 601.4 tps , lat 1.655 ms stddev 0.206, 0 failed
progress: 30.0 s, 582.4 tps , lat 1.710 ms stddev 0.213, 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: 17704
number of failed transactions: 0 (0.000%)
latency average = 1.687 ms
stddev latency = 0.225 ms
initial connection time = 3.788 ms
tps = 590 .180430 (without initial connection time)
```


Benchmarking result

What to look for in pgbench results? Example of command output:

```
pgbench -T 30 -P 5
pgbench (17.0)
starting vacuum...end.
progress: 5.0 s, 582.0 tps , lat 1.709 ms stddev 0.252, 0 failed
progress: 10.0 s, 597.0 tps , lat 1.667 ms stddev 0.199, 0 failed
progress: 15.0 s, 596.0 tps , lat 1.670 ms stddev 0.274, 0 failed
progress: 20.0 s, 581.8 tps , lat 1.712 ms stddev 0.186, 0 failed
progress: 25.0 s, 601.4 tps , lat 1.655 ms stddev 0.206, 0 failed
progress: 30.0 s, 582.4 tps , lat 1.710 ms stddev 0.213, 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: 17704
number of failed transactions: 0 (0.000%)
latency average = 1.687 ms
stddev latency = 0.225 ms
initial connection time = 3.788 ms
tps = 590 .180430 (without initial connection time)
```

The main thing is this tps in the last line . This value will change with repeated runs, i.e. the value has a spread. The spread indicators for tps are not displayed. The standard deviation is displayed for the delay (latency) . In the example " accuracy " latency : 0.225 / 1.687*100= 13.33%. Roughly speaking, tps has the same accuracy . Why do we need accuracy ?

If you tune the performance and measure tps before and after tuning, then if two tps differ within the deviation, then the tuning did not affect the operation of the DBMS.

It is also convenient to use the -P parameter . The pgbench utility will output the current tps values. and latency. Visually you can see what the spread in tps values is.

The convenience is that you can run pgbench and while it is running change the instance parameters and monitor tps .

What causes the scatter ? Because of the activity of processes in the operating system. For example, a checkpoint is completed or autovacuum is launched.

pgbench - PostgreSQL benchmarking utility

- command line utility supplied standard with PostgreSQL
- used to create a test load for performance tuning
- The built-in tests use four tables:
  - pgbench_accounts (100 thousand rows) , pgbench_tellers (10 lines) , pgbench_branches (1 row) , pgbench_history (0 lines)

```
create table pgbench_history (tid int, bid int , aid int , delta int , mtime timestamp);
create table pgbench_tellers ( tid int primary key, bid int , tbalance int , filler char(84));
create table pgbench_accounts (aid int primary key, bid int , abalance int , filler char(84));
create table pgbench_branches (bid int primary key, bbalance int , filler char(88));
```

```
pgbench -i -s 100
dropping old tables...
creating tables...
generating data (client-side)...
100000 00 of 100000 00 tuples (100%) done (elapsed 39.06 s, remaining 0.00 s))
vacuuming...
creating primary keys...
done in 85.48 s (drop tables 0.01 s, create tables 0.01 s, client-side generate 39.29 s,
vacuum 10.91 s, primary keys 35.27 s).
```


pgbench - utility benchmarking PostgreSQL

The utility is used for more than just the default simple test. pgbench - This is a high-quality and simple tool for running arbitrary transactions .

The built-in tests use tables created by the " -i " option . By default

four tables pgbench_accounts (100 thousand rows) are created , pgbench_tellers (10 lines) , pgbench_branches (1 row) , pgbench_history (0 lines) :

```
create table pgbench_history ( tid int , bid int , aid int , delta int , mtime timestamp);
create table pgbench_tellers ( tid int primary key, bid int , tbalance int , filler char(84));
create table pgbench_accounts (aid int primary key, bid int , abalance int , filler char(84));
create table pgbench_branches (bid int primary key, bbalance int , filler char(88));
```

What can be changed in the test data ?

1) The - F parameter can be used to set the percentage of block filling (fillfactor) for three tables pgbench_accounts , pgbench_tellers and pgbench_branches . By default, all tables have fillfactor =100% .

2) Parameter -s (scale factor) specifies how many times to increase the number of rows in tables. If -s is 20000 or greater, the aid columns will be of type int8 , not int4 . Example:

```
pgbench - i - s 1 - F 100
```

3) By default, foreign keys are not created.

You can add foreign keys with the command:

```
pgbench - i -I f
creating foreign keys...
```

4) You can specify what actions need to be performed when creating tables:

```
pgbench - i -I dtgvpf
d roping old tables...
creating t ables...
g enerating data (server-side)...
in acuuming...
creating primary keys...
creating primary keys...
```

5) The pgbench_accounts table can be made partitioned by parameters:

--partitions= number_of_partitions

--partition-method=range or hash

pgbench utility parameters can be found in the documentation:

https://docs.tantorlabs.ru/tdb/ru/16_4/se/pgbench.html

Three built-in pgbench tests

- by default pgbench runs the test TPC-B of 7 commands in one transaction:

```
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
CURRENT_TIMESTAMP);
END;
```

- TPC-B is a simple benchmark and does not represent the load created by typical applications running against a DBMS.
- list of built-in tests:

```
pgbench -b list
Available built-in scripts:
      tpcb -like: < builtin : TPC-B (sort of)>
simple-update: < builtin : simple update>
select-only: < builtin : select only>
```

- running a select-only test:

```
pgbench -b select-only -T 10 -P 3
```


Three built-in pgbench tests

By default, pgbench runs a test roughly corresponding to TPC-B, which consists of seven commands in a single transaction:

```
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = : tid ;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history ( tid , bid, aid, delta, mtime ) VALUES (: tid , :bid, :aid, :delta,
CURRENT_TIMESTAMP);
END;
```

Substitution variables are filled with random values.

The test is too simple and does not correspond to the load created by typical applications working with DBMS. Why is it needed then, why is it not complex ? **The test allows you to determine the maximum achievable tps .**

`CURRENT_TIMESTAMP` function returns the value at the beginning of the transaction, equivalent to `the transaction_timestamp () , now () functions` and differs from `the clock_timestamp () , statement_timestamp () functions` . When writing your commands, this should be taken into account in order to get the time you need. The `now () function is often used` because the function name is short.

List of built-in tests in pgbench :

```
pgbench -b list
Available built-in scripts:
      tpcb -like: < builtin : TPC-B (sort of)>
  simple-update : < builtin : simple update>
select-only: < builtin : select only>
```

simple-update test consists of three teams:

```
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
INSERT INTO pgbench_history ( tid , bid, aid, delta, mtime ) VALUES (: tid , :bid, :aid,
:delta, CURRENT_TIMESTAMP);
```

select-only consists of one query:

```
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

The tests are selected by the parameter `- b` :

```
pgbench -b select-only -T 10 -P 3
```

pgbench launch options

- `-P seconds` specifies the interval in seconds after which the utility will output a line with execution statistics.
- `-T seconds` test duration
- `--protocol=prepared`. In `prepared mode`, the pgbench utility transmits the command once in each session (prepares) and then transmits only new parameter values and the command to execute

```
pgbench -T 10 -P 3
        ...
        progress: 10.0 s, 530.1 tps, lat 1.883 ms stddev 1.558, 0 failed
        progress: 20.0 s, 427.8 tps, lat 2.319 ms stddev 7.186, 0 failed
        progress: 30.0 s, 174.9 tps, lat 5.753 ms stddev 25.774, 0 failed
        ...
        latency average = 2.645 ms
        stddev latency = 11.181 ms
        ...
        tps = 377.649988 (without initial connection time)
```


pgbench launch options

- `P` seconds specifies the interval in seconds after which the utility will output a line with execution statistics.
- `T` seconds sets the test duration:

```
pgbench -T 30 -P 10
...
progress: 10.0 s, 530 .1 tps , lat 1.883 ms stddev 1.558, 0 failed
progress: 20.0 s, 427 .8 tps , lat 2.319 ms stddev 7.186, 0 failed
progress: 30.0 s, 174 .9 tps , lat 5.753 ms stddev 25.774, 0 failed
...
latency average = 2.645 ms
stddev latency = 11.181 ms
...
tps = 377 .649988 (without initial connection time)
```

The main result is tps - the number of transactions per second. **latency** correlates with **tps** and indicates the execution time of a transaction or script . Unlike TPC-C, the TPC-B test does not have artificial delays that increase the number of clients (threads), complicate the testing logic and interpretation of the results. TPC-B executes commands at maximum speed. and defines the system limits .

By default, commands are executed without using binding variables, in raw text. Each command is parsed and an execution plan is built. But you can use the advanced mode and the mode with prepared commands by setting the parameter

`--protocol=extended` or `prepared`. In `prepared mode`, the pgbench utility passes the command once in each session (prepares) and then passes only new parameter values and the command for execution. In this case, the execution plan cached at the session level is used, which is usually faster, since the command is not parsed again.

In `extended mode` the command (including the parameter names `$1, $2, ...`) is sent for execution each time. This mode does not provide any advantages, only overhead. Binding variables are passed in the same call, but as a separate parameter, just like in the psql utility :

```
select $1 as id, $2 as s \bind 5 'b' \g
```

or

```
insert into t values($1, $2) \bind 5 'b' \g
```

Recommendations for using pgbench

- `-c N` specifies the number of parallel sessions in which transactions or scripts will be executed. Sessions are created by one pgbench process by default with one thread . `The -j N` parameter you can specify the number of threads
- the number of sessions must be no less than the number of threads
- when using `- c N` with built-in tests, it is advisable to create tables with `the -s M parameter` , so that `M` there were no less than N
- parameter `- f script.sql @10` you can send your own set of commands saved in a file for execution. In this case, the " transaction " in the pgbench report will be considered the execution of all commands in the file

```
pgbench -T 10 -P 5 -s 3 -c 50 -j 80
pgbench: warning: scale option ignored , using count from pgbench_branches table (100)
scaling factor: 100
Number of clients: 50
Number of threads: 50
```


Recommendations for using pgbench

By default, pgbench creates one session with the database. In practice, DBMSs serve tens or hundreds of sessions. To determine whether the DBMS parameter settings will affect its ability to serve a large number of sessions, the `-c N parameter is used` . The parameter can also be used for other purposes. For example, determining whether the DBMS can serve a given number of active sessions and with what tps . To obtain tps in one session, you need to divide the tps returned by the utility by the number of sessions.

Parameter `-c N` specifies the number of parallel sessions in which transactions or scripts will be executed. Sessions are created by one pgbench process by default with one thread . `The -j N` parameter you can specify the number of threads . With a large number of sessions, the pgbench process can become a bottleneck, since it will use one core. Several threads can use several processor cores. The number of sessions must be no less than the number of threads : `c>j` .

When using -c N with the built-in tests, it is advisable to create tables with the -s M option so that M is not less than N , otherwise the result will be affected by the wait for row-level locks to be acquired, since there is a high probability that UPDATE commands in different sessions will collide on the same row.

If you don't remember which `-s M` created tables, then `M` equal to the number of rows in the `pgbench_branches table` . When testing `-s M` there is no point in asking (`-s` must be specified when creating tables, that is, with the parameter - i), about which a warning will be issued:

```
warning: scale option ignored, using count from pgbench_branches table
```

The results are affected by the execution of checkpoints, maintaining the database horizon, and starting the autovacuum.

Parameter `- f` file you can send your own set of commands saved in the file for execution. In this case, the " transaction " in the pgbench report will be considered the execution of all commands in the file.

You can specify several scripts and an integer that specifies the weighting factor: in what proportion the scripts will be launched:

```
pgbench -f a.sql @8 -f b.sql @2
```

Script `a.sql` will be run 4 times more often than `b.sql` . The default is 1.

When using a table structure, a set of indexes, and commands close to a real application, pgbench allows you to qualitatively test changes in instance parameters.

Example of using pgbench

- task: check which is faster `count( * ), count( 1 ), count( pk )`
- creating a table with data:

```
drop table if exists t;
create table t( pk bigserial , c1 text default ' aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ');
insert into t select *, 'a' from generate_series (1, 100000);
alter table t add constraint pk primary key ( pk );
```

- creating scripts:

```
echo " select count( * ) from t; " > count1.sql
echo " select count( 1 ) from t; " > count2.sql
echo " select count( pk ) from t; " > count3.sql
```

- running tests:

```
pgbench -T 30 -f count1.sql 2> /dev/null | grep tps
tps = 74
pgbench -T 30 -f count2.sql 2> /dev/null | grep tps
tps = 66
pgbench -T 30 -f count3.sql 2> /dev/null | grep tps
tps = 54
```

- result: `count(1)` faster than `count( pk )` by ~ 18%, `count(*)` faster `count(1)` by ~10%

Example of use pgbenc h

Let's look at an example of using pgbench. The task is to check what is better to use: `count(*), count(1), count(c)` when working with PostgreSQL?

First step: create a table for the test :

```
drop table if exists t;
create table t( pk bigserial , c1 text default ' aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ');
insert into t select *, 'a' from generate_series (1, 100000);
alter table t add constraint pk primary key ( pk );
analyze t;
```

Creating files with queries that will be compared:

```
echo "select count(*) from t;" > count1.sql
echo "select count(1) from t;" > count2.sql
echo "select count( pk ) from t;" > count3.sql
```

Executing tests:

```
pgbench -T 300 -f count1.sql 2> /dev/null | grep tps
tps=74
pgbench -T 300 -f count2.sql 2> /dev/null | grep tps
tps=66
pgbench -T 300 -f count3.sql 2> /dev/null | grep tps
tps=54
```

Result: `count(*)` faster `count(1)` by ~10%, `count(1)` faster `count(c)` by ~ 18%,

Count by primary key `count(c)` slowest of all .

The command `explain (analyze) select count(1) from t;` gives a result with a large spread.

The result is consistent with the results of other researchers of the question " `COUNT(*)` vs `COUNT(1)` ":

https://blog.jooq.org/whats-faster-count-or-count1/

plpgsql block loop was used for testing . In our example, pgbench was used.

https://gist.github.com/lukaseder/2611212b23ba40d5f828c69b79214a0e

1-3

Using sysbench and fio utilities

sysbench - performance testing utility

- in testing SQL commands the functionality is similar to pgbench
- testing is done using a simple table:

```
\d sbtest1
Table "public.sbtest1"
Column | Type | Collation | Nullable | Default
--------+-------+-----------+----------+-------------------------------------
id | integer | | not null | nextval ('sbtest1_id_seq':: regclass )
k | integer | | not null | 0
c | character(120) | | not null | '':: bpchar
pad | character(60) | | not null | '':: bpchar
Indexes:
"sbtest1_pkey" PRIMARY KEY, btree (id)
"k_1" btree (k)
```

- has several scripts for testing:

```
ls / usr /share/sysbench
bulk_insert.lua oltp_insert.lua oltp_read_write.lua oltp_write_only.lua
oltp_common.lua oltp_point_select.lua oltp_update_index.lua select_random_points.lua
oltp_delete.lua oltp_read_only.lua oltp_update_non_index.lua select_random_ranges.lua
sysbench --db-driver= pgsql -- pgsql -port=5432 -- pgsql -db=postgres -- pgsql -user=postgres
-- pgsql -password=postgres -- table_size =100000 / usr /share/sysbench/oltp_read_only.lua
prepare
```


sysbench - performance testing utility

pgbench utility does not have the ability to test the main resources used by the DBMS: processor speed (`cpu`) , memory access (`memory`) , file system (`fileio`) . This is useful when comparing hardware or reconfiguring Linux . The sysbench utility is convenient for testing resources . The utility is available in Astralinux and other linux distributions .

Sysbench was developed for load testing of the MySql DBMS . Currently, it has tests for PostgreSQL. In testing SQL commands, the functionality is similar to pgbench.

Installing sysbench :

```
sudo apt install sysbench
```

Available scripts for testing DBMS (SQL command sets) :

```
ls / usr /share/sysbench
bulk_insert.lua oltp_insert.lua oltp_read_write.lua oltp_write_only.lua
oltp_common.lua oltp_point_select.lua oltp_update_index.lua select_random_points.lua
oltp_delete.lua oltp_read_only.lua oltp_update_non_index.lua select_random_ranges.lua
```

You can create your own tests in lua language .

Testing is performed using one table:

```
\d sbtest1
 Table "public.sbtest1"
Column | Type |Collation| Nullable | Default
--------+-------+---------+----------+-------------------------------------
id | integer | | not null | nextval ('sbtest1_id_seq':: regclass )
k | integer | | not null | 0
c | character(120) | | not null | '':: bpchar
pad | character(60) | | not null | '':: bpchar
Indexes:
"sbtest1_pkey" PRIMARY KEY, btree (id)
"k_1" btree (k)
```

When initializing tests, you can specify the number of such tables and the number of rows in the tables:

```
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
```

After creating the tables (prepare) you can execute tests :

```
sysbench --db-driver= pgsql -- pgsql -port=5432 -- pgsql -db= postgres -- pgsql -user= postgres -- pgsql -password= postgres --threads=10 --time=60 --report-interval=5 / usr /share/sysbench/ oltp_read_only.lua run
```

Parameter `--time` (analog of the `-T parameter` for the pgbench utility) sets the test time in seconds ; `--report-interval` (analog of the `-P parameter` for the pgbench utility) interval for outputting lines with statistics ; `--threads` (analog of `the -j -c parameters` (in the pgbench utility) .

Deleting created objects:

```
sysbench --db-driver= pgsql -- pgsql -port=5432 -- pgsql -db= postgres -- pgsql -user= postgres -- pgsql -password= postgres / usr /share/sysbench/oltp_read_only.lua cleanup
```

Using sysbench to test CPU, memory, disks

- built-in tests: cpu , memory, fileio , threads, mutex
- testing the speed of multiple threads:

```
sysbench cpu run --time=10 --threads= 4 | grep sec
events per second: 4709.35
sysbench cpu run --time=10 --threads= 8 | grep sec
events per second: 4684.77
```

- optimal value --num-threads number of CPU cores
- operating speed (reading or writing) with memory:

```
sysbench memory run --memory-block-size= 4K --time=10 -- memory- oper =read -- memory-access-
mode=sequential --memory-scope=local --threads=2 | grep transf
5673.72 MiB transferred (567.27 MiB /sec)
sysbench memory run --memory-block-size=2M ...
102400.00 MiB transferred (38767.60 MiB /sec)
```


Using sysbench to test CPU, memory, disks

The utility is successfully used for testing and comparing central processors:

```
sysbench cpu run --time=10 | grep sec
events per second: 1218.32
sysbench cpu run --time=10 --threads=4 | grep sec
events per second: 4709.35
sysbench cpu run --time=10 --threads=8 | grep sec
events per second: 4684.77
```

The optimal value of `--num-threads` is the number of CPU cores (in the example 4).

Memory performance:

```
sysbench memory run --memory-block-size=4K --time=10 --memory- oper =read --memory-
access-mode= seq --memory-scope=local --threads=2 | grep transf
5673.72 MiB transferred (567.27 MiB /sec)
sysbench memory run --memory-block-size=8K ...
11331.38 MiB transferred (1132.94 MiB /sec)
sysbench memory run --memory-block-size=2M ...
102400.00 MiB transferred (38767.60 MiB /sec)
sysbench memory run --memory-block-size=1G ...
102400.00 MiB transferred (23047.27 MiB /sec)
```

The product of `-- threads` and `-- memory-block-size` must not exceed free physical memory ; `--memory-scope=local` each thread works with its own part of the memory. Threads will use MMU (memory management unit, device for accessing the main memory) its core. There is a performance dependence on the number of threads and the size of the memory chunk. Multithreading testing parameters :

The number of threads (`-- threads`), the number of locks (`-- thread-locks`), and the number of times a thread should execute its workload according to the algorithm: lock - > yield -> work -> unlock (`-- thread-yields`).

```
sysbench threads run --time=10 --thread-locks=1 --threads=4 | grep events:
Total number of events: 1453
sysbench threads run --time=10 --thread-locks=1 --threads=1 | grep events:
Total number of events: 2775
sysbench threads run --time=10 --thread-locks=4 --threads=4 | grep events:
Total number of events: 7697
```

Testing hardware resources

- starting threads and running them simultaneously:

```
sysbench threads --threads= 128 --time=10 run | grep events:
Total number of events: 1502
sysbench threads --threads= 8 --time=10 run | grep events:
Total number of events: 3991
sysbench threads --threads= 4 --time=10 run | grep events:
Total number of events: 8278
sysbench threads --threads= 1 --time=10 run | grep events:
Total number of events: 2773
```

- test of working with files:

```
sysbench fileio --file-total-size=128M --file-num=8 prepare
134217728 bytes written in 1.19 seconds (107.12 MiB /sec).
sysbench fileio --file-block-size=8K --file- fsync -mode= fdatasync -- file- fsync -end=on --
file-total-size=128M --file-num=8 --file-test-mode= rndrw --max-time=10 run | grep /s
reads/s: 435.92
writes/s: 290.61
    fsyncs /s: 939.62
read, MiB /s: 6.81
written, MiB /s: 4.54
events ( avg /stddev): 16613.0000/0.00
execution time ( avg /stddev): 9.8619/0.00
```

- test of lightweight lock usage (mutex)

```
sysbench mutex run -- mutex -num=9000000 | grep exec
execution time ( avg / stddev ): 0.1511/0.00
```


Testing hardware resources

Starting threads and running them simultaneously:

```
sysbench threads --threads=128 --time=10 run | grep events:
Total number of events: 1502
sysbench threads --threads=8 --time=10 run | grep events:
Total number of events: 3991
sysbench threads --threads=4 --time=10 run | grep events:
Total number of events: 8278
sysbench threads --threads=1 --time=10 run | grep events:
Total number of events: 2773
```

Test of lightweight locks (mutex). Each thread does a simple action like incrementing a number in a loop (`-- mutex-loops`), then the thread takes a random mutex (one of `-- mutex-num`), increments the variable, and releases the mutex . This is repeated `-- mutex-locks` number of times.

```
sysbench mutex run -- mutex -num=9000000 | grep exec
execution time ( avg /stddev): 0.1511/0.00
```

Test of working with files:

```
sysbench fileio --file-total-size=128M --file-num=8 prepare
134217728 bytes written in 1.19 seconds (107.12 MiB /sec).
sysbench fileio --file-block-size=8K --file- fsync -mode= fdatasync --file- fsync -end=on
--file-total-size=128M -- file-num=8 --file-test-mode= rndrw --max-time=10 run | grep /s
reads/s: 563.16
writes/s: 375.44
    fsyncs /s: 75.19
read, MiB /s: 4.40
written, MiB /s: 2.93
events ( avg /stddev): 10253.0000/0.00
execution time ( avg /stddev): 9.9455/0.00
sysbench fileio cleanup
```

The `--file-test-mode=` parameter value is `rndrw` random read-write. The parameters `--file-total-size=128M --file-num=8` define the file size. PostgreSQL uses 16MB for WAL files and 1 Gb for data files , fdatasync for WAL and fsync synchronization for data files.

The utility does not give all parameters by --help . Description of parameters should be looked at : man sysbench.

Testing I/O with Flexible IO Tester (fio)

- `sysbench` uses test files filled with zeros
- for testing input-output, a utility is usually used `fio`
- Main parameters when testing equipment for PostgreSQL :
  - `bs =8k` PostgreSQL reads and writes in 8K blocks by default
  - `direct=0` PostgreSQL by default works with files through the page cache

```
sudo apt update && apt install fio -y
sudo fio -- rw = rw -- rwmixread =75 --size=16m --directory=/ -- fadvise_hint =0 -- blocksize =8k
--direct=0 -- numjobs =1 -- nrfiles =1 --runtime=5s -- time_based -- exec_prerun ="echo 3 >
/proc/sys/ vm / drop_caches " -- group_reporting --name=test1
...
  read : IOPS=15.2k , BW= 119MiB/s ( 124MB/s )(594MiB/5011msec)
     lat ( usec ): min=3, max=51909, avg = 53.39 , stdev =860.06
   bw ( KiB /s): min=110946, max=147129, per=100.00%, avg = 121645 .70, stdev =12274.53,
samples=10
   iops : min=13868, max=18391, avg = 15205 .60, stdev =1534.21, samples=10
  write : IOPS=5045 , BW= 39.4MiB/s ( 41.3MB/s )(198MiB/5011msec); 0 zone resets
     lat ( usec ): min=4, max=137, avg = 8.57 , stdev =4.52
   bw ( KiB /s): min=37184, max=48542, per=100.00%, avg = 40428 .10, stdev =3900.63, samples=10
   iops : min= 4648, max= 6067, avg = 5053.30 , stdev =487.36, samples=10
  cpu : usr =23.27%, sys=5.33% , ctx =2279, majf =0, minf =14
```


Testing I/O with Flexible IO Tester (fio)

When analyzing the performance of the disk subsystem, `sysbench` uses test files filled with zeros. The equipment can optimize the work with such files. For accurate I/O testing, use the **`fio`** (Flexible IO Tester) and **`flashbench utilities`** . I/O performance can be limited by the I/O bus: the number of PCIe lines .

Installing the utility:

`apt install fio -y`

Parameters:

`bs =8k` PostgreSQL reads and writes in 8K blocks by default

direct=0 PostgreSQL by default works with files through the page cache. For comparison storage systems can use direct=1

`numjobs` - the number of threads that fio is running loads the system. Optimally, the value should be equal to the number of cores or hardware threads

`iodepth` - command queue depth. The numjobs and iodepth parameters are increased to obtain maximum IOPS values, since one thread is unlikely to utilize the entire PCIe bus bandwidth . A large iodepth value can load the processor core up to 100% and the core will become a bottleneck.

`rwmixread =75` read / write ratio. For OLTP : 80/20 or 75/25

`fadvise_hint =0` sets the POSIX_FADV_DONTNEED hint

`exec_prerun ="echo 3 > /proc/sys/ vm / drop_caches "` Before restarts, clean page cache pages and slab structures it is worth freeing

`size=1G` maximum size of data file in PostgreSQL 1GB . There must be free space in the directory where the files will be created

`rw = randread , read , write, randwrite , randread`

`filename` path to a file or block device for testing. Do not specify a block device with a file system for write tests (`readwrite , randrw , write, trimwrite`) the file system will be corrupted (the contents of the device will be erased). Also, when specifying a file, the data in it is overwritten.

`ioengine = libaio` the fastest (because it is non-blocking), `psync` by default

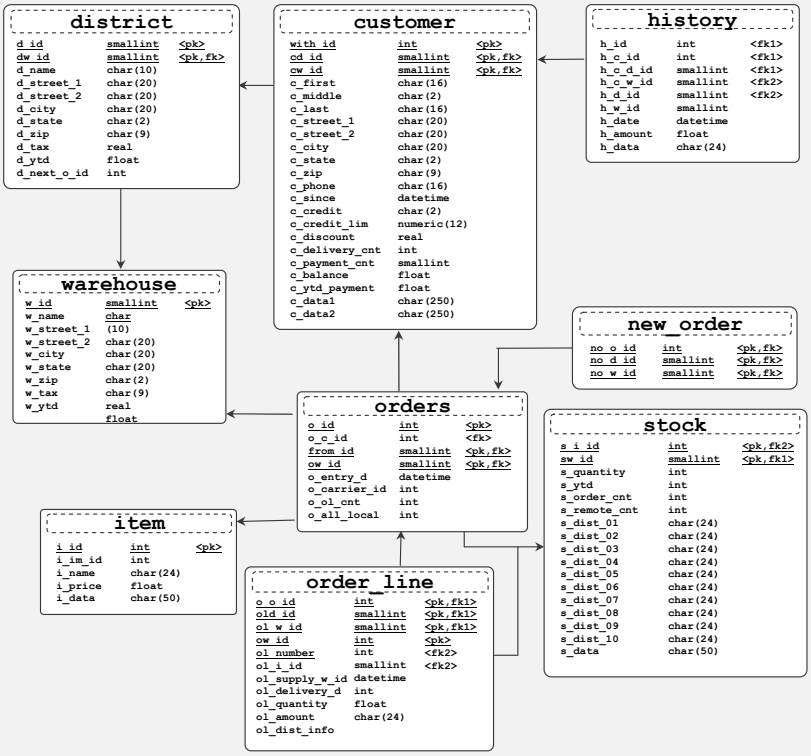
Main metrics: iops , bw - data transfer rate, latency , cpu (usr , sys)

1-4

TPC tests

TPC-B and TPC-C tests

- In TPC-B, commands are executed on the database side at maximum speed
- TPC-B test emulates the operation of a bank
- TPC-C test emulates the operation of a wholesale warehouse
- transaction execution time in TPC-C is 5 seconds for all transactions except one where the execution time is up to 20 seconds
- the main result is tpmC - number of transactions per minute



TPC-B and TPC-C tests

In 1988, the Transaction Processing Performance Council was formed from 26 companies and the TPC-A and TPC-B benchmark specifications were approved . These benchmarks differed in that the former required client emulation (" terminals ") with a delay in client response time. In TPC-B , commands are executed on the database side at maximum speed. In both tests, the work of a bank was emulated. In the TPC-C test , the work of a wholesale warehouse was emulated. This reflected the fact that DBMSs began to be actively used by trading companies, and banks had already been automated. The unit of scaling (scale factor) in TPC-C is a warehouse (table Warehouse). Each warehouse has ten districts (table **district**), equipped with a " terminal " (client program of the DBMS) . Each terminal is designed to register orders (table **orders**) . Each order (trade transaction) consists of positions (table **order_line**) of one product sold (table **stock**). In each district, no more than 1.2 orders can be made per minute. As soon as this limit of orders per minute (tpmC) is reached, according to the test conditions, another warehouse is added (table **warehouse**) . Warehouses are created in advance and transactions are carried out on these warehouses during testing. While 90% of transactions are carried out in less than 5 seconds, the use of warehouses continues. In addition to orders (45% of the total load), transactions for receiving payments with updating the client's balance (43%), checking the status of the last order of any one client (4%), checking the quantity of goods in the warehouse (4%), and generating delivery requests (4%) are simultaneously performed. The request includes orders placed at the time of generating the request. The threshold time for generating a delivery request is 20 seconds. The execution time of each trade transaction (request, order), which corresponds to a database transaction consisting of one or more SQL commands , is calculated separately and the distribution of values can be used as one of the test results. The main result is tpmC - number of transactions per minute. Other non-technical results: transaction cost (depends on the cost of licenses for the DBMS or hardware and software complex), estimated value of how many watts of electricity will be spent on a thousand transactions per minute (W/ ktpmC) . TPC-R tests for reports, TPC-D for OLAP , TPC-W for orders in the online store have not become widespread . Based on TPC-D, a more successful TPC-H test was created for data warehouses and analytical queries ("OLAP load "). The number of tables is 8, integrity constraints are 17. In TPC-H, nominations were allocated for the sizes of processed data from " up to 100 GB " to 30-100 TB.

TPC - E Test , Network Failure Resilience Testing

- TPC-E benchmark for OLTP , appeared in 2006
- instead of a wholesale warehouse, the work of a brokerage company is described, the number of tables is 33 (TPC-E) , instead of 9 (TPC-C)
- 33 primary keys instead of 8, 50 foreign keys instead of 9
- Due to the complexity of reproducing and implementing TPC tests , simple utilities are in demand
- PostgreSQL family databases the pgbench utility is used

TPC-E test, network failure resilience testing

In 2006, the TPC-E test for OLTP appeared, replacing TPC-C . Instead of a wholesale warehouse, the work of a brokerage company is described, the number of tables is 33, instead of 9. There are 33 primary keys instead of 8, 50 foreign keys instead of 9. Added data types for boolean columns and lob. The metric became known as tpsE .

Because of the complexity of reproducing and implementing TPC tests , simple utilities are in demand that measure simple metrics with repeatability of results and confidence intervals (acceptable data spread). For PostgreSQL, pgbench is used .

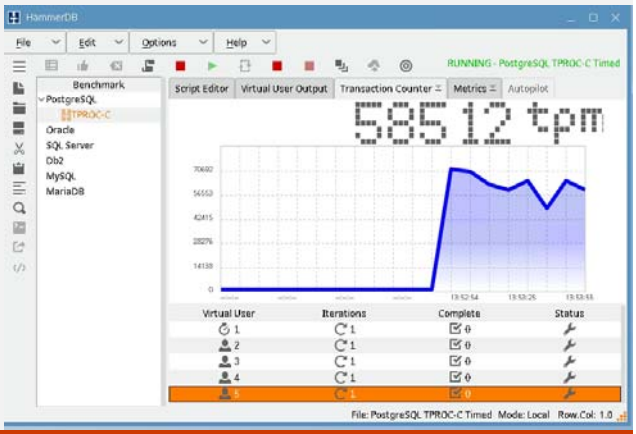
Clients connect to databases over the network. Packets (messages at some network level) can be lost, the order of packet delivery can change, packets can be duplicated. The reasons can be: network card failure, driver errors, lack of memory, equipment that prevents network traffic from passing. To test the resilience of databases to such errors, jepsen is used. framework. For PostgreSQL , it was found that if the client received a confirmation of transaction commit, then in 100% the transaction is committed and durability is ensured. If the client received an I / O error message , then the transaction may or may not be committed. The probability of such an event is rare. The **2PC protocol does not protect against this type of error** , the E3PC protocol provides protection against network failures . However, these protocols reduce performance.

Oracle Database, starting with version 12, has an Oracle Transaction Guard option to protect against this type of failure. Redis and MongoDB lost a significant percentage of data (**https://www.infoq.com/articles/jepsen/**). Technologies that promise high performance should be used taking into account fault tolerance. Suitable and proven solutions should be selected for specific tasks. For example, in tasks where guaranteed response time (transaction execution) is important, real-time databases are used. In cases where transaction losses are unacceptable (finance), Oracle Database and PostgreSQL are used.

When measuring performance, transaction execution time (may be called latency , response time) is a stand-alone metric .

Implementation of TPC-C test

- uses tpmC as the measurement result the number of " transactions " of this test per minute
- can be used to compare the performance of DBMSs from different manufacturers
- not suitable for assessing the impact of changes to the instance configuration
- the disadvantage of the test is that the test rules are overcomplicated , which leads to an increase in the amount of data in the tables, the number of clients and the memory they consume
- uses a set of SQL commands an order of magnitude more complex than TPC -B and uses 9 tables
- HammerDB application includes TPC-C and TPC-H benchmark variants



Implementation of TPC-C test

The TPC-C test uses a set of commands that is an order of magnitude more complex than TPC-B and uses 9 tables : Warehouse , District , Customer , Order , New - Order , History , Item , Stock , Order - Line . It uses 5 transaction types: New order , Payment , Order status , Delivery , Stock level , which are randomly selected in a proportion specified by the test.

Both tests simulate OLTP workload: short transactions with selection and updating of a small number of rows. The disadvantage of the test is the overcomplication of the test rules, which leads to an increase in the amount of data in the tables, the number of clients and the memory they consume ; there is no program code, only a description of the test. TPC-C uses tpmC as the measurement result the number of " transactions " of this test per minute. The test can be used to compare the performance of DBMS from different manufacturers. It is not suitable for assessing the impact of changes in the instance configuration.

applications implementing TPC tests , most of them are not working. One of the working programs is HammerDB , it includes TPC-C and TPC-H tests. The application's website publishes the test results. Results performed in HammerDB are comparable to each other.

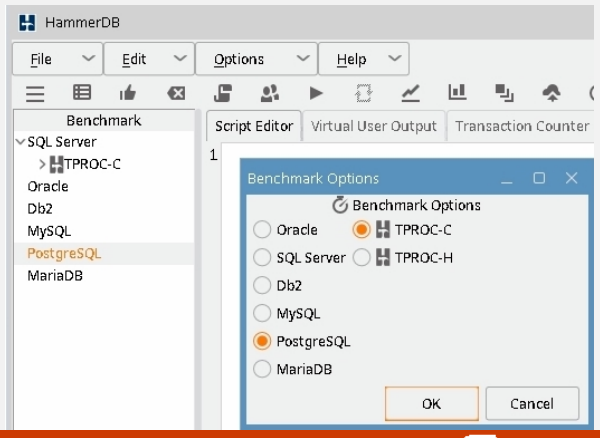
TPC-H benchmark is designed for data warehouses and includes 22 queries called Q1 ... Q22 . The TPC-H benchmark does not change the data in the tables, so it is suitable for repeated runs without recreating the tables.

Queries **Q17** and **Q20** correlated, are the most difficult for any DBMS . If the DBMS operation with data warehouses is optimized, it is probably worth paying attention to optimizing the execution of these particular queries. The clickhouse DBMS specializing in analytics cannot be in 2024 execute these two queries.

https://habr.com/ru/companies/ydb/articles/801587/

HammerDB Application

- implements a subset of the full TPC-C specification, modified to make the workload simpler and easier to execute
- HammerDB Type- C test results are not comparable to other tests that use the tpmC metric
- the results are comparable if they are run in HammerDB
- The difference with TPC-C is that by default HammerDB runs without input and reflection delays. The Type-C test runs the TPC-C workload without delays.
- HammerDB test result type -C: TPM and NOPM



HammerDB Application

The application implements a test based on the TPC-C specification, but does not implement the full TPC-C test rule specification by default.

HammerDB results are not comparable to officially published TPC-C benchmarks . Official TPC-C benchmarks are extremely expensive, labor-intensive, and complex. HammerDB is designed to allow a TPC-C benchmark to be run at low cost on any system, providing professional, reliable, and predictable load testing for all database environments. HammerDB results are not comparable to other benchmarks that use the tpmC metric . However, the results produced by the application are comparable to each other.

In the test results, HammerDB provides two metrics for comparison with other HammerDB test results : TPM and NOPM. NOPM is the number of new orders per minute. NOPM can be used to compare the performance of different types of DBMS.

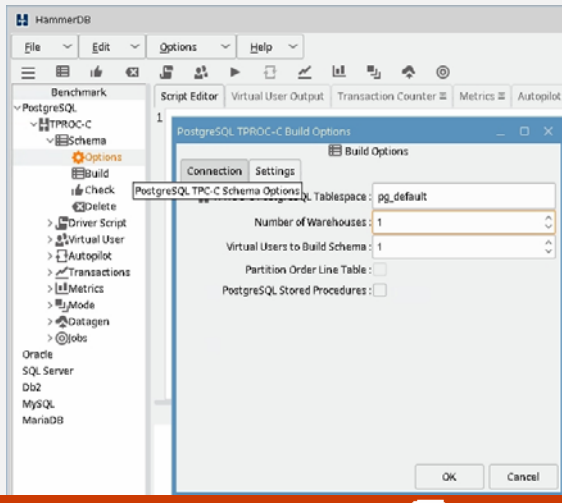
HammerDB can be considered as a subset of the full TPC-C specification, modified to simplify and facilitate the workload execution. The main similarities are the definition of the storage schema and the data itself, as well as 5 transactions implemented as stored procedures. The main difference is that by default HammerDB runs without input and think delays. This means that HammerDB TPROC-C runs the TPC-C compliant workload without delays, maximizing the load on the DBMS . The number of users and the required data volume on which maximum performance is achieved will be much smaller than in a full TPC-C implementation.

HammerDB does not implement " terminals " as the full specification does. This eliminates the need for large numbers of clients and huge amounts of data, while still providing a robust test of the capabilities and performance of relational databases.

Parameters for the HammerDB Type -C test

- The official TPC-C benchmark has a fixed number of users per storage and uses input and think time so that the workload generated by each user is not intensive

- HammerDB does not use input and think time by default, and therefore the number of Virtual Users is approximately equal to the number of cores on the host running the DBMS.
- 4-5 warehouses per Virtual User will be the minimum value to ensure an even distribution of Virtual Users across the warehouse



Parameters for HammerDB Type-C test

When creating tables for TPC-C, the following are specified:
Number of Warehouses - the number of warehouses. For 2000 " warehouses " the table size is ~250 GB. The number of warehouses should be 10 times greater than the number of " clients " , since the number of simultaneous transactions for one warehouse is limited by the test conditions. In terms of the test:
Virtual Users number of " clients " - the same as threads, sessions for parallel load.
Driver Script - the command file is created automatically.
Rampup Time - time of gradual increase of load.
How many warehouses to create ? The base number of warehouses is 250-500 per CPU. The official TPC-C benchmark has a fixed number of users per warehouse and uses think time so that the workload generated by each user is not intensive. This increases the number of clients and requires a large number of hosts. HammerDB does not use think time by default, and therefore the number of Virtual Users is approximately equal to the number of cores on the DBMS host. With Hyper Threading, the number of Virtual Users can be increased by a third of the number of physical cores.
1 0 warehouses to serve 100 Virtual Users will mean that the workload will spend significantly more time competing for locks and tpm will be lower. 4-5 warehouses per Virtual User will be the minimum value to ensure an even distribution of Virtual Users across the warehouse. For 100 Virtual Users, it is worth creating a minimum of 400-500 warehouses.
For top configurations of DBMS of any manufacturer, maximum performance for tests without time for thinking is achieved in the region of 2000 warehouses and up to 500 sessions (Virtual Users).

Go Utility - TPC

- the utility is written in the go language, works in the command line, which allows you to automate its launch
- implements TPC-C, TPC-H, CHmark tests
- CHmark combines both tests, uses the TPC-C table scheme and the simplified TPC-H scheme
  - was created for databases serving mixed loads: OLTP and OLAP at the same time
  - For PostgreSQL, it is recommended to migrate OLAP workload to physical replicas

```
wget https://raw.githubusercontent.com/pingcap/go-tpc/master/install.sh
chmod +xinstall.sh
./install.sh
cd .g- tpc /bin
./go-tpc tpcc prepare --warehouses 1 -d postgres -U postgres -p postgres -H 127.0.0.1 -P 5432
./go-tpc tpcc run --time 30s -d postgres -U postgres -p postgres -H 127.0.0.1 -P 5432
```


Go-TPC Utility

G0-TPC utility is written in the go language and runs on the command line, which allows you to automate its launch. and this is its advantage. The utility implements TPC-C, TPC-H, CHmark tests . CHmark combines both tests, uses the TPC-C table scheme and a simplified TPC-H scheme. The test was created for databases serving mixed load: OLTP and OLAP at the same time. For PostgreSQL, it is recommended to transfer the OLAP load to physical replicas. The test may be interesting as a use of a more complex load than short-term TPC-C transactions .

Installing the utility:

```
wget https://raw.githubusercontent.com/pingcap/go-tpc/master/install.sh
chmod +xinstall.sh
./install.sh
/ var /lib/ postgresql /. bash_profile has been modified to add go- tpc to PATH
Installed path: / var / lib/ postgresql /.go-tpc /bin/go- tpc
cd .g- tpc /bin
./go-tpc tpcc prepare -d postgres -U postgres -p postgres - D tpcc -H 127.0.0.1 -P 5432
```

By default creates 10 warehouses . With the parameter `-- warehouses N` You can specify the desired number of warehouses. By default, it creates a database named test. Tables for tests should use different databases. If the tables are not needed, it is enough to delete the created database.

To create tables for the TPC-H test , use the command:

```
./go-tpc tpch prepare -d postgres -U postgres -p postgres - D tpch -H 127.0.0.1 -P 5432
```

Running tests:

```
./go-tpc tpcc run -d postgres -U postgres -p postgres -D tpcc -H 127.0.0.1 -P 5432
./go-tpc tpch run -- sf 1 -d postgres -U postgres -p postgres -D tpch -H 127.0.0.1 -P 5432
```

Parameter `-- sf N` sets the scale factor , default is 1. By default, the size of tables is small: in the lineitem table 6 million lines, orders 1.5 million.

`the --count N` parameter you can set the number of iterations. For tpch This is the number of requests.

`-T` parameter can be used to set the number of threads; the default is 1.

`the --time NhNmNs` parameter You can set the duration of the test.

Testing can be interrupted by pressing < ctrl+c > and getting the result:

```
Got signal [interrupt] to exit.
Finished
tpmC : 939.7, tpmTotal : 2089.7, efficiency: 730.7%
```

Practice

- Part 1. Standard pgbench test
- Part 2. Using pgbench with your own script
- Part 3. Using the sysbench utility
- Part 4. Using the HammerDB application
- Part 5. Using the Go-TPC application

Practice

Objective of the practice: to use performance testing utilities.
In practice you :
run te c t pgbench and check how the test result changes if you hold the database horizon ;
check which is faster count(*) or count(1);
how many times the execution time of the EXPLAIN command is reduced when using the timing off option;
sysbench utility reports .
Install the HammerDB program , run the test and see how autoanalysis worsens the DBMS performance.
Install the Go - TPC program and learn how small queries can take several hours to execute (not all DBMS can execute such queries) .

2-1

Memory

RAM

Memory is managed in pages.

- the size of a normal page is 4Kb, set by hardware
- TLB size (Translation Lookaside Buffer (associative translation buffer) 2-4Kb
- miss rate 0.01-1%
- when hitting the TLB, the memory access speed is 1 /2-1 processor cycle
- if the reference is not in the TLB, a slow translation mechanism is used
- misses are processed in 10-100 processor cycles
- the size of a normal memory page is given by the command:

```
astra@tantor :~$ getconf PAGE_SIZE
4096
```


RAM

RAM is one of the main resources used by instance processes. RAM size is not the only parameter that affects the efficiency of memory work. This chapter discusses the features of memory work that affect efficiency.

The MMU (memory management unit) divides the virtual address space (a one-dimensional array of addresses used by the processor) into equal-sized chunks called pages. The address of a memory cell (which the processor processes in one clock cycle) consists of an offset within the page and a page number. Concatenating the physical page number with the offset within the page yields the physical address. The MMU converts virtual page numbers to physical page numbers using the TLB .

The operating system runs on top of the hardware and allocates memory in pages that the hardware uses. The size of a typical memory page is 4 kilobytes (4096 bytes). The page size is given by the command:

```
astra@tantor :~$ getconf PAGE_SIZE
4096
```

Why is the page size 4KB, can it be changed and can it be different ?

All x86 processors currently have a standard page size of 4 KB . The size is chosen empirically and is determined limitations of the mutual arrangement of semiconductor elements on a silicon crystal. For x86 and ARM, it is assumed that 64K pages will be used in the future.

To translate virtual memory into physical memory addresses, the CPU chip has a TLB (Translation Lookaside Buffer , associative translation buffer) . The TLB functions as an array of references to the main memory cache. The amount of memory that the TLB can simultaneously display is determined by the number of TLB " entries " and the size of the " entry " . The TLB size is 2K-4K " entries " . Access to memory referenced in the TLB occurs in 1 or half a clock cycle (dual mode) CPU. If there is no link, then access takes 10-100 CPU cycles (miss penalty) .

If the reference is not in the TLB, a slow translation mechanism is used: hardware or software structures (page table , PT, page tables) of the operating system. The data in these structures is called page table entries (PTE). The procedure is called page walk , it walks PT. PT has a tree structure (radix-tree).

Virtual memory addressing

- the operating system and processes work with a virtual address space
- The MMU (memory management unit) divides the virtual address space into equal-sized areas called pages.
- the translation of a virtual address to a physical address should be as fast as possible
- the operating system must transparently save (swap) the contents of physical memory to external storage and read back (swapping)

Virtual memory addressing

PTE and TLB may contain additional information: page write flag bit (dirty bit), time of last access to the page (accessed bit), which is used to implement the page eviction algorithm (least recently used , LRU), which processes (user) or system (supervisor) can read or write data to the page, whether the page needs to be cached.

Practical (" empirical ") TLB cache miss rate 0.01 -1% (1:100... 10000).

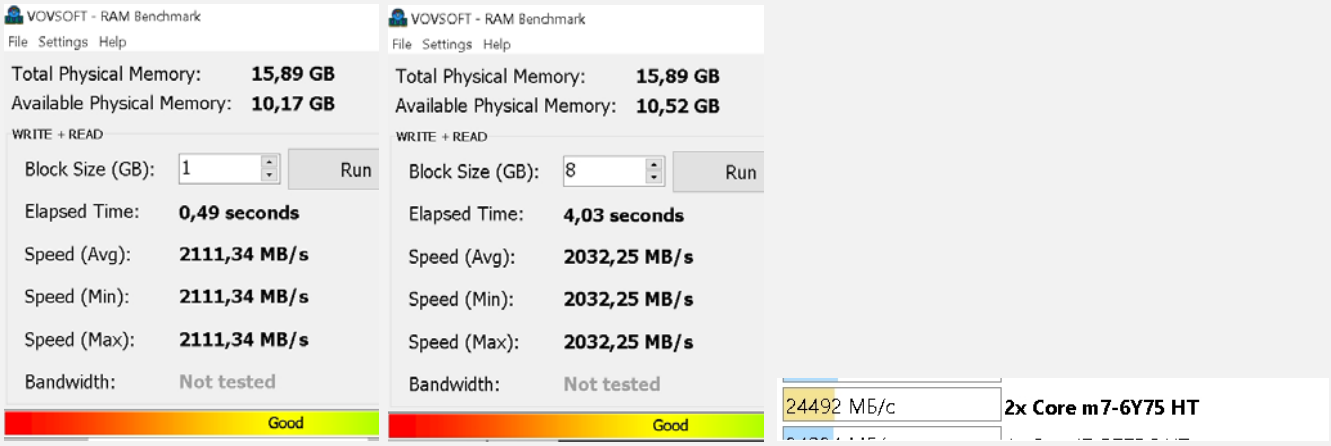
Why ? After all, even 4GB of virtual memory corresponds to a million pages. The TLB size is 4K (and is usually even divided in half into pages with dTLB data and executable code iTLB). It turns out that the difference is a million times, not 10,000 times . It helps that programs access memory in a highly non-uniform way (non-linear distribution): very often to some memory pages, rarely to others. For example, if a program allocates memory by gigabyte (writes, then reads), then the speed will be about 2 gigabytes per second, which is slower than accessing an SSD. At the same time, the speed of access to small volumes will be an order of magnitude higher (20 gigabytes per second).

x86 architecture handles TLB misses at the hardware level, not the operating system (software handlers). When handled by the operating system, the TLB miss handler code is typically 10-100 instructions long (when handled at the hardware level: " clocks "). The CPU can be evicted from the CPU instruction cache and the miss can take much longer to handle than when handled at the hardware level. Software TLBs were found in the MIPS, SPARC, Alpha and PA-RISC architectures. Only on these architectures **could the operating system (linux) use 8Kb pages.**

The translation of a virtual address into a physical address must be as fast as possible. The operating system transparently saves (pushes out) the contents of physical memory to external storage (files or swap partitions) and reads the pages back. This is called swapping. The content is stored in pages, swapping works with normal sized pages. Huge Pages in Linux are not mapped to the page cache, so they are not swapped out , which provides more predictable performance.

Memory page size

- processors support 2 types of pages, normal and huge (Huge Page s)
- On x86-64 architecture, huge pages can be 2MB and 1GB in size.



Memory page size

Modern processors support both regular and huge (Huge Page s) pages.
For processors x86-64 architecture huge pages can be 2MB and 1GB in size.
The processor supports large pages of 2MB if the commands:

```
astra@tantor :~$ lscpu | grep pse
```

or

```
astra@tantor :~$ cat /proc/ cpuinfo | grep pse | uniq
flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse 36
clflush mmx fxsr sse sse2 ht syscall nx ...
```

give a non-empty string, in the given string the word `pse` is present .
This was an example of the output for

```
model name : Intel(R) Core(TM) i3 -8100 CPU @ 3.60GHz
```

The processor supports HugePages 1GB in size, if the team

```
astra@tantor :~$ cat /proc/ cpuinfo | grep pdpe1gb | uniq
flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse 36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc arch_perfmon pebs bts rep_good nopl ...
```

will give a non-empty line, the output line will contain the word `pdpe1gb` .
This was an example of the output for

```
model name : Intel(R) Core(TM) i7 -4700HQ CPU @ 2.40GHz .
```

In the output for `i3 -8100` this word was not there, which means the i3 processor does not support large pages of 1 GB.
For `AMD Ryzen 5 1600X Six-Core Processor`

```
flags: .. pae .. pse36 .. pdpe1gb ..
TLB size: 2560 4K pages
```

The output for AMD contains the line TLB size.
The slide shows the measurement of the write-read speed of a piece of memory of 1GB, 8GB ~ 2GB / s and individual pages ~ 24GB / s on the same host with regular pages, which is an order of magnitude slower. Initial allocation of 1GB or 8GB does not introduce any significant delay. The main role is played by the increase in the TLB miss rate . Access to a page that was recently accessed is an order of magnitude faster than to one whose reference was evicted from the TLB.
https://wiki.debian.org/Hugepages#x86_64

Translation Lookaside Buffer (TLB) Size

- TLB organization depends on the processor (generation, architecture)
- performance when choosing 2M or 1G pages on different processors can vary significantly
- Intel Sky Lake, Coffee Lake, Cascade Lake processors store 1536 references to 4K and 2M pages and 16 references to 1G pages in the L2 TLB
- Intel Sunny Cove family processors have 2048 links, of which up to 1024 links can be used for pages of 2M or 1 G size

Translation Lookaside Buffer (TLB) Size

The choice of page size (4K, 2M, 1G) depends on the processor model. **Performance when choosing 2M or 1G pages on different processors can vary significantly** .

the Intel "Lake" processor family :

```
TLB consists of dedicated L1 TLB for instruction cache (ITLB) and another one for data cache (DTLB) . Additionally there is a unified L2 TLB (STLB) .
DTLB
4K page translations: 64 entries ; 4-way set associative fixed partition
2M page translations: 32 entries; 4-way set associative fixed partition
1G page translations: 4 entries; 4-way set associative fixed partition
STLB
4K+2M page translations: 1536 entries ; 12-way set associative fixed partition
1G page translations: 16 entries ; 4-way set associative fixed partition
```

The characteristics are given without translation, as they consist mainly of terms. When translating highly specialized terms, the meaning is lost.

Intel processor families are given in the manual:

https://cdrdv2-public.intel.com/821613/355308-Optimization-Reference-Manual-050-Changes-Doc.pdf

Meaning of terms:

"partition" - distribution of the number of entries for pages of different sizes.

"fixed" - the number of entries for pages of a certain size is fixed.

"shared" - the operating system will be able to choose how many entries to use for pages of one size and the rest of the space under entries for pages of another size.

Intel "Lake" processor families The L2 TLB can store references to 16 1GB pages .

The specified characteristics define the capabilities of processors, are published by processor manufacturers partially. Not only processor manufacturers avoid direct comparison of their products. This is justified by the fact that to determine the real capabilities of processors, it is necessary to compare the values as a whole, and also by the fact that due to differences in implementation, the characteristics of different manufacturers and even products are only consonant and cannot be directly compared. Example: the cost of the execution plan (cost) are comparable only for one request, for different requests, cost is not comparable. Another example: one processor can have 256 vCPU , the second 8 vCPU , while the real performance of the second processor can be higher. Performance is determined by the balance of processor characteristics. This does not mean that you do not need to know about the characteristics of the processors. When determining the characteristics, you need to find out in detail what is meant by the name of the characteristic and how a change in the characteristic affects the overall performance, since the effect is not always linear.

The characteristics of the processor families are given on the website:
https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

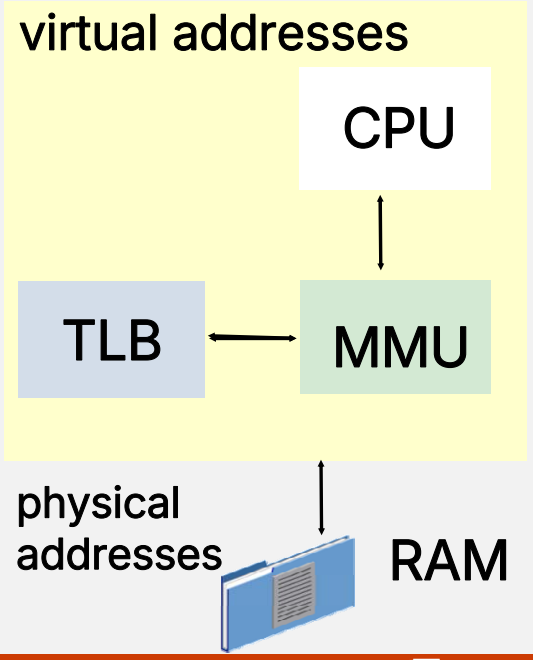
```
DTLB 4 KiB TLB competitively shared (from fixed partitioning)
Single unified TLB for all pages (from 4 KiB+2/4 MiB and separate 1 GiB )
STLB uses dynamic partitioning (from partition fixed partitioning) :
4K pages can use all 2,048 entries
2M pages can use 1,024 entries (8-way sets), shared with 4K pages
1G pages can use 1,024 entries (8-way sets), shared with 4K pages
DTLB
4K page translations: 64 entries ; 4-way set associative competitively shared
2M page translations: 32 entries ; 4-way set associative competitively shared
1G page translations: 8 entries ; 8-way set associative competitively partition
STLB
All pages: 2,048 entire ; 16-way set associative
```

About the degree of associativity (*N* `-way set associative`) it is enough to know: the larger the number (*N* `-ways`), the greater the efficiency of the cache . A 1K **4-way is about as efficient as a 2K 2-way . An 8 -way of 1K size is about as efficient as a 4- way of slightly less than 2K size** .

https://ru.m.wikipedia.org/wiki/Cache_processor

Huge pages (Huge Page s)

- the number of Huge Pages must be set manually in the operating system parameter `vm.nr_hugepages`
- it is possible and necessary to install with a small reserve
- change the number of Huge Pages without rebooting
- consider using with a large amount of physical memory (more than ~128 GB)



Huge Pages

Before enabling Huge Pages (HP) usage, you need to estimate how much HP you plan to use. To get the amount of memory allocated to an operating system process, you can find the PID of the process. In the example , the PID of the postmaster process is taken to see how much virtual memory the process has allocated based on the fact that HugePages are allocated or reserved when the instance is started:

```
astra@tantor :~$ ps - ef | grep postgres
postgres     926 1 0 00:00:02 / usr /lib/ postgresql /15/bin/ postgres
postgres 978 926 0 00:00:00 postgres : 15/main: logger
postgres 991 926 0 00:00:00 postgres : 15/main: checkpointer
```

Next, find the line in the statistics using the command:

```
astra@tantor : ~ $ cat /proc/ 926 /status | grep VmPeak
VmPeak : 265284 kB
```

In this example , `265284kB was output,` which is slightly less than 260MB or 130 2MB pages.

HugePages can use the shared pool and parallel processes. In the example, the shared pool is 128MB. 260MB is much larger than the shared pool. **When allocating HugePages, you can use VmPeak only as a guideline.** You can also use the sum of the values `of shared_buffers + min_dynamic_shared_memory` . To get a more accurate estimate, you will have to stop the instance and start postgres from the command line with the **`-C shared_memory_size_in_huge_pages parameter`** . However, in forks PostgreSQL huge pages can use other memory structures as well.

Example of reserving address space in virtual memory for 300 pages:

```
root@tantor :~# sysctl -w vm.nr_hugepages =300
vm.nr_hugepages = 300
```

To save the new value after restarting the operating system, you can insert the desired number of huge pages at the end of the configuration file:

```
root@tantor :~# echo " vm.nr_hugepages = 300" > > /etc/ sysctl.conf
```

Apply the changes that appeared in the file:

```
root@tantor :~# sysctl -p
vm.nr_hugepages = 300
```

How big should HP make its page reserve? This is determined by the `CommitMemory formula` on the slide " Setting overcommit and swap values ", which will be a little further . In the absence of swap, the reserve should be minimal.

Using Huge Pages

Usage statistics :

- `cat /proc/ meminfo | grep Huge`
  - `HugePages_Rsvd` : in use
  - `HugePages_Free` : can be used
  - `HugePages_Total` : available

```
root@tantor :~# cat /proc/ meminfo | grep Huge
AnonHugePages : 0 kB
ShmemHugePages : 0 kB
FileHugePages : 0 kB
HugePages_Total : 300
HugePages_Free : 281
HugePages_Rsvd : 72
HugePages_Surp : 0
Hugepagesize : 2048 kB
Hugetlb : 614400 kB
```


Using Huge Pages

The use of HP in PostgreSQL is determined by the `huge_pages` parameter . By default, it is set to `try` . If HP are available, they are allocated; if allocation fails, regular pages are allocated. If you set the parameter to "on" , then if it is not possible to allocate an HP page, regular pages will not be allocated and the instance may not start if the shortage occurs during instance startup. You should not use the " `on` " value unless you have ensured that there are enough HP pages .

Let's check that large pages are available to processes:

```
root@tantor :~# cat /proc/ meminfo | grep Huge
AnonHugePages : 0 kB
ShmemHugePages : 0 kB
FileHugePages : 0 kB
HugePages_Total : 300
HugePages_Free : 300
HugePages_Rsvd : 0
HugePages_Surp : 0
Hugepagesize : 2048 kB
Hugetlb : 614400 kB
```

Let's restart the instance:

```
root@tantor :~# systemctl restart postgresql
```

Let's check that the pages have been selected:

```
root@tantor :~# cat /proc/ meminfo | grep Huge
AnonHugePages : 0 kB
ShmemHugePages : 0 kB
FileHugePages : 0 kB
HugePages_Total : 300
HugePages_Free : 281
HugePages_Rsvd : 72
HugePages_Surp : 0
Hugepagesize : 2048 kB
Hugetlb : 614400 kB
```

There are 72 pages allocated in the virtual address space, which is 144 MB. Of these, the buffer pool (`shared_buffers = 128 MB`) takes up 128 MB. It is allocated but not used, since these pages were not accessed (the buffer pool is not full), so the free -- mega command will show that after starting the instance, the free memory has decreased by only 8 MB:

```
root@tantor :~# free --mega
total used free shared buff/cache available
Mem : 2074 1378 371 12 505 696
Mem : 2074 1386 363 14 506 688
```

Using Huge Pages Instance

- Huge Pages are used for the shared buffer pool and Dynamic Shared Memory (DSA) memory , which is used by parallel processes
- memory is allocated in the manner specified by the `shared_memory_type parameter` .
- in linux Large page memory allocation is only supported by the `mmap method` (' anonymous ' pages) .
- Huge Pages are not pushed to swap.
- If this amount of memory is not enough, then parallel processes will additionally allocate (and then free) memory in 4Kb pages and in the manner specified in the `dynamic_shared_memory_type parameter` :

```
postgres =# select name, setting, enumvals from pg_settings where name like '% memory_type ';
name | setting | enumvals
----------------------------+---------+------------------
 dynamic_shared_memory_type | posix | { posix, sysv, mmap }
 shared_memory_type | mmap | { sysv ,mmap }
```


Using Huge Pages Instance

The largest size in an instance is the buffer pool. In busy instances, parallelization of command execution is usually used. HP allocation is a relatively long operation, so **Huge Pages are used only for the shared buffer pool and memory used by parallel processes (parallel workers)** . The size of the buffer pool is specified by the `shared_buffers parameter` . Memory is allocated in the manner specified by the `shared_memory_type parameter` . In Linux Memory allocation by large pages is supported by the `mmap method` (" anonymous " pages) . Large pages are not pushed out to swap.

HP can also be used by parallel processes. The amount of reserved HP is specified by the **`min_dynamic_shared_memory`** parameter . If this amount of memory is not enough, parallel processes will additionally allocate (and then free) memory in the manner specified by the `dynamic_shared_memory_type parameter` :

```
postgres =# select name, setting, enumvals from pg_settings
where name like '% memory_type ';
name | setting | enumvals
----------------------------+---------+------------------
 dynamic_shared_memory_type | posix | { posix , sysv , mmap }
 shared_memory_type | mmap | { sysv , mmap }
```

The default values are optimal and should not be changed.

`posix` - memory is allocated in **regular 4K pages using the** `shm_open` system call . **You should not use the mmap value for the parameter `dynamic_shared_memory_type`** , since when using mmap the directory on disk `PGDATA/ pg_dynshmem` is used (created if not created) and the files in it will be used to display shared memory between parallel processes if they do not have enough memory reserved by **`min_dynamic_shared_memory`** . When **`min_dynamic_shared_memory is exhausted`** Parallel processes allocate memory in regular pages. **For the `shared_memory_type parameter`, you need to use the value `mmap`** .

Whether an instance can use Huge Pages is controlled by the `huge_pages` parameter . The default value is **`try`** . This means that if the instance succeeds in allocating Huge Pages when it starts , then they will be allocated and used. If it fails, then they will not be used at all.

https://www.cybertec-postgresql.com/en/huge-pages-postgresql/

If the PostgreSQL configuration parameter `huge_pages = on` , and `HugePages pages` if there is not enough when starting the instance, the instance will not start.

```
root@tantor :~# sysctl -w vm.nr_hugepages =300
vm.nr_hugepages = 300
root@tantor :~# systemctl restart postgresql
postgres@tantor :~$ psql -p 5435 -c "show shared_buffers ;"
 shared_buffers
----------------
 128MB
(1 row)
root@tantor :~# cat /proc/ meminfo | grep HugeP
HugePages_Total : 300
HugePages_Free : 281
HugePages_Rsvd : 72
HugePages_Surp : 0
```

72 were allocated two megabytes large pages. Of these, 64 pages are directly under the shared pool. The `HugePages_Rsvd value may not increase, but then the` HP allocation will indicate a decrease in value `HugePages_Free` .

```
postgres@tantor :~$ psql -p 5435 -c "alter system set min_dynamic_shared_memory =' 200MB
';"
ALTER SYSTEM
postgres@tantor :~$ sudo systemctl restart postgresql
postgres@tantor :~$ cat /proc/ meminfo | grep HugeP
HugePages_Total : 300
HugePages_Free : 280
HugePages_Rsvd : 1 71
HugePages_Surp : 0
```

200MB has been allocated for use by parallel processes.

List of processes using HP:

```
root@tantor :~# grep " KernelPageSize : 2048 kB " /proc/[[:digit:]]*/ smaps |
awk {'print $1'} | cut -d "/" -f3 | sort | uniq
62133
...
```

Size and type of HP used by the process :

```
root@tantor :~# cat /proc/ 62133 / smaps_rollup | grep tlb
Shared_Hugetlb : 30720 kB
Private_Hugetlb : 10240 kB
```

postgres process from Astralinux PostgreSQL.

It is possible to check on a stopped instance how many HP pages with the current configuration parameter settings the instance can request at startup:

```
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
91
```

The number 91 was given, 72 pages were allocated at launch.

For `dynamic_shared_memory_type = posix` pages are not swapped , but memory mappings are created in the form of directory files mounted on a `tmpfs file system` :

```
postgres@tantor :~ $ ls -al /dev/ shm
total 1136
- rw ------- 1 postgres postgres 1048576 PostgreSQL.1357156412
- rw ------- 1 postgres postgres 108576 PostgreSQL.2756012128
- rwx ------ 1 astra    astra 32 sem.user_1000_label_none
```

In version 17, a descriptive (readable) parameter `huge_pages_status appeared` , which shows whether HP is used by the instance.

Transparent huge pages

- slow down the work of DBMS processes
- not highlighted if
  - `AnonHugePages : 0 kB`
- check if use is enabled:

```
root@tantor :~# cat /sys/kernel/mm/ transparent_hugepage
/enabled
always [ madvise ] never
root@tantor :~# cat /sys/kernel/mm/ transparent_hugepage
/defrag
always defer defer+madvise [ madvise ] never
```


Transparent Huge Pages

Beyond Huge Pages in Linux available for use transparent huge pages (Transparent Huge Pages , THP), which appeared in the Linux kernel 2.6.38 in 2012.

THP slows down the DBMS and THP is not worth using at this time .

The use of THP is indicated by the line `AnonHugePages` :

```
root@tantor :~# cat /proc/ meminfo
AnonHugePages : 0 kB
ShmemHugePages : 0 kB
FileHugePages : 0 kB
...
```

check if THP is disabled using the command:

```
root@tantor :~# cat /sys/kernel/mm/ transparent_hugepage /enabled
always [ madvise ] never
```

The current value is highlighted in square brackets . The value is **`never`** disables the use of THP. The value **`madvise`** allows processes to request the use of THP via the `madvise ()` system call . PostgreSQL does not use such a system call, so it is sufficient to check that the parameter value is not set to `always` .

With THP , " defragmentation " of THP pages is used . You can check the defragmentation mode with the command:

```
root@tantor :~# cat /sys/kernel/mm/ transparent_hugepage /defrag
always defer defer+madvise [ madvise ] never
```

Meaning of `always` or `defer` leads to synchronous defragmentation (direct compaction), blocking the work of processes. For the most part, the lack of optimization of defragmentation and the slow movement of huge ("huge") memory ranges are the reason for the slowdown of applications when using THP.

You can change the values using the commands:

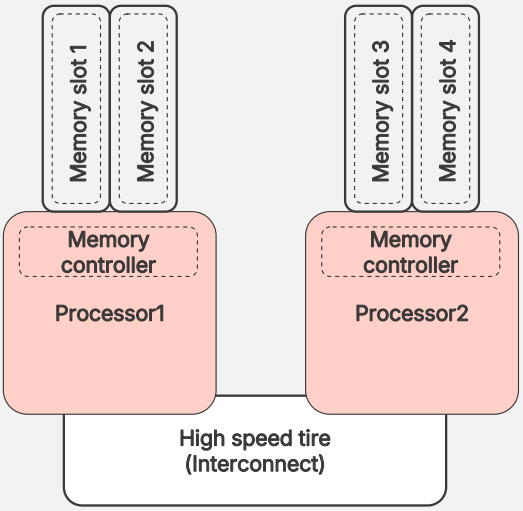
```
echo never > /sys/kernel/mm/ transparent_hugepage /enabled
echo never > /sys/kernel/mm/ transparent_hugepage /defrag
```

To save the disabled state after reboot, and also during the Linux startup process THP was not used, you can specify this in the Linux boot loader parameters .

Non-Uniform Memory Access (NUMA)

Non-Uniform Memory Access
(non-uniform memory access) :

- Each processor has local physical memory and accesses it in the normal way through its memory controller. In addition, each processor has access to the local memory of other processors through a slower I/O bus
- PostgreSQL is optimized to work with uniform access to physical memory. and is not optimized for NUMA



Non-Uniform Memory Access (NUMA)

NUMA (Non-Uniform Memory Access (non-uniform memory access) is a hardware architecture in which each processor has local physical memory and accesses it in the normal way through its memory controller. In addition, each processor has access to the local memory of other processors through a slower I/O bus.

UMA (Uniform Memory Access (uniform memory access) is used in SMP (symmetric multiprocessing). Processors use physical memory simultaneously. Access speed does not depend on which processor is accessing the memory, nor on which memory chip contains the required data. Each processor can use its local caches .

PostgreSQL is optimized to work with Uniform Memory Access (UMA) and is not optimized for working with NUMA , at least because the buffer cache is uniform.

When using hardware that has the ability to use NUMA , in the firmware (BIOS) of such equipment there may be a parameter called "Node Interleaving" or "Memory Nodes" (alternation of memory nodes) . This parameter must be enabled, then the presence of NUMA will not be represented by the operating system and the memory allocated by the operating system will be automatically distributed between memory nodes .

If this is not done, then by default shared memory structures (buffer cache, log cache, and others) will be allocated, if possible, in the local memory of one processor (on which the postgres process is running) and all other processors will have slow access (via a slower I/O bus) to the shared memory structures. This will be worse than if the physical memory in which the shared memory structures are allocated were evenly distributed among the local memories of all processors (Memory Nodes) .

Each processor has its own memory access interface and its own cache contents (L1, L2, L3) and buffers (TLB). If there are several processors, it may be optimal, so that processes are executed on the same processor whenever possible, rather than migrating between processors. Migration between cores of the same processor does not matter. The task of binding processes to processors is called CPU affinity . The relevance of the task increases if the number of active processes is much greater than the number of processor cores.

2-2

Out of memory

Out of Memory (OOM)

- processes are sent a `SIGKILL signal`
- the best candidate is considered to be the process that
  - will free up maximum memory
  - is the least important for the system
  - Ideally, one process should stop
- The score for each process is pre-calculated and stored in `/proc/PID/oom_score`
- Process termination messages are written to the operating system log

Out of Memory (OOM)

The operating system tries to use all the RAM so that it does not " sit idle " . Memory that is not used by processes is used by the page cache . In the page cache, some pages may be " dirty " and such memory is not released quickly, since the pages need to be written to the disk. If processes request memory, then part of the page cache is released to allocate memory. To avoid a situation where the allocation takes a long time and slows down the work of all processes, memory can be released in Linux by forcibly stopping processes that consume a lot of memory. This is done by the oom killer process . It stops processes by sending them the `SIGKILL signal` .

Oom killer algorithm , by which the process or processes are selected to be stopped is considered " heuristic " . The algorithm takes into account many parameters and can change from version to version of Linux . The best candidate is considered to be the process that will free up the maximum amount of memory, and is also the least important for the system. It is also desirable to stop a smaller number of processes, so processes that have allocated a lot of memory are selected. **The score for each process can be viewed in `/proc/PID/ oom_score` . The higher the number, the more likely it is that this process will be stopped.**

oom kill message in the operating system log :

```
Out of memory: Kill process 58302 (postmaster) score 837 or sacrifice child
Killed process 58302 (postmaster) total-vm:72328760kB, anon - rss :55470760kB,
file - rss :4753180kB
```

" `total-vm` " - the size of **virtual** memory used by the process. **" - `rss` " - the portion of virtual memory that is mapped to RAM (allocated and used)** .

" **`anon - rss`** " - memory allocated by the process in physical pages of RAM and has no mapping to files and devices (no name, anonymous).

For example, if a process executes `the malloc () system call` allocating 1GB and writes or reads from the allocated 1GB, then " **`anon - rss`** " and " `total-vm` " will increase by 1GB. If the process does not write or read, then " `total-vm` " will increase by 1GB, and " **`anon - rss`** " will not change.

" **`file - rss`** " - memory allocated in physical pages of RAM and mapped to files or devices. " **`file - rss`** " will be high if you open a large file for reading and read its contents.

https://www.baeldung.com/linux/processes-memory-short

Resident Set Size (RSS)

- RSS (resident size) - the amount of memory allocated to the process and located in physical memory
- PSS (proportional size) provides a complete picture of the distribution of physical memory between processes and shared libraries
- attractiveness of the process for `oom kill` :
  - `/proc/'$PID'/ oom_score`

Resident Set Size (RSS)

" - **rss** " means , which is displayed in the OOM kill message and linux statistics .

RSS (Resident Set Size) - the amount of memory allocated to the process by the operating system and currently in RAM (physical memory). Does not include pages of memory in the swap space (swap size) This takes into account the physical memory allocated for shared libraries that the process uses.

RSS inflates memory usage .

PSS (Proportional Set Size) provides a complete picture of the distribution of physical memory between processes and shared libraries.

To determine PSS, you can use the `smem utility` or the command:

```
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
```

Example: if a library uses 100Kb, and the library is used by 10 processes, then each process's memory RSS will additionally take into account 100Kb. PSS in this example will add 10Kb to each process's memory, i.e. it will divide the memory used by the library proportionally between the processes using the library. Of course, one process could force the library to actively use memory for its own service, but for the purpose of estimating the memory used, this does not matter.

Example:

P.S.S. Without HP:

```
66573 - 85 -> / usr /lib/ postgresql /15/bin/postgres (PID 58302)
5799 - 669 -> postgres: 15/main: walwriter (PID 58308)
3726 - 669 -> postgres: 15/main: autovacuum launcher (PID 58309)
3212 - 668 -> postgres: 15/main: logical replication launcher (PID 58310)
2913 - 668 -> postgres: 15/main: background writer (PID 58306)
1716 - 668 -> postgres: 15/main: checkpointer (PID 58305)
1507 - 668 -> postgres: 15/main: logger (PID 58304)
```

Sum: 66573 (for the postgres process) + 18873 (first column of the remaining rows) = 85446

```
root@tantor :~# free
total used free shared buff/cache available
Mem : 2025796 647160 846888 47436 611812 1378636
```

P.S.S. with HP:

```
7583 - 74 -> / usr /lib/ postgresql /15/bin/postgres (PID 58353)
2966 - 668 -> postgres: 15/main: autovacuum launcher (PID 58359)
2724 - 668 -> postgres: 15/main: logical replication launcher (PID 58360)
1528 - 668 -> postgres: 15/main: checkpointer (PID 58355)
1508 - 668 -> postgres: 15/main: logger (PID 58354)
1493 - 668 -> postgres: 15/main: walwriter (PID 58358)
1469 - 668 -> postgres: 15/main: background writer (PID 58356)
```

Sum 7583 (for the postgres process) + 11688 = 19271

```
root@tantor :~# free
total used free shared buff/cache available
Mem : 2025796 1022900 471092 12624 577048 1002896
```

Output of the **`free command`** by default in kilobytes. After increasing the number of huge pages, the command output shows the values of free and available decreases and the used value increases .

oom_score the postgres process has decreased slightly due to the fact that significantly less memory is now taken into account: for PSS it was `66573` , now it is `7583` . For RSS there is a similar reduction.

When using HP size The memory of the instance processes, which PSS shows , has decreased several times (was 85446 , became 19271). The amount of memory with which the instance works has remained approximately the same.

It is difficult to estimate how much the performance will increase based on the indicators. It is even more difficult to determine the mechanism of the increase. On small memory volumes, the fact that HP are not forced out of memory, which ensures more predictable operation. On large amounts of RAM Fast access will be provided to pages referenced in the TLB.

Example: number of unemployed operating system (under the Linux kernel pages) and libraries) entries in the TLB 1,024 . For normal pages of 4K size, fast access (via TLB) will be to 1,000 * 4K = 4MB of memory. For HP of 2MB, fast access will be up to 2GB of memory.

In any case, using HP with DBMS will not lead to performance degradation, unlike THP.

https://www.percona.com/blog/why-linux-hugepages-are-super-important-for-database-servers-a-case-with-postgresql/

oom_score_adj parameter

- The value is set after startup for each process:
  › echo -900 > /proc/ 58253 / oom_score_adj
- changes /proc/PID/ oom_score for this process
- value -1000 for postgres process installed in the Tantor DBMS service file tantor-se-server-16.service
- for other instance processes the default value does not change and is set to zero

oom_score_adj parameter

In the example given earlier, oom_score was output instance processes. The postgres process oom_score was 74 or 85 , the other processes had 668-669. How was the scoring lowered ? Because the oom_score_adj value was changed and set to -900

```
root@tantor :~# cat /proc/ 58253 / oom_score_adj
-900
```

The remaining instance processes use the default value of zero:

```
root@tantor :~# cat /proc/ 58359 / oom_score_adj
0
```

If you change the value of the postgres process on the fly :

```
root@tantor :~# echo 0 > /proc/ 58253 / oom_score_adj
```

then the oom_score value for the postmaster will be approximately the same as for the other processes in the instance:

```
root@tantor :~# cat /proc/ 58253 / oom_score
673
```

postgres process spawns other instance processes and the oom_score_adj value Not inherited by child processes.

Stopping any process with SIGKILL signal causes the instance to be restarted by the postgres process. However, stopping the postgres process with the SIGKILL signal worse. Therefore, the postgres process is set to oom_score_adj . The Tantor DBMS sets the value to " -1000 " and oom_score will be equal to zero. This is done by a directive in the Tantor DBMS service file tantor-se-server-16.service :

```
# Prevents OOM kill from killing the postmaster process
OOMScoreAdjust = -1000
# ... but allows you to kill processes that postmaster spawns
Environment = PG_OOM_ADJUST_FILE =/proc/self/ oom_score_adj
Environment = PG_OOM_ADJUST_VALUE = 0
```

You can change oom_score_adj replacing the remaining processes of the instance in the service file 0 for example to -300. **Change** oom_score_adj **only makes sense if the host has multiple instances of processes running, or processes that consume a lot of memory** . The options set the environment variables before starting the postgres process .

https://docs.tantorlabs.ru/tdb/ru/15_6/se/kernel-resources.html#LINUX-MEMORY-OVERCOMMIT

vm.overcommit_memory parameter

- has three meanings:
  - 0 - default value. Allocates as much memory as the process requests.
  - 1 - not used with DBMS.
  - 2 - memory allocation failure if the total amount of allocated memory exceeds the size of the swap space plus the amount of physical memory multiplied by the percentage value of the vm.overcommit_ratio parameter (default 50%) or absolute value specified by vm.overcommit_kbytes
- **if the swap partition is disabled and vm.overcommit_ratio <100 , then you shouldn't set the value to 2, it should be 0**
- vm.overcommit_ratio parameter does not matter when the vm.overcommit_memory parameter is set to 0 or 1

vm.overcommit_memory parameter

Let's look at the parameters that directly affect the triggering of OOM kill or the issuance of a message to the process about insufficient memory. The formula that links the parameters is shown on the next slide. Changing one parameter without taking into account the values of the others can lead to the triggering of OOM kill or a refusal to allocate memory even if there is enough memory.

vm parameter . overcommit_memory can be set to one of three values:

•0 - default value. Allocates as much memory as the process requests. Only those pages that will be used by the process are swapped / res erved . The mmap memory allocation system call (you can see the description of the call with the man mmap command) without the MAP_NORESERVE option , it does not check how much memory is available and tells the process that the memory is allocated. If you try to use the allocated memory, there is a chance that an oom -kill will be triggered.

•1 - checking how much memory is available is not performed. This mode is usually used for scientific tasks working with large arrays. It is not used with DBMS.

•2 - memory allocation failure if the total amount of allocated memory exceeds the size of the swap space plus the amount of physical memory multiplied by the percentage value of the vm.overcommit_ratio parameter (**default 50%, value chosen based on the fact that swap partition is equal to 50% of physical memory**) or absolute value specified by vm.overcommit_kbytes parameter .

vm.overcommit_ratio parameter does not matter when the vm.overcommit_memory. parameter is set to 0 or 1.

The choice for the host servicing the DBMS comes down to the values 0 and 2.

The current values of the memory volume that can be selected can be viewed using the command:

cat /proc/ meminfo | grep Commit

If the swap partition is disabled And **vm.overcommit_ratio <100 ,** then you should not set the value to 2.

Setting the value to 2 will cause processes to receive a memory allocation error instead of triggering oom kill .

https://www.kernel.org/doc/Documentation/vm/overcommit-accounting

Setting overcommit and swap values

- when servicing in Linux only DBMS it is worth setting the values **`vm.overcommit_ratio=100 ,vm.overcommit_memory=2`**
  - with such values oom triggering kill is unlikely and will use all physical memory
  - it will be possible to disable the swap partition without increasing the probability of oom kill triggering
  - if you set the value `vm.overcommit_ratio > 100 the probability of` oom triggering increases kill
- The total size of virtual memory that can be allocated is determined by the formula:
  `CommitLimit =( total_RAM - total_huge_TLB )* vm.overcommit_ratio /100+total_swap`
- active use of the swap partition takes up I/O bus bandwidth

Setting overcommit and swap values

The total amount of virtual memory that can be allocated. is determined by the formula:
`CommitLimit =( total_RAM-total_huge_TLB )* vm.overcommit_ratio /100+total_swap`
This formula relates the size of the swap partitions (files) and memory that can be allocated to processes.

The use of swap is undesirable because it slows down and worsens the responsiveness of all processes, since the I/O bus is used. When using NVMe as a swap partition , the I/O speed is comparable to the access speed of physical memory and using swap may make sense. The disadvantage may be that PCIe lines will be used by the NVMe device. If the PCIe bandwidth is a bottleneck, then access to the disks on which the database cluster files are located will slow down and in this case, using swap is undesirable.

What is the advantage of having even a small swap partition ? Processes request memory with rounding and some pages of allocated memory may not be used until the memory is freed. Such pages can be pushed out to swap without affecting performance. Without swap, such pages are in physical memory and physical memory is " underused " , but could be used (during normal operation for page cache).

What values should be set in the absence of swapping and at the same time reduce the probability of OOM-kill triggering? You need to set `vm.overcommit_ratio = 100 and vm . overcommit_memory = 2` . With such values, according to the formula `CommitLimit = total_RAM` . With such values, the oom- kill triggering kill is unlikely and all physical memory will be used. If set the value of `vm.overcommit_ratio` greater than `100` , then the probability of triggering oom kill increases .

If oom kill is triggered and sends a SIGKILL signal to the PostgreSQL instance process , the postmaster process stops all instance processes and starts them again. This has the effect of restarting the instance. If the postmaster fails to restart the processes, the postmaster stops and the instance becomes unavailable. It is also possible that when restarting the processes, one of the background processes (checkpointer) will hang . In this case, the instance will have to be stopped with `the pg_ctl stop -m immediate command` . The `pg_stl stop` command in this case it will hang , and `systemctl stop` will not stop the processes.

vm.swappiness parameter

- by default the value is 60
- You can change the value without restarting the operating system
- the optimal value is about 10
- value 0 should not be used
- affects which parts of memory will be candidates for displacement to swap:
  - `anon _prio = swappiness;`
  - `file _prio = 200 - anon_prio;`
- linux log message about stopping the process :

```
Out of memory: Kill process 58302 (postmaster) score 837 or sacrifice child
Killed process 58302 (postmaster) total-vm:72328760kB, anon -rss:55470760kB, file -
rss:4753180kB
```


vm.swappiness parameter

`vm.swappiness` parameter affects the likelihood of using swap if it is used.
The default value is 60:

```
root@tantor :~# sysctl -a | grep swap
vm.swappiness = 60
```

For a host with physical memory greater than 8-16 GB, the value 60 is not optimal. The optimal value is about 10 . It is worth changing the value in small increments, monitoring the result (whether any pages will be pushed out to the swap partition; during normal operation, they should not be pushed out). You can change the value without restarting the operating system by editing the file `/etc/ sysctl.conf` and applying the changes with `the sysctl -p command` .

```
/*
* With swappiness at 100, anonymous and file have the same priority.
* This scanning priority is essentially the inverse of IO cost.
*/
anon _prio = swappiness ;
file _prio = 200 - anon_prio ;
```

The value 0 tells the linux kernel avoid page evictions as much as possible. This value on modern versions of linux **not worth** using, as OOM killer can work even if there is free swap space. OOM is usually easier terminate the process rather than release memory from file cache. Example of a linux log message about stopping the process :

```
Out of memory: Kill process 58302 (postmaster) score 837 or sacrifice child
Killed process 58302 (postmaster) total-vm:72328760kB, anon -rss:55470760kB,
file -rss:4753180kB
```

Swapping is optional and can be disabled. By using swapping , the operating system can run programs that require more memory than is physically available. It also helps prevent the system from crashing if it runs out of RAM. Performance is also improved by increasing the file cache.
https://eklitzke.org/swappiness

Memory Page Deduplication (KSM)

Kernel Same-page Merging (memory page deduplication):

- can merge local memory pages anon- rss different processes
- does not merge file- rss and shared memory
- used in virtualization, not with DBMS
- memory scans for identical pages linux service

check that KSM is disabled using the command:

```
cat /sys/kernel/mm/ ksm /run
0
```


Memory Page Deduplication (KSM)

KSM (kernel same-page merging , deduplication of memory pages) allows the Linux kernel since version 2.6.32 to merge identical memory pages `anon- rss` (but not `file- rss`) used by different processes into one page for sharing. Shared memory is not deduplicated . Identical pages of local memory of processes are merged into one and marked as subject to copy on modification (writing to the page), so that the pages are shared when modified by one of the processes.

It has the greatest effect when running virtual machines from similar images.

KSM parameters and statistics are located in files in the `/sys/kernel/mm/ ksm directory` :

```
ls /sys/kernel/mm/ ksm
full_scans        merge_across_nodes  pages_to_scan   sleep_millisecs     use_zero_pages
general_profit    pages_scanned       pages_unshared  stable_node_chains
ksm_zero_pages    pages_shared        pages_volatile  stable_node_chains_prune_millisecs
max_page_sharing  pages_sharing run stable_node_dups
```

To work effectively, it requires the ksm service , which scans memory for duplicate pages.

Not used with DBMS.

Disabled by default. You can check that it is disabled with the command:

```
cat /sys/kernel/mm/ ksm /run
0
```

Zero - disabled.

Check that ksm if it didn't work, you can use the command:

```
cat /proc/ vmstat | grep ksm
ksm_swpin_copy 0
cow_ksm 0
```

Zero means there was no activity.

https://docs.kernel.org/admin-guide/mm/ksm.html

Local memory allocation by instance processes

- PostgreSQL implements memory management logic through " memory contexts "
- palloc () is used , which allocates memory in a memory area called MemoryContext , which is also called the "Arena"
- memory contexts form a hierarchy
- TopMemoryContext - the root of the server process memory context hierarchy
- **when a memory allocation request is made, the memory size allocated is the nearest power of 2 to the larger side of the requested size**
- if there is not enough free memory in the context, then memory is added to it **in double the size of the initial one**

Allocating memory to instance processes

To reduce the likelihood of memory leaks, PostgreSQL implements memory management logic through " memory contexts ". The program code does not call the operating system every time by malloc (). When memory needs to be allocated, it uses the palloc () call , which allocates memory in a memory area called MemoryContext , which is also called the "Arena". MemoryContext is notable for the fact that it can completely free the memory allocated for it and there is no need to keep track of multiple calls to allocate memory. The main thing is that the memory is allocated in a context with a suitable lifetime. After the context's lifetime expires, there is no danger of leakage (non-release) of memory. For example, at the beginning of a transaction, TopTransactionContext is allocated , the memory of which will be freed upon completion of the transaction. The amount of memory allocated to the context can be significant. The second aspect of contexts is to minimize the number of malloc () calls and allocate memory in advance in large chunks with rounding by powers of two .

Memory contexts form a hierarchy. TopMemoryContext is the root of the server process's memory context hierarchy. All other memory contexts have a parent context. When the program code deletes a memory context (completion of a transaction, closing a cursor, portal), all child contexts are deleted.

Memory can be allocated using Slab logic (slab.c). This is the implementation of MemoryContext , when memory is allocated **in equal sizes (chunks)** . The chunk size is specified when creating a context of this type. **Memory chunks of different sizes are called blocks , not chunks** . The AllocSet (aset.c) and Generation (generation.c , blocks with a similar lifetime) logic are used more often . The peculiarity of the memory management logic of these contexts is that **when requesting memory allocation, the memory size allocated is the closest power of 2 up from the requested size** ("round request sizes up to the next power of 2"). For example, if 600 KB is requested, then 1 MB is allocated. If exactly 1 MB is requested, then 1 MB is allocated.

If there is not enough free memory in the context, then memory is added to it (by calling malloc ()) **in double the size of the initial** . The block can be limited to 1 GB or 2 GB (when using `enable-large-allocations`), but not in all cases. The part of memory allocated with reserve may not be used (read and written), but it will be taken into account in the operating system as allocated in the virtual address space. It is better not to bring the operating system to a lack of physical memory: linux page cache must be large (comparable in size to the buffer cache). Linux Page Cache uses almost all unused physical memory and serves as a reserve for servicing memory allocation requests.

Memory allocation in generation.c :

```
/*
* The first such block has size initBlockSize , and we double the
* space in each succeeding block , but not more than maxBlockSize .
*/
blksize = set-> nextBlockSize ;
set-> nextBlockSize << = 1;
if (set-> nextBlockSize > set-> maxBlockSize )
set-> nextBlockSize = set-> maxBlockSize ;

/* we'll need a block hdr too, so add that to the required size */
required_size += Generation_BLOCKHDRSZ ;

/* round the size up to the next power of 2 */
if ( blksize < required_size )
 blksize = pg_nextpower2_size_t( required_size );

block = ( GenerationBlock *) malloc ( blksize );

if (block == NULL)
return NULL;

context-> mem_allocated += blksize ;
```

In practice, this chapter will include an example where, when loading a 1GB file, the server process allocates **6GB of virtual memory when reading a 1GB dump file** :

```
Killed process 12518 ( postgres ) total-vm: 6523848kB , anon-rss:3151300kB
```

An example of analysis of memory allocation and deallocation by contexts:

https://dev.to/yugabyte/postgres-memory-allocation-and-os-memory-allocation-30f1

"operating system reporting 70M and the PostgreSQL level memory dump saying it released memory down to approximately 1M!

What we see is 'Arena 0', which is roughly put the administration of memory allocations of malloc () for this process, which has allocated from 'system' 63832064 bytes (60.9M), while actually in use (by PostgreSQL) is 917696 bytes (1M). What malloc () tries to do, is keep memory allocated to prevent having to deallocate and allocate over and over ."

Once memory has been allocated for a context, it is not returned to the operating system. The memory will be returned during the lifetime of the server process, when the context's life cycle ends and the context is freed.

In the report https://pgconf.ru/media/2024/10/21/12/613/ Alexandrov . pdf

`CachedPlan` contexts is considered for 5 private plans and one common one, each 32MB in size each in the parent context `CacheMemoryContext` . Memory allocated for the context is 192 MB (total memory of the server process at the time of memory release in the context USS 188.7M , PSS 189.5 M, RSS 195.0M) was not returned to the operating system .

ERROR: invalid memory alloc request size

- Error reason: attempt to allocate a block of memory exceeding the 2GB or 1GB-1 limit set by the `MaxAllocSize macro`
- in text functions like lpad , rereat checks are inserted that return an error: requested length too large
- if the memory size after the error text is more than 2GB, this means that the memory size that should be allocated is calculated incorrectly and may indicate the presence of corruption of records in data blocks
  - › also the calculation may be performed incorrectly due to errors in the extension libraries loaded into the process memory
- Example of commands that lead to errors:

```
postgres =# create table a as select repeat('a', 1024 * 1024 * 1024 - 5);
ERROR: invalid memory alloc request size 1073741887
postgres =# select repeat('x', 1024 * 1024 * 1024);
ERROR: requested length too large
```


Allocating memory to instance processes

The program code of the instance process does not make calls to the operating system every time it calls malloc (). If the context allocates memory of a uniform size, it is called a chunk , if of an arbitrary size, it is called a block . If there is not enough memory in the context, a call to malloc () is sent to the operating system . Before executing the call, a check is made whether the requested size will exceed the limit:

```
#define AllocSizeIsValid (size) ((Size) (size) <= MaxAllocSize )
```

which is set by the macro:

```
#define MaxAllocSize ((Size) 0x3fffffff) /* 1 gigabyte - 1 */
```

The error with the text "invalid memory alloc request size" is caused by a check for exceeding this limitation.

Similar errors:

array size exceeds the maximum allowed

Text functions like lpad (..) , repeat(..) have checks inserted that produce a less scary error:

requested length too large.

Dynamic Shared Memory (DSA) memory allocation also has checks. The error text when the limits are exceeded is "invalid DSA memory alloc request size" .

If the memory size after the error text is more than 2GB, this means that the memory size that should be allocated is calculated incorrectly and may indicate that there is corruption of records in the blocks. data . Also, the calculation may be performed incorrectly due to errors in the extension libraries loaded into the process memory.

Contexts can allocate blocks and chunks larger than 1GB, but then an operating system error will be issued - either out of memory , or Cannot allocate memory , or the process will be stopped by oom kill .

The total size of the context can exceed 1 GB, the allocation of a block or chunk larger than 1 GB-1 is limited. Example of description of context sizes:

```
CacheMemoryContext : 59375840 total in 21 blocks; 8112520 free (13 chunks)
CachedPlan : 34199136 total in 24 blocks; 7386392 free ( 0 chunks )
```

`0 chunks` means that the context allocates memory not in chunks , but in pieces of variable size, that is, blocks.

Parameter enable_large_allocations

- Tantor DBMS parameter that increases the size of StringBuffer from 1 gigabyte to 2 gigabytes

```
postgres =# select * from pg_settings where name like '%large%'\ gx
name | enable_large_allocations
setting | off
category | Resource Usage/Memory
short_desc | whether to use large memory buffer greater than 1Gb, up to 2Gb
context | superuser
vartype | bool
boot_val | off
```

- can be set at session level and by pg_dump , pg_dumpall utilities

```
postgres@tantor :~$ pg_dump --help | grep alloc
  --enable-large-allocations enable memory allocations with size up to 2Gb
```

- applications, Integrated automation, Manufacturing enterprise management

enable_large_allocations parameter

Tantor 16.2 DBMS parameter that increases the size of StringBuffer in the local memory of instance processes from 1 gigabyte to 2 gigabytes . The size of one table row when executing SQL commands must fit in StringBuffer . If it does not fit, then any client with which the server process works will receive an error, including the `pg_dump utilities` and `pg_dumpall` . The size of a table row field of all types cannot exceed 1GB, but there can be several columns in a table and the row size can exceed both a gigabyte and several gigabytes.

`pg_dump` utility may refuse to dump such lines because it does not use the `WITH BINARY option COPY` commands . For text fields, a non-printable character occupying one byte will be replaced by a sequence of printable characters occupying 2 bytes (for example, \n) and the text field can increase in size up to twice.

```
postgres =# select * from pg_settings where name like '%large%'\ gx
name | enable_large_allocations
setting | off
category | Resource Usage/Memory
short_desc | whether to use large memory buffer greater than 1Gb, up to 2Gb
context | superuser
vartype | bool
boot_val | off
```

and for command line utilities :

```
postgres@tantor :~$ pg_dump --help | grep alloc
  --enable-large-allocations enable memory allocations with size up to 2Gb
```

The parameter can be set at the session level. StringBuffer is allocated dynamically during the processing of each line, and not when the server process is started. If there are no such lines, the parameter does not affect the operation of the server process.

This problem occurs with the config table row 1C:ERP applications, Integrated automation, Manufacturing enterprise management . Example:

```
pg_dump : error: Dumping the contents of table " config " failed: PQgetResult () failed.
Error message from server: ERROR: invalid memory alloc request size 1462250959
The command was: COPY public.config
(filename, creation, modified, attributes, datasize , binarydata ) TO stdout ;
```

2- 3

Page cache

Linux Page Cache

Page cache (operating system cache, file cache):

- cache of 4K pages mapped to files in the file system
- Under the linux page cache can use all free (not occupied by processes and the kernel) physical memory, except for the part whose size is indirectly determined by the `vm.min_free_kbytes parameter`
  - The parameter does not allow using all physical memory for page cache
  - default value is not greater than 66 Mb

Linux Page Cache

Page cache (operating system cache, file cache) is a cache of 4K pages mapped to files in the file system. Current cache size:

```
root@tantor :~# cat /proc/ meminfo | grep Cached
With ached: 898924 kB
SwapCached : 0 kB
```

The command output also includes memory-mapped pages of executable code (executable files and libraries).

Under the linux page cache can use all free memory, except for a part whose size is indirectly determined by the parameter `vm.min_free_kbytes` . This parameter determines the threshold values (boundaries) for triggering the kernel swap daemon processes (`kswapdN`) , by which processes start and stop clearing physical memory from dirty blocks by writing them to their original locations on the disk. This parameter does not allow using all physical memory for the page cache. Also, if a process wants to allocate memory greater than `vm.min_free_kbytes` and available (`free` or `cat /proc/ meminfo | grep Available`) there is no memory or it is fragmented, then `the oom killer is triggered` .

When Linux is running normally the amount of memory not allocated for cache turns out to be approximately equal to the value of `vm.min_free_kbytes` .

The default value (if physical memory is more than 4GB) is small and is equal to 66 MB :

```
root@tantor :~# sysctl -a | grep vm.min_free
vm.min_free_kbytes = 67584
```

The percentage of dirty pages in physical memory is specified by `the vm.dirty_ratio` and `vm.dirty_background_ratio parameters` . The percentage is taken from the amount of physical memory not occupied by processes and the kernel (available), and not from the amount of all physical memory.

Once `vm.dirty_background_ratio is reached` , dirty pages are written to disk: kswapd marks the pages, bdflush initiates the write, p dflush write. Upon reaching `vm.dirty_ratio` (the value must be greater than the first), processes that write to file blocks (dirty pages in the cache) are blocked. This avoids `oom kill` , but introduces delays in the operation of processes (responsiveness). There are no recommendations for exact values, otherwise they would have been set. The typical range of values for these parameters is 5-10% and 10-20%.

https://hydrusnetwork.github.io/hydrus/Fixing_Hydrus_Random_Crashes_Under_Linux.html

Percentage of modified (" dirty ") pages in cache

- The percentage of modified (" dirty ") pages in physical memory is specified by the vm.dirty_background_ratio parameters and vm.dirty_ratio
- The percentage is taken from the volume of physical memory not occupied by processes and the kernel (available), and not from the volume of all physical memory.
- there are also parameters vm.dirty_background_bytes and vm.dirty_bytes , which specify absolute values, default is zero
- current values of the threshold for starting to write dirty pages to their original locations (first in the background without blocking, then with blocking), in the number of pages:

```
root@tantor :~# cat /proc/ vmstat | grep dirty
nr_dirty 4
nr_dirty_threshold 152308
nr_dirty_background_threshold 76061
```


Percentage of modified ("dirty") pages in cache

There are also parameters vm.dirty_background_bytes and vm.dirty_bytes , which set absolute values, zero by default. If you set them, then not less than 1Gb and 100Mb.

Command echo 1 > /proc/sys/ vm / drop_caches releases all clean cache pages. This command can be used before running benchmarks to ensure that subsequent benchmark runs are not affected by pages being cached from the previous benchmark run . After using the command on a host with a large amount of physical memory, a large amount of memory may be freed up. **The percentage of dirty pages will increase dramatically because the percentage of dirty pages held is calculated as a percentage of free physical memory, not total physical memory . The condition of an unfilled cache (a large amount of free memory) is not typical of the normal mode in which the** Linux operating system operates .

View the set values:

```
root@tantor :~# sysctl -a | grep dirty\.*ratio
vm. dirty_background_ratio = 10
vm.dirty_ratio = 20
```

The amount of memory available to processes :

```
root@tantor :~# free
               total used free shared buff/cache available
Meme : 3908744 476408 428256 112560 3004080      3040044
```

Current values **of the threshold for triggering the start of writing dirty pages to their original locations** . First in the background without blocking, then with blocking of processes that dirty (change the contents) pages. Measured in the number of pages:

```
root@tantor :~# cat /proc/ vmstat | grep dirty
nr_dirty 4
nr_dirty_threshold 152308
nr_dirty_background_threshold 76061
```

There are few dirty pages , the cache is busy with blank pages.

The threshold for the start of background recording is approximately equal to formula :

nr_dirty_background_threshold (76061) ~ = 3040044 (free memory in kilobytes) * dirty_background_ratio /4 (page size in kilobytes)

https://www.yugabyte.com/blog/linux-performance-tuning-memory-disk-i o/

Memory fragmentation

- fragmentation of the virtual memory address space is indicated by the fact that there are more than 1000 pieces of memory 16Kb above 16MB no:

```
root@tantor :~# cat /proc/ buddyinfo
Node 0, zone      DMA    1 0 0    1 2 1 1 0 1 1 3
Node 0, zone    DMA32  3173 856 529    0 0 0 0 0 0 0 0
Node 0, zone   Normal 19030 8688 7823    0 0 0 0 0 0 0 0
```

- an example where most of the memory is in chunks no smaller than 4MB:

```
root@tantor :~# cat /proc/ buddyinfo
Node 0, zone      DMA    0 0 0    0 0 0 0 0 0 1 3
Node 0, zone    DMA32    1 2 4    2 3 4 2 2 2 2 863
Node 0, zone   Normal    1 0 3 91 14 10 9 5 5 23 352
```

- each zone is divided into parts of the memory address space of size (4096 bytes *2 ^n): 4 KB, 8 KB, 16 KB, 32Kb, 64Kb, 128Kb, 256Kb, 512Kb, 1Mb, 2Mb, 4Mb .

Memory fragmentation

Fragmentation of the virtual address space is indicated by the fact that there are more than 16 KB of memory pieces above 16 MB. No :

```
root@tantor :~# cat /proc/ buddyinfo
Node 0, zone      DMA    1 0 0 1 2 1 1 0 1 1 3
Node 0, zone    DMA32  3173 856 529    0 0 0 0 0 0 0 0
Node 0, zone   Normal 19030 8688 7823   0 0 0 0 0 0 0 0
```

An example when most of the memory is in chunks no less than 4MB :

```
root@tantor :~# cat /proc/ buddyinfo
Node 0, zone      DMA    0 0 0    0 0 0 0 0 0 1 3
Node 0, zone    DMA32    1 2 4    2 3 4 2 2 2 2 863
Node 0, zone   Normal    1 0 3   91 14 10 9 5 5 23 352
```

Node 0 - physical processor number. Zones :

1)DMA - virtual memory with an offset from zero to 16 MB
2)DMA32 from 16MB to 4GB
3)Normal - from 4GB to 2^48

Each zone is divided into parts of the memory address space of size (4096 bytes *2 ^n): 4 KB, 8 KB, 16 KB, 32Kb, 64Kb, 128Kb, 256Kb, 512Kb, 1Mb, 2Mb, 4Mb .

Based on these data, the derived metric " index " is calculated :

```
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
```

closer to **-1** (often -1.000 is called minus thousand) everything is fine, closer to 1 - the memory is fragmented , closer to 0 - not enough memory (needs to be freed). It is assumed that it can be used to determine whether it is necessary to free up memory or defragment .

What fragmentation looks like from an administrator's perspective:
https://habr.com/ru/companies/odnoklassniki/articles/266005/
Kernel history of changes to deal with fragmentation:
https://habr.com/ru/companies/ruvds/articles/673024/

Memory defragmentation

- The default value for minimum free RAM may not be sufficient:

```
root@tantor :~# sysctl -a | grep min_free
vm.min_free_kbytes = 67584
```

vm.min_free_kbytes

- It is recommended to set it to 2% of the physical memory size.
- vm.watermark_scale_factor parameter sets the second boundary for the defragmentation process
- the default value **is 0.1% (number 10)** of the **free** physical memory size:

```
root@tantor :~# sysctl -a | grep watermark
vm.watermark_boost_factor = 15000
vm.watermark_scale_factor = 10
```


Memory defragmentation

If there is a lot of free memory (it is occupied by the page cache), and at the same time there are out of memory errors, for example, oom kill is launched, then in this case you can effectively eliminate fragmentation by simply freeing up the pieces of memory occupied by " clean " cache pages:

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

or

```
postgres@tantor :~# echo 1 | sudo tee /proc/sys/ vm / drop_caches
```

Dirty pages are written to disk by the **sync command** .

also periodically run defragmentation during minimal loading of the operating system using the command:

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

You can also configure defragmentation. The default value for the minimum free RAM size may not be sufficient:

```
root@tantor :~# sysctl -a | grep min_free
vm.min_free_kbytes = 67584
```

The page cache is the primary fragmenter of the virtual address space. Defragmentation is triggered when free memory drops below vm.min_free_kbytes . Recommended to set to 2% of physical memory size. Recommended range is 1-3%.

Parameter vm.watermark_scale_factor sets the second boundary for the defragmentation process. The default value is 0.1% (number 10) of the **free** physical memory:

```
root@tantor :~# sysctl -a | grep watermark
vm.watermark_boost_factor = 15000
vm.watermark_scale_factor = 10
```

The maximum value of vm.watermark_scale_factor =1000 , which means 10% of free physical memory. It is recommended to set it to 1%, i.e. a value of 100.

Linux Page Cache It cannot be switched off or limited in size.

https://www.alibabacloud.com/help/en/alinux/support/solutions-to-memory-fragmentation-in-linux-operating-systems

Description of drop_caches : https://www.kernel.org/doc/Documentation/sysctl/vm.txt

Dirty Page Retention Duration

- `vm.dirty_expire_centisecs` - how long a buffer can be dirty before it is marked for writing
- `vm.dirty_writeback_centisecs` - the period of waiting between writes to disk
- `net.ipv4.tcp_timestamps` can reduce periodic delays due to timestamp generation

Example settings:

```
vm.dirty_expire_centisecs = 500
vm.dirty_writeback_centisecs = 250
vm.swappiness = 10
vm.dirty_ratio = 10
vm.dirty_background_ratio = 3
net.ipv4.tcp_timestamps=0
```

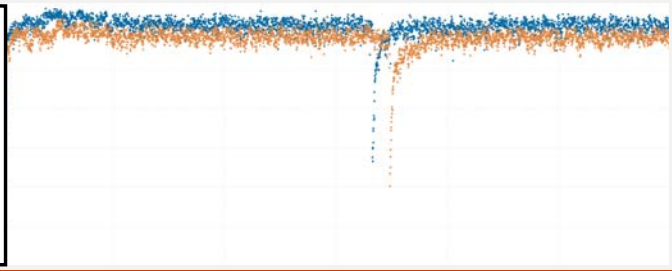

Duration of retention of dirty pages in cache

The duration of page retention from the moment of change is determined by the parameter:
`vm.dirty_expire_centisecs` - how long a buffer can be dirty before it is marked for writing . **The default value is 3000 (30 seconds)** . The value can be reduced to 500 (5 seconds).
You can also set the parameters:
`vm.dirty_writeback_centisecs` - the wait period between writes to disk , default is 500 (5 seconds), can be reduced to 250 (2.5 seconds)
`vm.swappiness = 10`
`vm.dirty_ratio = 10`
`vm.dirty_background_ratio = 3`
`net.ipv4.tcp_timestamps=0`
The last parameter is useful and can reduce periodic delays due to timestamp generation. The parameter adds 12 bytes to the header of each TCP packet with a timestamp . The parameter is described in RFC1323. It can be used in implementations of the BBR algorithm based on network delay measurement . It is also used in the `tcp_tw_reuse option` , which allows reuse of TIME-WAIT sockets in cases where it is considered safe.
https://www.enterprisedb.com/blog/tuning-debian-ubuntu-postgresql

Parameter backend_flush_after

- the number of dirty blocks evicted from the buffer cache by each server process, upon reaching which a command will be sent to evict the file pages that correspond to these blocks from the page cache
- limits the amount of dirty pages in the page cache linux and reduces the likelihood of performance degradation when performing fsync calls on data files at the end of a checkpoint

```
\ 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
```



backend_flush_after parameter

PosgreSQL parameter backend_flush_after can be set to the number of dirty blocks that each server process flushes from the buffer cache before sending a command to flush the file pages that correspond to those blocks from the page cache. This parameter reduces the likelihood of a performance hit when performing fsync calls on files at the end of a checkpoint. The default value is zero (disabled), because it can cause a performance hit. This happens when the number of blocks that all processes in the instance flush between checkpoints is greater than the shared pool, but significantly less than the page cache. This parameter can be set if a performance hit at the end of a checkpoint is noticeable and undesirable.

At the end of a checkpoint, the checkpointer process sends fsync system calls to files, blocks of which were transferred to the page cache . This is done by calling the BufferSync function (flag). Only checkpointer can call this function. Flags when calling the function CHECKPOINT_FLUSH_ALL, CHECKPOINT_IS_SHUTDOWN, CHECKPOINT_END_OF_RECOVERY, CHECKPOINT_IMMEDIATE remove delays in transmitting fsync calls . Block identifiers (BufferTag) are passed to the checkpointer process by server processes. The PendingWriteback and WritebackContext structures are used to store identifiers . Sorting of blocks to determine page ranges is done in the IssuePendingWritebacks (..) function. When displacing one block from the buffer cache, server processes call the ScheduleBufferTagForWriteback (..) function and a check is inserted into it whether the server process has exceeded the backend_flush_after value .

The range of values is from 0 to 256 blocks (2 MB).

```
\ 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
```

The checkpoint_flush_after and bgwriter_flush_after parameters have the same purpose (to reduce the performance drop at the end of a checkpoint), only for blocks sent for writing by processes of the same name.

Practice

- Part 1. Launching a Huge Pages Instance
- Part 2. Changing the oom_score value
- Part 3. Unloading long lines with the `pg_dump utility`
- Part 4. Out of Memory
- Part 5. Enabling swapping
- Part 6. Page cache

Practice

The main goal of all practices is to consolidate the material covered in memory.

Executing the commands described in the chapter helps to better remember the technical details that were studied in the chapter. In practice, you will see what the oom kill process looks like . Watching an instance crash causes emotions in responsible administrators, and emotions help to remember the material studied. Example of using the Tantor DBMS configuration parameter `enable_large_allocations` helps you remember where this setting might be useful .

In the practice, you will launch an instance with Huge Pages and see what commands you can run to verify that they are being used by the instance.

You will trigger OOM kill . The practical task will allow you to understand how memory is accounted for and displayed if processes use shared memory.

You will learn how to call OOM killer with a simple command :

```
select repeat('a', 100 0 000000) from generate_series (1, 1 0 00);
```

You will see how to diagnose memory fragmentation linux and how to defragment it .

3-1

Processors

Simultaneous Multi-Threading (SMT) and Hyper-Threading (HT)

- Hyper-Threading is the name of Intel's technology
- Simultaneous Multi - Threading is the name of AMD technology
- one physical processor core is defined by linux as two virtual (logical) cores
- implements the concept of simultaneous multithreading
- check how many threads there are per core and whether there is SMT support and whether SMT is active :

```
root@tantor :~# lscpu | grep Thread
Thread(s) per core: 1
root@tantor :~# cat /sys/devices/system/ cpu / smt /active
0
root@tantor :~# cat /sys/devices/system/ cpu / smt /control
not supported
```


Simultaneous Multi-Threading (SMT) and Hyper-Threading (HT)

Simultaneous multithreading (S imultaneous Multi - Threading (SMT) name of the technology . The technology is implemented in AMD processors. In Intel processors , the technology is called Hyper-Threading (HT). One physical core of the processor is defined by Linux as two " logical " (as an antonym for the word " physical ") cores. The physical core can store the state of two execution threads (threads), contains one set of registers and one interrupt controller (APIC) for each logical core. The remaining resources of the physical core are shared by the logical cores. For the operating system, this looks like two logical cores.

When executing a stream of commands from a logical core, a physical core will pause execution and start executing commands from another core if:

a cache miss occurred while accessing the processor cache ;

a branch misprediction occurred;

The result of the previous instruction is expected.

While the logical core is waiting for data to be received from memory into the processor cache, the computing resources of the physical core will be used by the instruction flow of the second logical core.

The CPU's computing power is usually not a bottleneck in a DBMS. Using multithreading can lead to a high frequency of context switches and cache reloads . In this case, the CPU uses not only the instance code, but also the Linux kernel code . It is better to test whether multithreading will improve the overall performance of the instance.

Enabling and disabling multithreading is performed in BIOS.

`pgbench` test results with scale (`-s`) 300:

```
clients | tps (HT)| tps ( no HT)
--------+---------+------------
4 | 517 | 520
8 | 1013 | 999
16 | 1938 | 1913
32 | 3574 | 3560
64 | 5873 | 5412
128 | 8351 | 7450
256 | 9426 | 7840
512 | 9357 | 7288
```

https://www.postgresql.org/message-id/53FD5D6C.40105%40catalyst.net.nz

https://elibsystem.ru/node/490

Process affinity (CPU affinity)

- increases the probability of data processes getting into processor caches (TLB and others)
- can be ignored if:
  - processor one
  - the load on the processor cores is low (less than 80%)
- example of binding running process :

```
root@tantor :~# ps - ef | grep check
postgres 2181 2179 0 00:00:01 postgres: checkpointer
root@tantor :~# taskset -p 1 2181
pid 2181's current affinity mask: f
pid 2181's new affinity mask: 1
```


Process affinity (CPU affinity)

Processor affinity is an instruction to the operating system scheduler to restrict the execution of a process to one or more processors. several processors. The idea is to increase the probability of getting data from processes into various hardware caches , such as TLB . A tied process will not load its data into the caches of other processors.

The process itself can programmatically manage the binding. PostgreSQL relies on the operating system to access hardware resources, and the PostgreSQL developers did not implement binding.

To bind processes to a processor, you can use the **taskset utility** .

If there is only one processor, binding does not provide any advantages, since the processor caches are shared by all cores of one processor . If there are several processors and instances, then binding the processes of different instances to different processors can increase performance if the load on the processor cores is high (more than 80-90%). If there is no load on the processor cores, then the cores are not a bottleneck and processes rarely migrate between cores.

Example of viewing which core (psr) a process is running on:

```
root@tantor :~# ps - eLo psr ,pid,cmd --headers | grep checkpointer
  3    2181 postgres: 15/main: checkpointer
1 126674 postgres: checkpointer
```

Example of binding running process:

```
root@tantor :~# taskset - p -c 1 2181
pid 2181's current affinity mask: f
pid 2181's new affinity mask: 1
```

Example output current binding running process:

```
root@tantor :~# taskset -p 2181
pid 2181's current affinity mask: 1
```

https://www.postgresql.org/message-id/4B0ADE3F.2080703%40meteorsolutions.com

Viewing the list of processes using the `ps utility`

- `ps` command line utility gives a list of running processes
- Using pipes you can format the output of the utility
- An example of the first 4 processes in the output, including the first row of the header describing the value in the columns:

```
ps -A -o pid,psr,cmd | head - 4
PID PSR CMD
1 2 / sbin / init splash
2 1 [ kthreadd ]
3 0 [ pool_workqueue_release ]
```

- example output of processes named postgres and their sorting by column `pss` in descending order :

```
p.s. -C postgres -o pcpu,vsz,rss,pss,rops,wops,cmd --sort - pss | head -4
%CPU VSZ RSS PSS ROPS WOPS CMD
0.0 231628 18416 14263 5 3069 postgres : checkpointer
0.0 233712 16172 9491 87 26 postgres : postgres postgres [local] idle
0.0 231516 6768 4301 34 98 postgres : walwriter
```


Viewing the list of processes using the `ps utility`

ps command line utility gives a list of running processes and / or threads. The utility produces a static report and after it is issued, it terminates its work, unlike interactive utilities `top, htop , atop` . The ps utility collects data from the / proc directory , which contains information about all processes, and presents it in a convenient form.

Using pipes you can format the output of the utility. An example of the first 4 processes in the output, including the first line of the header describing the value in the columns:

```
ps -A -o pid,psr,cmd | head - 4
PID PSR CMD
1 2 / sbin / init splash
2 1 [ kthreadd ]
3 0 [ pool_workqueue_release ]
```

Example of outputting a list of processes whose names contain the letter combination postgres and limiting the output to the first two lines:

```
ps -A -o pid,psr,cmd | grep postgres | head - 2
1458 3 / usr /lib/ postgresql /15/bin/ postgres -D / var /lib/ postgresql /15/main
            -c config_file =/etc/ postgresql /15/main/ postgresql.conf
1696 3 postgres : 15/main: logger
```

An example of the output of the last 8 processes, in the command name (parameter cmd) there is a combination of letters postgres :

```
ps -o pid,psr,cmd -C postgres | tail -8
20406 3 postgres : checkpointer
20407 1 postgres : background writer
20409 3 postgres : walwriter
20410 2 postgres : autovacuum launcher
20411 1 postgres : autoprewarm leader
20412 2 postgres : pg_wait_sampling collector
20413 2 postgres : logical replication launcher
65139 3 postgres : postgres postgres [local] idle
```

Parameter -C Specifies the name of the command that runs the process.

Example of using the `-o parameter` to list the columns you want to see in the utility output:

```
ps -e - o user,pcpu,vsz,rss,pss,cls,nlwp,psr,rops,wops,ppid,pid,tid,s,cmd - -sort - wops
```

This is also an example of output sorted by the `wops column.` in descending order

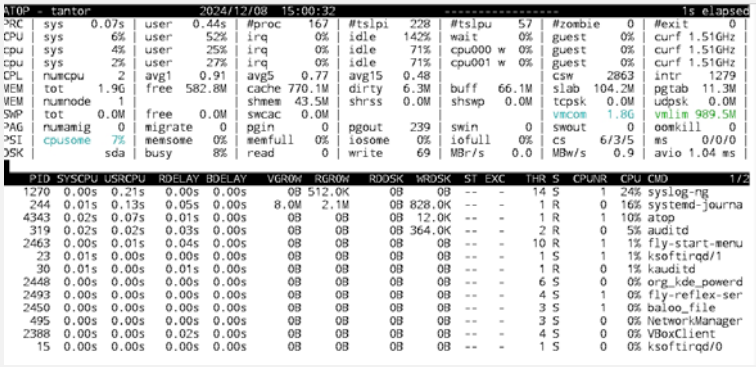
Description of columns:
USER is the name of the user with whose rights the process is running.
% CPU - CPU core load in %
VSZ - virtual memory in KB
RSS - resident (located in physical memory, not swapped out) memory size in KB
PSS - proportional memory size in KB
CLS - Process Scheduler Policy
NLWP number of process threads
PSR - the number of the processor core (cpu) on which the process or thread is running
ROPS - Input / Output Operations Ratio
WOPS - number of input / output operations
RBYTES - number of bytes read
WBYTES - number of bytes written
PPID - PID of the parent process
PID - PID of the process itself
TID - PID of the thread (LWP)
S - process status
CMD - the command that launched the process with parameters

Not all column names that the ps utility can output are listed.

Recording and viewing metrics with the `atop utility`

- for monitoring processes and the current load It is convenient to use the command line utility `top`
- `top` utility that it is usually installed by default
- atop utility allows you to record operating system performance indicators into a binary file and then visualize the collected indicators:

```
root@tantor :~# apt install atop -y
root@tantor :~# dpkg -S / usr /bin/atop
atop: / usr /bin/atop
root@tantor :~# atop -w / atop.record 1 15
root@tantor :~# atop -r / atop.record
```



Recording and viewing metrics with the `atop utility`

For monitoring processes and the current load It is convenient to use the command line utility `top` . The advantage of the `top` utility is that it is usually installed by default . The `top` utility included in the `procps package` :

```
root@tantor :~# dpkg -S / usr /bin/top
procps : / usr /bin/top
```

On hosts in production use, installing additional software is usually difficult. The probability of installing simple and standard utilities is higher. Utility atop is available in the extended repository Astralinux .

atop utility is that it allows you to write operating system performance indicators to a binary file and then visualize the collected indicators. You can visualize the collected metrics on another host.

To record, use the command:

`atop -w / path_to_file interval recording_duration`

The interval (default 10 seconds) and recording duration (default infinite) are specified in seconds.

To play, use the command:

`atop -r / path_to_file`

When playing, the keyboard key ' **`t`** ' switches to displaying the next time interval, the key in uppercase ' **`T`** ' to go back to the previous interval. The ' **`b`** ' key to go to the time, which will be prompted to enter in the format `[YYYYMMDD] hhmm` . The ' **`r`** ' key to return to the beginning of the file.

The second advantage of the atop utility is that metrics in binary files can be visualized in Grafana :

https://github.com/rchakode/atop-graphite-grafana-monitoring

the top and atop utilities, the htop utility can be used . Advantages of the htop utility V color pseudographics and saved settings.

https://wiki.astralinux.ru/tandocs/instruktsiya-dlya-podgotovki-k-nagruzochnomu-testirovaniyu-302054346.html

Switching execution context

- voluntary context switching:
  - the process cannot execute its code because it is waiting for an I/O operation to be performed, or for a lock to be acquired
- involuntary context switching:
  - the process exceeds time (timeslice) allocated to it by the scheduler
  - in accordance with the policy established for the process the scheduler has the right to suspend the execution of a process (displace the process from the processor core)
- a persistently high number of involuntary context switches indicates that there is too much parallelism for the number of processor cores

```
root@tantor :~# grep ctxt /proc/212233/status
voluntary_ctxt_switches : 0
nonvoluntary_ctxt_switches : 62728
```


Switching execution context

There are usually fewer processor cores than processes. A voluntary context switch (VCX) occurs when a process makes a system call (disk or network I/O, waiting for a lock) to the kernel code (sys) and waits for a result, and until the result is received, it cannot perform its code (user) . An involuntary context switch (ICX) occurs when the scheduler suspends execution of a process's code because the process's code exceeds the time slice the scheduler has allocated to the process or a higher-priority process has arrived. ICX number should be ten times less than VCX . The opposite indicates that the number of active processes (the degree of parallelization of some task: the pool of sessions with the database, the number of workers) is too large. VCX and ICX You can see it in the statistics of a specific process :

```
root@tantor :~# grep ctxt /proc/212233/status
voluntary_ctxt_switches : 0
nonvoluntary_ctxt_switches : 62728
```

or **`perf stat - p PID`** . System-wide **`perf stat - a sleep 1`** .

perf does not output ICX , which makes the utility useless. Other **utilities (`pidstat` , `vmstat`) average values and indicators (cs , nvcswch / s) and will be misleading.** For example, a process can be preempted 100 times per second on a core there might be 500 switches per second, a process on that core might have 10 switches per second, and in the operating system the average number of switches per process is 0.2 per second. The time is determined by the scheduler policy, but not lower than `kernel.sched_rr_timeslice_ms` , by default 100 milliseconds , (1 /10 seconds, zero means the default value) the value of which can be changed. By default, the `SCHED_OTHER policy is used.` - Completely Fair Scheduler (CFS) up to version 6.6 of the Linux kernel , after Earliest Eligible Virtual Deadline First (EEVDF). Time (`timeslice`) is floating .

The disadvantage of CFS is that under load, CFS introduces a delay before the task (process that has become active) starts executing and this delay is added to the response time , reducing the responsiveness of the process . EEVDF eliminates this problem.

You can view the kernel version using the command:

```
root@tantor :~# cat /proc/version
Linux version 6.6 .28-1-generic
```

https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/sched_policy_prio/start

Operating system scheduler

- By default, CFS uses the `SCHED_OTHER policy` with the algorithm:
  - CFS - up to linux kernel version 6.6
  - EEVDF - replaces CFS since Linux kernel version 6.6 , eliminates delay at the start of task execution
- minimum time a process can run before being evicted: not less than `kernel.sched_rr_timeslice_ms` , default is 1/10 second, value can be changed
- policies can take into account process priority or nice or ignore them
- policies play a role when processors are fully loaded
- The policy can be changed for each process:

```
root@tantor :~# chrt -r -p 10 96878
root@tantor :~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_RR
pid 96878's current scheduling priority: 10
```


Operating system scheduler

The 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 - to receive timeslices with some frequency) include: SCHED_FIFO - is preempted only by processes with higher priority or the SCHED_DEADLINE policy, which leads to the process occupying the core for a long time; SCED_RR (Round Robin Scheduler) - the time is the same and equal to `kernel.sched_rr_timeslice_ms` . Processes with SCHED_DEADLINE can preempt processes with SCHED_FIFO and SCHED_RR, that is, they have the highest priority.

The scheduler policy can be changed for a process. Current policy:

```
root@tantor :~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_OTHER
pid 96878's current scheduling priority: 0
```

Install new:

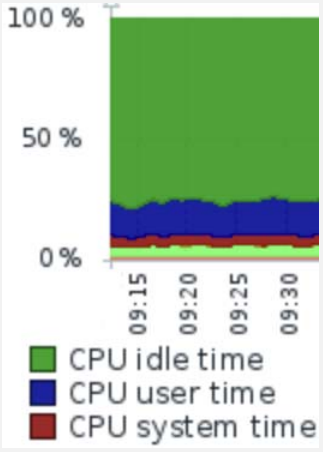
```
root@tantor :~# chrt - r -p 10 96878
root@tantor :~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_RR
pid 96878's current scheduling priority: 10
root@tantor :~# chrt - f -p 10 96878
root@tantor :~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_ FIFO
pid 96878's current scheduling priority: 10
```

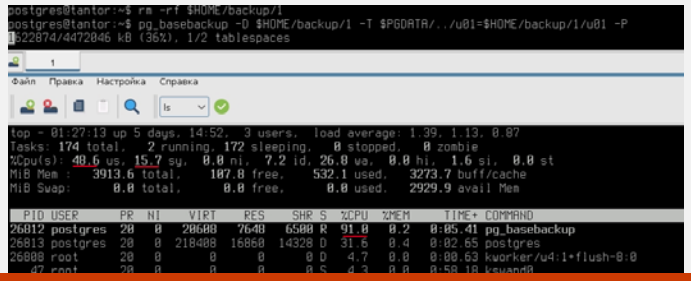
Set the process with pid=96878 to a guaranteed 5 milliseconds execution time with a period of 15 milliseconds and a deadline of 10 ms:

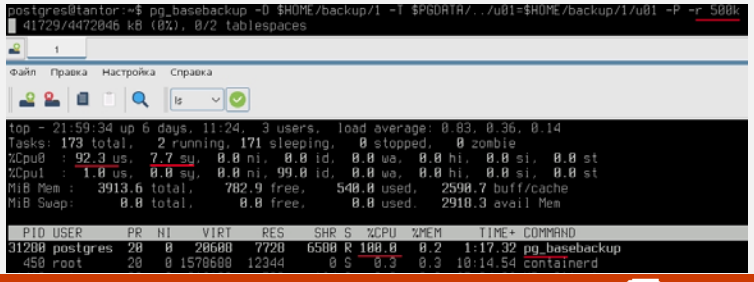
```
root@tantor :~# chrt - d -- sched -runtime 5000000 -- sched -deadline 10000000 -
- sched -period 15000000 -p 0 96878
root@tantor :~# chrt -p 96878
pid 96878's current scheduling policy: SCHED_ DEADLINE
pid 96878's current scheduling priority: 0
pid 96878's current runtime/deadline/period parameters:
5000000/10000000/15000000
```

CPU usage by application code and kernel (USER/SYS ratio)

- the User CPU time/System CPU time ratio should be around 60/40
- if User is larger it means inefficient application code
- if System is bigger - linux is experiencing problems







CPU usage by application code and kernel (USER/SYS ratio)

USER/SYS CPU Time metric shows the proportion of processor time used by application code and kernel code.

For quick monitoring you can use the top command .

Values:

`us` - User CPU time , the processor time spent by the application process code that was not niced .

`sy` - System CPU time , CPU time spent by linux kernel code .

`ni` - CPU time spent by the application process code that is niced .

nice property taken into account by some types of schedulers.

When the application and all processes are working normally, the total **proportion is (us+ny)/ sy should be approximately 60/40 .** For example: `us=30, sy =20, ni =0 .`

If the proportion is strongly (80 /20) shifted towards **us** , it means that the application code is inefficient. If it is shifted towards **sy** This means that the operating system is experiencing problems.

Example of a problem (error) in the application code (pg_basebackup utility code) 16 versions):

```
postgres@tantor :~$ time pg_basebackup -c fast -D $HOME/backup/1 -P -r 10M
51752/51752 kB (100%), 1/1 tablespace
real 0m7.647s
user 0m4.991s
sys 0m0.091s
```

When using throttling (slowing down the backup), one of the processor cores is loaded at 100% and `us/ sy` = ~100/1 . The code implementing throttling has an error. The `mpstat utility` does not allow diagnosing the problem:

```
root@tantor :~# mpstat -n
Linux 6.6.28-1-generic ( tantor ) _x86_64_ (4 CPU)
NODE % usr % nice %sys % iowait %soft %idle
all 2.27 0.03 0.23 0.11 0.29 97.08
```

gives % idle=97.08% and everything looks good - there is no load . If you calculate the USER/SYS ratio in the **`mpstat utility`** (2.27+0.03%)/ 0.23 % which is equal to 10/1 (real ~ 100/1), you can see that 10/1 is far from the proportion 60/40 .

Note: A bug in the `pg_basebackup code` that caused the CPU core to load at 100% when throttling was enabled was fixed in PostgreSQL version 17.

Time source

- are also used by the Linux kernel and applications for obtaining timestamps
- during boot linux checks available time sources and selects one to use
- fastest source: Time Stamp Counter (TSC)
- ACPI Power Management Timer (ACPI_PM) is several times slower
- Available sources:

```
cat /sys/devices/system/ clocksource /clocksource0/ available_clocksource
tsc hpet acpi_pm
```

```
cat /sys/devices/system/ clocksource /clocksource0/ current_clocksource
acpi_pm
```


Time source

The clock source is used by the Linux kernel. and programs for obtaining timestamps. For example, linux can mark each network packet with a timestamp. Also marks are used to ensure that audio and video playback is smooth and not floating. Time sources can be accessed quite frequently. The detail (frequency, fine grain) of time stamps, real accuracy (drift, jitter), absence of desynchronization between processor cores, absence of issuing stamps to different processes in reverse order, i.e. not corresponding to the course of real time are important. This occurs due to low-level out-of-order exe optimizations with ution . Using reverse order for timestamps can lead to poorly diagnosed errors in the operation of programs , especially third-party " cluster solutions " .

During boot, linux checks for available time sources and selects one to use. Preferred `Time Stamp Counter (TSC)` . It uses the RDTSC processor instruction to obtain a stamp , which causes a 64-bit integer quantum (tick, jiffy) of time to appear in the processor register. The number is reset when the power is reset, sleep. You can view the parameters of the time counter with the command `cat /proc/cpuinfo` . The command can return the value `rdtsp` (the processor has a register that outputs the TSC time) or `constant_tsc` (the processor normalizes the ticks so that they do not depend on the processor frequency but correspond to real time). The next preferred source is integrated into the chipset : `High Precision Event Timer (HPET)` . The technology was created by Intel and Microsoft. One of the problems with the technology is that the access time to the counter levels out its detail. HPET is banned for use with Intel Coffee Lake processors starting with the Linux kernel version 5.4 due to inaccuracy . HPET is used for TSC calibration. If HPET is disabled in BIOS or prohibited in Linux , then another method (PMTIMER) is used for calibration . Next source: ACPI Power Management Timer (ACPI_PM , aka PMTIMER). The following Programmable Interval Timer (PIT) and Real Time Clock (RTC), they are less preferred.

List of sources that linux considered it possible to use :

```
cat /sys/devices/system/ clocksource /clocksource0/ available_clocksource
tsc hpet acpi_pm
```

Source used:

```
cat /sys/devices/system/ clocksource /clocksource0/ current_clocksource
acpi_pm
```

Comparison of time sources

- to check the speed of the time source you can use a program that requests the time many times

for acpi_pm :
```
time ./ clock_timing
real 0m38.889s
user 0m15.760s
sys 0m23.126s
```

For tsc :
```
time ./ clock_timing
real 0m13,967s
user 0m13,938s
sys 0m0,008s
```

- **speed difference between acpi_pm and tsc significant : 2.8 times**
- `real` : the time from program invocation to termination. real includes user and sys and may be greater than their sum if the program was preempted by the scheduler (involuntary context switches).
- `user` : time of execution of program code.
- `sys` : Linux kernel code execution time (working with hardware, memory, files, threads, network)

Comparison of time sources

You can check the speed of the time source by creating a file `clock_timing.c` :

```
#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;
}
```

compile the file with the command:

`gcc clock_timing.c -o clock_timing - lrt`

and check what load reading the time **`10000000`** times gives.

For `acpi_pm` :

```
time ./ clock_timing
real 0m38.889s
user 0m15.760s
sys 0m23.126s
```

For `tsc` :

```
time ./ clock_timing
real 0m13,967s
user 0m13,938s
sys 0m0,008s
```

The difference is 2.8 times .

If the BIOS has an option to enable / disable HPET , then HPET should be disabled, as it usually reduces performance.

You can replace `CLOCK_MONOTONIC` on `CLOCK_MONOTONIC_COARSE` and the speed will increase 50 times.

`real` : time from program call to completion. `real` includes `user` plus `sys` and may be greater than their sum if the program was preempted by the scheduler (involuntary context switches).

`user` : time of execution of program code.

`sys` : execution time of the Linux kernel code (working with hardware, memory, files, threads, network).

Comparing Time Sources in PostgreSQL

- the time source is actively used when enabling configuration options `track_wal_io_timing , track_io_timing , track_commit_timestamp`
- An example of the effect of choosing a time source on the execution time of the `explain analyze command :`

```
postgres=# \! sudo sh -c 'echo acpi_pm > /sys/devices/system/ clocksource /clocksource0/ current_clocksource
'
postgres=# explain analyze select count( pk ) from t;
QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual time=836.212..836.228 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.019..408.416 rows=100000 loops=>
Planning Time: 0.056 ms
Execution Time: 836.334 ms
(4 rows)
postgres=# \! sudo sh -c 'echo tsc > /sys/devices/system/ clocksource /clocksource0/ current_clocksource '
postgres=# explain analyze select count( pk ) from t;
QUERY PLAN
-----------------------------------------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual time=308.180..308.187 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.022..153.991 rows=100000 loops=>
Planning Time: 0.375 ms
Execution Time: 308.373 ms
(4 rows)
```


Comparing Time Sources in PostgreSQL

source can be actively used by instance processes. An example is the explain analyze command. Time is actively read when `the track_wal_io_timing , track_io_timing , track_commit_timestamp configuration parameters are enabled .`

An example of how the time source affects the explain analyze command:

```
postgres=# drop table if exists t;
create table t( pk bigserial, c1 text default ' aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ');
insert into t select *, 'a' from generate_series (1, 100000);
DROP TABLE
CREATE TABLE
INSERT 0 100000
ALTER TABLE
postgres=# \! sudo sh -c 'echo acpi_pm > /sys/devices/system/ clocksource /clocksource0/ current_clocksource '
postgres=# explain analyze select count( pk ) from t;
QUERY PLAN
-----------------------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual time=836.212..836.228 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.019..408.416 rows=100000
Planning Time: 0.056 ms
Execution Time: 836.334 ms
(4 rows)
postgres=# \! sudo sh -c 'echo tsc > /sys/devices/system/ clocksource /clocksource0/ current_clocksource '
postgres=# explain analyze select count( pk ) from t;
QUERY PLAN
-----------------------------------------------------------------------------------------
Aggregate (cost=1791.00..1791.01 rows=1 width=8) (actual time=308.180..308.187 rows=1 loops=1)
-> Seq Scan on t (cost=0.00..1541.00 rows=100000 width=8) (actual time=0.022..153.991 rows=100000
Planning Time: 0.375 ms
Execution Time: 308.373 ms
(4 rows)
```

In addition to the WAL synchronization method testing utility `pg_test_fsync` , PostgreSQL also comes with a command-line utility for testing the time source speed **`pg_test_timing`** .

https://docs.tantorlabs.ru/tdb/ru/16_4/se/ **pgtesttiming** .html

Replacing the time source

- when booting linux the time source acpi_pm can be selected , which is slower than tsc
- to change source to tsc need to be added to file `/etc/default/grub` after `quiet splash` options:

```
clocksource = tsc nohpet processor.max_cstate =1 intel_idle.max_cstate =0
```

- and run the `update-grub command`
- after reboot check that `tsc is used` And `max_cstate =0` :

```
cat /sys/devices/system/ clocksource /clocksource0/ current_clocksource
tsc
cat /sys/module/ intel_idle /parameters/ max_cstate
0
```

- You can check the speed and stability of the time source using the command line utility `pg_test_timing`
  - the utility also gives the distribution of delays in the speed of time transfer

Replacing the time source

When booting linux the acpi_pm time source may be randomly selected , which is slower than tsc . This may be due to hardware errors (https://bugzilla.kernel.org/show_bug.cgi?id=203183).

HPET is not selectable, but is used for TSC calibration in recent kernels . If HPET cannot be used, PIT or PMTIMER is used for calibration.

To change the source to tsc need to add to file `/etc/default/grub` after `quiet splash` parameters:

```
nohpet processor.max_cstate =1 intel_idle.max_cstate =0
```

or

```
clocksource = tsc nohpet processor.max_cstate =1 intel_idle.max_cstate =0
```

and run the **`update-grub command`** , which will update the `/boot/grub/grub.cfg file`

In addition to an incorrectly selected time source due to running in a virtual machine, hardware errors can also occur. For example, the TSC clock can stop or become desynchronized when the processor goes into power saving states (sleep) or when the processor frequency changes. This can look like this hanging linux when loading, when stopping. In such cases, use workaround , disabling " energy saving " parameters in /grub.cfg :

```
processor.max_cstate =1 intel_idle.max_cstate =0 idle=poll
```

Check which CSTATEs are used:

```
cat /sys/module/ intel_idle /parameters/ max_cstate
0
```

Using `idle=poll` may result in higher heat output on processors if they are not constantly under load .

If in the list:

```
cat /sys/devices/system/ clocksource /clocksource0/ available_clocksource
tsc hpet acpi_pm
```

there is a source, then you can change the clock source without rebooting using the command:

```
echo tsc > /sys/devices/system/ clocksource /clocksource0/ current_clocksource
```

You can check the speed and stability of the time source using the command line utility `pg_test_timing`

https://linuxreviews.org/Linux_Kernel_Disables_Coffee_Lakes_HPET_On_The_Grounds_That_It_Is_%22Unreliable%22

You can also check with the command line utility `pg_test_timing`
The difference will be similar.

acpi_pm :

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 3998.44 ns
Histogram of timing durations:
< us % of total count
1 0.00147 11
2 0.00200 15
4 21.37522 160377
8 78.16336 586455
16 0.24870 1866
32 0.08970 673
64 0.09663 725
128 0.01839 138
```

tsc :

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 1388.24 ns
Histogram of timing durations:
< us % of total count
1 0.00801 173
2 63.83282 1379438
4 36,02110 778422
8 0.00139 30
16 0.07807 1687
32 0.04262 921
64 0.01328 287
128 0.00199 43
```

Linux builds may run different time sources on the same hardware.
Example of messages when tsc not selectable:

dmesg | grep tsc

```
tsc : Marking TSC unstable due to clocksource watchdog
TSC found unstable after boot , most likely due to broken BIOS. Use ' tsc =unstable'.
clocksource : Checking clocksource tsc synchronization from CPU 1 to CPUs 0.3.
clocksource : Override clocksource tsc is unstable and not HRT
compatible - cannot switch while in HRT/NOHZ mode
```

Messages about instability may be related to hardware errors .
Example of messages after **clocksource = tsc** without **nohpet**

dmesg | grep tsc

```
tsc : Fast TSC calibration using PIT
tsc : Detected 3600.150 MHz processor
Kernel command line: BOOT_IMAGE=/boot/vmlinuz-6.6.28-1-generic root=UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 ro
quiet splash clocksource = tsc processor.max_cstate =1 intel_idle.max_cstate =0 parsec.max_ilev =0 parsec.mac=0
pcie_aspm =performance
clocksource : tsc -early: mask: 0xffffffffffffff max_cycles : 0x33e4e0fd970, max_idle_ns : 440795362981 ns
clocksource : Switched to clocksource tsc -early
tsc : Refined TSC clocksource calibration: 3600.003 MHz
clocksource : tsc : mask: 0xffffffffffffff max_cycles : 0x33e4564530a, max_idle_ns : 440795343825 ns
clocksource : Switched to clocksource tsc
clocksource : wd- tsc -wd read-back delay of 427149ns, clock-skew test skipped!
```

Example of messages after **nohpet**

```
tsc : Fast TSC calibration failed
tsc : Unable to calibrate against PIT
tsc : using PMTIMER reference calibration
tsc : Detected 3599.954 MHz processor
Kernel command line: BOOT_IMAGE=/boot/vmlinuz-6.6.28-1-generic root=UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 ro
quiet splash nohpet processor.max_cstate =1 intel_idle.max_cstate =0 parsec.max_ilev =0 parsec.mac=0 pcie_aspm
=performance
clocksource : tsc -early: mask: 0xffffffffffffff max_cycles : 0x33e42840770, max_idle_ns : 440795330420 ns
clocksource : Switched to clocksource tsc -early
tsc : Refined TSC clocksource calibration: 3600.002 MHz
clocksource : tsc : mask: 0xffffffffffffff max_cycles : 0x33e4559ce44, max_idle_ns : 440795364889 ns
clocksource : Switched to clocksource tsc
clocksource : wd- tsc -wd read-back delay of 159517ns, clock-skew test skipped!
```

3-2

Net

Basic network parameters

- Main parameters:
  - bandwidth
  - network latency
  - presence and frequency of network failures
- throughput is usually not a bottleneck
- instance uses:
  - unix sockets for local connections
  - TCP/IP for connections via network interfaces
- network latency is important when synchronously committing transactions with confirmation by replica

Basic network parameters

The main network parameters are: bandwidth , network latency , and network failure rate (including packet loss). Postgresql uses unix sockets for local connections and TCP/IP over network interfaces. The network is not usually a bottleneck for the DBMS. The amount of data transferred to clients during normal operation is not very large. The network can be a bottleneck when using synchronous transaction commit, where network latency is important. Typical network latency is 8-25 microseconds. Using RDMA (Remote Direct Memory Access) theoretically allows to reduce the latency to 1-2 microseconds, and also increase the actual throughput by reducing the load on the central processors. This is achieved by the remote host's network card controller writing to the address space of the memory of an arbitrary process on the remote host, which unloads the central processors from operations related to data transfer via network interfaces (marketing term zero-copy) . There are RDMA over Converged Ethernet (RoCE) standards in Ethernet and built-in functionality in InfiniBand solutions . In 2019, NVIDIA acquired the only manufacturer of Infiniband equipment Mellanox company . In Oracle Exadata X8M and newer 100 gigabit Ethernet is used (for example, Cisco Nexus 9336C-FX2 switches) .

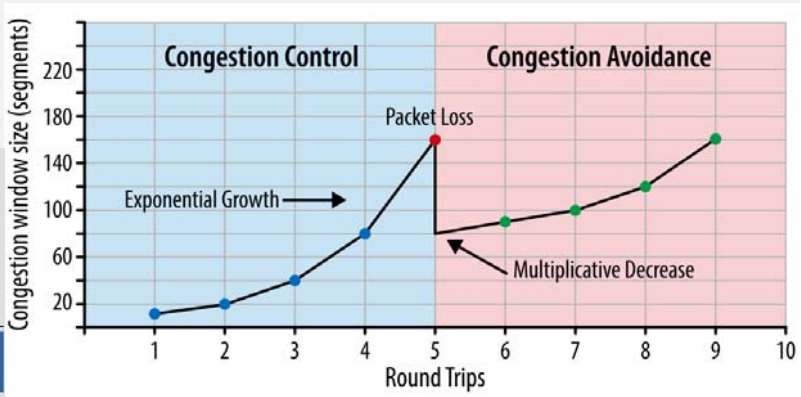
RDMA technology was used in the fork of Postgres Pro Enterprise 10 and 11 versions (the parameter `listen_rdma_addresses`) , in newer versions of the fork the technology is not mentioned and the parameter is missing. When using up to 3 synchronous replicas and Mellanox equipment the effect was not too noticeable.

Under high load, practical performance is determined by the balance of the hardware used, not by the use of any technology. RDMA can be used in the Tantor hardware and software complex xData for backup speeds of 2.5 gigabytes per second (10 terabytes per hour).

https://ibs.ru/media/superkompyuternoe-mezhsoedinenie-v-mashinakh-baz-dannykh/

Congestion and slow start algorithms

- `net.ipv4.tcp_available_congestion_control` defines the algorithm for selecting the data transfer rate separately for each network connection (socket)
- `net.ipv4.tcp_slow_start_after_idle=0` disables slow start
- 3 packets are transmitted without confirmation
  package size
  1095-2190 bytes



Algorithms with ongestion and slow start

In low latency networks, when the transmission channel is not used at 100%, there is no packet loss, and small amounts of data are transmitted (3 packets of 1095 to 2190 bytes). (according to the formula from RFC5681) network parameters do not affect performance. If load balancers are used on intermediate hosts, then network latency and differences in network card characteristics may play a role. If there is one network interface and it is used for backup at maximum speed, then connections of server processes with clients via the same network interface may experience delays in data transfer. Why does this happen ? Let's imagine that a process (server or wal_sender) must pass through the socket TCP field (data) of 500 MB. It will transmit not by byte and wait for confirmation, the process will indicate " transmit " with a system call and a reference to the memory area with 500 MB. Then, let's say the operating system has a tcp buffer 10 MB in size and it transmits all 10 MB in a stream to the transmission medium, and the operating system of the receiving process has a 1 MB buffer and a slow processor. 9 MB will not be accepted (ignored) because there is nowhere to receive them, and the transmission medium was occupied with all 10 MB. P o tcp 9MB will be retransmitted sometime when it is found that they were not received. That is, there may be situations when data is " dropped " because the other side or the intermediate process (balancer) cannot accumulate data coming from the transmission medium. On the other hand, underloading of the channel is also bad, underloading appears if the network delay is relatively high. At the transmission medium level (channel and other levels), MTU and other parameters are set, jumbo frames are used. We consider the TCP level , since this protocol uses PostgreSQL .

The congestion window value is reset after idle time. This may impact the performance of long-lived TCP connections that may be idle due to client inactivity. **It is better to disable slow start on the server to improve the performance of long-lived connections** :

`net.ipv4.tcp_slow_start_after_idle = 0 .`

tcp level the algorithm is responsible for selecting the initial volume of data to be transferred

`net.ipv4.tcp_congestion_control = cubic`

`net.core.default_qdisc = pfifo_fast`

https://habr.com/ru/companies/yandex/articles/533530/

https://habr.com/ru/companies/webo/articles/327050/

BBR (Bottleneck Bandwidth and Round Trip Time) Algorithm

- by default the CUBIC algorithm is used
- CUBIC is sensitive to packet loss, BBR is not
- BBR, when fully loaded on the transmission medium, consumes all available bandwidth and displaces other sockets that use c ubic and other algorithms
- inclusion:
  - **`sysctl -w net.ipv4.tcp_congestion_control= bbr`**
  - **`sysctl -w net.core.default_qdisc = fq`**
  - **`net.ipv4.tcp_timestamps = 1`**

BBR (Bottleneck) Algorithm Bandwidth and Round Trip Time)

tcp level the algorithm is responsible for selecting the initial volume of data to be transferred

BBR support is available since the Linux kernel 4.9 . The fact that it is not issued does not mean that BBR is not supported. By default in Linux cubic is used since kernel 2.6.19 (this algorithm is also used in Windows 10) .

BBR (Bottleneck Bandwidth and Round Trip Time , channel width and round trip packet transmission time metrics) is one of the algorithms that determines how packets go into the network. Developed by Google in 2016. The algorithm is configured by two related parameters. For BBR , you need to set `net.core.default_qdisc = fq` , otherwise, when the network is fully loaded, what can sockets with BBR do, other sockets will not be able to transmit data. Why does this happen ? The BBR algorithm is **not based on packet loss, but on channel width and network latency** . The algorithm is not sensitive to packet loss for any reason.

Other algorithms set the volume of transmitted data by measuring data loss, which is undesirable and leads to their retransmission. Loss begins, volumes are reduced.

These algorithms are very sensitive to packet loss for any reason.

Sockets using algorithms that rely on packet loss will fail to transmit packets when the channel is fully utilized. Sockets with BBR do not rely on data loss. BBR consumes all available bandwidth and displaces other sockets that use c ubic (RFC 8312) and other algorithms.

What controls the packet sending speed? The `pacing technique` , which is implemented in the FQ scheduler. The scheduler setting and do not depend on the value of `net.core.default_qdisc`

Which side do the parameters apply to ? The side that initiates the transmission. The receiving side can use any settings and not support BBR.

Therefore, test measurements may show that in one direction the data transfer graph increases and the speed is high, and in the opposite direction there is a surge, then a dip and settles at a low speed. In this case, you need to set the values on the client or unload the transmission channels from the client to the server.

Monitoring:

```
ss -tin
ss --options --extended --memory --processes --info
```

https://djangocas.dev/blog/huge-improve-network-performance-by-change-tcp-congestion-control-to-bbr/

Network connection parameters

- `tcp_user_timeout` how long transmitted data can remain unacknowledged before a decision is made to forcibly close the TCP connection
- `client_connection_check_interval` (in milliseconds) interval between checks during command execution by polling the state of a socket on which no data is being transmitted

```
root@tantor :~# sysctl -a | grep keepalive
net.ipv4.tcp_keepalive_intvl = 75 -> 20
net.ipv4.tcp_keepalive_probes = 9 -> 5
net.ipv4.tcp_keepalive_time = 7200 -> 240

net.ipv4.tcp_slow_start_after_idle = 1 -> 0
net.ipv4.tcp_retries2 = 15 - > 3
net.ipv4.tcp_timestamps = 1 -> 0
```


Network connection parameters

Resilience of network connections, especially under load, helps avoid performance degradation associated with retrying actions that failed due to a network failure.

The following parameters have default values of zero, which is the default interval for sockets in Linux . The default values are conservative and can be reduced:

```
root@tantor:~# sysctl - a | grep keepalive
net.ipv4.tcp_keepalive_intvl = 75 -> 20
net.ipv4.tcp_keepalive_probes = 9 -> 5
net.ipv4.tcp_keepalive_time = 7200 -> 240
```

`tcp_user_timeout` interval (in milliseconds) during which transmitted data may remain unacknowledged before a decision is made to forcibly close the TCP connection. `tcp_user_timeout` must be set in the range from `( tcp_keepalives_idle + tcp_keepalives_interval * ( tcp_keepalives_count - 1))` to **`( tcp_keepalives_idle + tcp_keepalives_interval * tcp_keepalives_count )`** . Can be set to an upper bound .

`tcp_keepalives_idle` (in seconds) the period of no traffic, after which Linux sends a TCP packet to keep the connection. `tcp_keepalives_interval` (in seconds) the interval for re-sending a connection-keeping packet if the other side has not responded to the first packet.

`tcp_keepalives_count` the number of connection-keeping packets sent after which the connection will be broken.

`client_connection_check_interval` (in milliseconds) interval between checks during command execution by polling the socket state when no data is being transmitted. Checking is disabled by default. The socket may have been closed by the other side or by the operating system kernel code due to the other side not responding to keepalive packets .

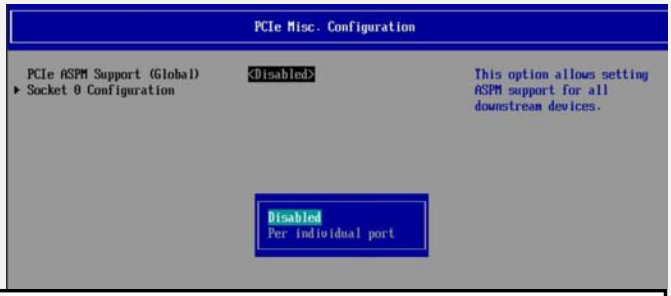
The values should not cause a false positive. There are also parameters `idle_in_transaction_session_timeout` , `idle_session_timeout` , `transaction_timeout` , `wal_receiver_timeout` , `wal_sender_timeout` , but they usually have **larger** values than the socket parameters .

https://www.postgresql.org/message-id/flat/160741519849.701.13355787096244067178%40wrigleys.postgresql.org

https://blog.cloudflare.com/when-tcp-sockets-refuse-to-die/

Energy saving parameters

- there is no need to enable it in the operating system
- due to implementation errors, it may reduce performance
- part of the settings in the firmware




```
root@student :~# cat /sys/module/ pcie_aspm /parameters/policy
[default] performance powersave powersupersave
root@student :~# echo performance > /sys/module/ pcie_aspm
/parameters/policy
root@student :~# cat /sys/module/ pcie_aspm /parameters/policy
default [ performance ] powersave powersupersave
```


Energy saving parameters

There may be power saving settings in the BIOS . For example, Intel TurboBoost , HyperThread , C-states, DDR Frequency. It is worth checking the values, as well as the virtualization parameters. On data center equipment, power saving parameters may be enabled by default, since increased power consumption leads to increased heating. Data centers pay attention to PM (power management) and RM (resource scheduling) . Usually, the BIOS allows the operating system to manage power saving. Usually, the operating system highlights several plans (policies), among them maximum performance, maximum power saving. When choosing something other than " performance " (maximum performance), the operating system refers to the firmware , which reduces power consumption. The problem is that the power saving functionality has errors in implementation. For example, a peripheral device may not exit power saving mode correctly. For example, high-speed (100 GbE) network cards may start working at half the speed. Peripheral energy saving does not generate much heat and there is no point in energy saving for a server with a DBMS. Manufacturers may explain implementation errors as the fact that when using Active-State Power Management (ASPM) adds a delay when waking the device from a low power level (https://edc.intel.com/content/www/us/en/design/products/ethernet/config-guide-e810-dpdk/active-state-power-management/) .

Equipment manufacturers may include functionality that increases performance, but since it leads to overheating, it cannot work constantly and must be switched off after some time (TurboBoost). Such functionality is unlikely to be included, since uneven speed of equipment operation can disrupt the logic of the operating system and applications, if they have self-tuning, it is unlikely to take into account sudden changes in equipment performance.

Optimal settings that guarantee the declared performance indicators are found in software and hardware systems. For example, arbitrary changes in the contents of memory cells during frequent reading of adjacent memory areas (" rowhammer ") can be found in chips from some manufacturers and absent from others, regardless of the presence of ECC parity control (does not recognize changes in three bits), which is indirectly (they select components for which there are few warranty claims) taken into account by manufacturers of hardware systems.

Practice

- Part 1. Standard pgbench test
- Part 2. Binding processes to a processor core
- Part 3. Switching execution contexts
- Part 4. Monitoring CPU load
- Part 5. Collecting statistics into a file and viewing it with the `atop utility`
- Part 6 . Linux Time Source
- Part 7. Network connections
- Part 8. Replacing the scheduling policy and checking the scheduler operation

Practice

In practice, what was studied in the chapter is consolidated.
You will learn
bind processes to processor cores ;
diagnose context switches and see how linux utilities perf , pidstat which commands give out useless data that is misleading and which commands give out real data ;
the EXPLAIN ANALYZE command several times.
You will change the parameters of the linux scheduler and you will see that the perf utility does not give the number of forced context switches. Also see how to get real data on context switches.

4

Storage system

Disk subsystem

- synonyms: input/output (I/O), data storage system (DSS)
- the most loaded DBMS resource from the main resources:
  - disk
  - CPU
  - memory
  - net
- instance can use direct I / O, in version 16 does not provide better fault tolerance and performance
- Setting up the disk subsystem includes:
  - choice of storage systems (eg SSD or HDD)
  - i / o selection scheduler
- Configuring cluster and tablespace parameters
- creating mount points, selecting file systems, volume managers, mount options
- I/O monitoring and related settings

Disk subsystem

The disk subsystem (input-output, storage system) is the most loaded resource of the DBMS from the main resources:
disk
CPU
memory
net.

The database cluster is stored in files in the PGDATA directory of the file system and other directories that are symbolically linked in the `PGDATA/ pg_tblspc directory` . By default, normal system calls are used when working with files and the work is done through the page cache.

The instance can use direct I/O (direct i / o , O_DIRECT). Theoretically, such access has advantages, but historically, I / O work was optimized for regular file access, not direct. Direct access in version 16 does not provide better fault tolerance and performance. When opening a file with O_DIRECT, the operating system does not cache its contents, but also does not ensure that it is written, that is, it does not send the ATA flush command to the disk controller and the write goes to the disk cache. To ensure that the write is done, processes will still need to perform fsync , or disable write caching in the disk firmware . Direct access is enabled by the `debug_io_direct parameter` . By default, an empty string (direct I / O is disabled). The value can be words separated by commas: `data` (direct access to data files), `wal` (direct access to WAL files), `wal_init` (direct access when creating WAL files).
Setting up the disk subsystem includes:
1.choice of storage systems (eg SSD or HDD)
2.i/o scheduler selection
3.Configuring cluster and tablespace parameters
4.PGDATA subdirectories , selecting file systems, volume managers for mount points, selecting mount options
5.monitoring the load on input / output devices, adjusting parameters and redistributing mount points across devices
6.Related settings: configuring cluster parameters to ensure fault-tolerant and efficient writing to WAL, monitoring free space and configuring junk file cleaning to free up free space in a timely manner.

HDD, SSD, NVMe

- SSD (solid-state drive) - solid-state (on microcircuits) " disk "
- NVMe , (Non-Volatile Memory express, non-volatile memory) - an interface for accessing SSDs connected via the PCIe bus
- SATA (Serial Advanced Technology Attachment) is a transport protocol that defines the interaction between the controller and storage devices.
- SAS (Serial Attached SCSI) - a set of SCSI commands over a physical interface similar to SATA
- AHCI (Advanced Host Controller Interface) is a standard that describes operations with SATA controllers.
  - one queue for each port with a depth of up to 32 commands
  - hot swapping of devices is supported
- M.2 (NGFF, Next Generation Form Factor) is a common name for the form factor and physical interface for SSDs, has 4 PCIe lanes with a total speed of 4 or 8 gigabytes per second
- SATA speed - 600 megabytes per second

HDD, SSD, NVMe

HDD (hard disk drive) - a hard disk, characterized by a delay in random access to data, acceptable sequential speed, low cost per storage unit. Lifespan depends on operating time.

SSD (solid-state drive, solid-state device, i.e. without moving parts) - a data storage device on microcircuits (NAND), the placement and reading of data is performed by the controller chip. More expensive than HDD , equally fast random and sequential access speed. The service life depends on the amount of recorded data. The recording speed depends on the volume of stored data.

NVMe (Non-Volatile Memory express) is an interface for accessing SSDs connected via the PCIe bus .

PCIe (Peripheral Component Interconnect Express) has high throughput and low latency. NVMe devices They may have random access memory (DRAM) of about 1 GB in size, in which they cache the data allocation table, they may not have it, or they may use the main memory of the host of about 32 MB (HMB, Host Memory Buffer) , using the direct memory access (DMA) capability of the PCIe bus .

SATA (Serial Advanced Technology Attachment) is a transport protocol (a set of commands and the sequence of their use) that defines the interaction between the controller and storage devices. SATA is also the name of the technical specifications followed by cable and connector manufacturers.

SAS (Serial Attached SCSI) is a set of SCSI commands over a physical interface similar to SATA. The advantages of SAS are the ability to connect a device simultaneously via several channels, and a cable length of up to 10 meters instead of 1 meter.

AHCI (Advanced Host Controller Interface) is a standard developed by Intel that describes operations with SATA controllers that serve storage devices such as HDD, SSD , DVD. Devices can support NCQ (a proprietary command queue that allows the device to accept more than one command at a time and independently determine which command from the queue will be executed first. In AHCI, each port (interface with the device) can have one queue with a depth of up to 32 commands (in SAS devices up to 256). The AHCI standard includes hot-plugging . This means that the operating system, when working with SATA controllers that support physical connection/disconnection of devices, will be able to process such events.

SATA speed is 600 megabytes per second. SAS has a maximum of 2400 MB, which is several times lower than the speed of M.2 PCIe 4x4 (version 4 lines 4) NVMe .

For comparison, USB (Universal Serial Bus) transfer speeds are:

3.0 (aka 3.1 Gen1, aka 3.2 Gen1) - 500 MB/ s

3.1 (aka 3.1 Gen2, aka 3.2 Gen2) - 1200MB/ s

3.2 (aka 3.2 Gen 2x2) - 2400MB/ s

USB4 - 40Gbps or 20Gbps , backward compatible with USB 3.2 and 2.0.

SATA interfaces can be used because the hardware is not very expensive.

SAS controllers can support connecting devices with a SATA interface, either directly connected using the SATA protocol, through expansion cards using tunneling via the STP (SATA Tunneled Protocol).

M.2 (NGFF, Next Generation Form Factor) is a common name for the form factor and physical interface for SSDs, WiFi adapters , Bluetooth 4G modems, and other devices. It is called so because it is considered the second version of the outdated Mini connector. PCIe , which also had up to 4 PCIe lines and one SATA. Devices with an M.2 connector can use any of the connector buses, so knowing that the device has an M.2 connector is not enough. For example, a 4G modem can use some PCIe lines or one USB.

Devices with an M.2 connector may have cutouts (" Key "). Commonly used are:

B.Key PCIe x2 (two lanes), SATA, USB, PMC, IUM, SSIC, I2C

M.Key PCIe x4 or SATA

B/M Key PCIe x2 or SATA.

Throughput of one PCIe lane :

version 3.x in each direction - slightly less than 1 Gigabyte per second

PCIe version 4 - 2GB/ s . **When using 4 lines (" PCIe 4x4") - slightly less than 8GB/ s**

SSDs of the M.2 form factor with SATA interface have a speed of no more than 600 MB/ s .

Manufacturers of boards for installing SSDs make an M.2 connector with an M key and two interfaces to choose from: PCIe or SATA. There are exceptions when the M.2 connector on the board is connected only to the PCIe bus or only to the SATA controller.

Intel Technology Optane (an alternative to NAND) was not developed. It was notable for the fact that in addition to the PCIe interface , it could use the DDR4 slot.

NVMe specification , which is useful for interpreting NVMe- related acronyms and metrics :

https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.1-2024.08.05-Ratified.pdf

Link to specifications page:

https://nvmexpress.org/specifications/

Block devices

- a type of special file in Linux that provides an interface for accessing a device (or a regular file)
- reading and writing occurs in blocks of equal size
- are located in the / dev directory mounted on the dev tmpfs virtual file system
- / dev contains files only for those devices that are currently available (connected)
- devices are detected are specified in files in the /lib/udev/rules.d directory
- list of block devices:

```
root@tantor :~# ls -l /dev | grep br
b rw-rw ---- 1 root disk 7 , 0 date time loop0
b rw-rw ---- 1 root disk 8 , 0 date time sda
b rw-rw ---- 1 root disk 8 , 1 date time sda1
b rw-rw ----+ 1 root cdrom    11 , 0 date time sr0
```


Block devices

Block device (block device) - a type of special file in Linux that provides an interface for accessing a device (or a regular file). They are called block files because reading and writing occurs in blocks of equal size.

Access is arbitrary, the block ordinal number is specified. Direct input/output can be used for access.

A block device file can reference a disk, a disk partition, or a volume.

File systems use block devices when accessing storage media.

Block devices are located in the / dev directory mounted on the devtmpfs virtual file system :

```
root@tantor :~# mount | grep /dev
udev on /dev type devtmpfs ( rw,nosuid,relatime,size =968888k,
                             nr_inodes =242222,mode=755,inode64)
hugetlbfs on /dev/ hugepages type hugetlbfs ( rw,relatime,pagesize =2M)
```

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

The udev process is responsible for this , it receives events that are generated when the device is initialized or removed. The files specified in the /lib/udev/rules.d directory The rules are checked against the event properties and the matching rules are executed and can create device files, run programs and command files to initialize and configure devices. For example, mount file systems when a drive is connected.

List of block devices:

```
root@tantor :~# ls -l /dev | grep br
b rw-rw ---- 1 root disk 7 , 0 date time loop0
b rw-rw ---- 1 root disk 8 , 0 date time sda
b rw-rw ---- 1 root disk 8 , 1 date time sda1
b rw-rw ----+ 1 root cdrom    11 , 0 date time sr0
```

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.

There is also a directory / sys / dev / block with symbolic links to devices.

I / O Scheduler

- when using high-speed NVMe devices , there is no point in using any scheduler, i.e. it is worth setting the scheduler **`to none`**
- NVMe memory controllers on PCIe low-latency buses handle the flow of parallel requests without putting a strain on CPUs and processor caches , unlike scheduler code
- The purpose of schedulers is to queue requests from a large number of processes, passing requests to the storage controller one after another, so as not to overload the controller
- modern planners: **`none mq-deadline kyber bfq`**

I / O Scheduler

The operating system scheduler was discussed earlier; it schedules the use of CPU resources and is not related to the I/O scheduler.

The I/O scheduler determines the order in which block I/O operations are submitted to storage devices. The purpose of the I / O scheduler is to optimize the processing of disk requests to improve I/O performance and throughput. When a system has a large number of processes sending I/O requests to the operating system, a queue of such requests is formed.

When using high-speed devices (NVMe), there is no point in using any scheduler, i.e. you can set the name to none . NVMe memory device controllers on low-latency buses (PCIe) cope with the flow of parallel requests and do not load the central processors and processor caches , unlike the scheduler code (especially those with many queues, mq). With high-speed devices, you can use the kyber scheduler if you need to reduce the read latency by 2-8 times by reducing the throughput (20-30%) and increasing the write latency (~50%).

Schedulers since Linux kernel versions 4.12: **`none mq-deadline kyber bfq`**

deadline logic means that the criterion is the length of time a request has been in the queue. It is guaranteed that each request will be serviced by the scheduler. By default, priority is given to read requests.

After the advent of NVMe SSDs, it became clear that the scheduler code only reduces performance and elegant software algorithms would have to be abandoned (use none), the blk-mq scheduler was created .

mq-deadline is a deadline implementation using blk-mq .

kyber uses two queues for write and read requests, kyber prioritizes read requests over write requests. The algorithm measures the completion time of each request and adjusts the actual queue size to achieve the configured latencies. Kyber can be used with fast devices and aims to reduce read latency, with priority for synchronous requests.

Recommendation for virtual machines:

https://access.redhat.com/solutions/5427

Changing the I/O Scheduler

- the scheduler type can be set separately for different devices and their types S S D or HDD
- view scheduler for block device :

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

- change the scheduler to another one without rebooting:

```
root@tantor:~# echo none > /sys/block/ sda /queue/scheduler
```

- To create a permanent rule, create or edit the file /etc/udev/rules.d/70-schedulerset.rules

```
ACTION==" add|change ", KERNEL==" sd [az] ", TEST!="queue/rotational",
                                              ATTR{queue/scheduler}=" none
"
```


Changing the I/O Scheduler

Type i/os with heduler used when working with a block device can be viewed:

```
root@tantor :~# cat /sys/dev/block/ 8:0 /queue/scheduler
```

or

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

Using `mq -deadline .`

Changing the scheduler without rebooting:

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

To create a permanent rule, you can create (you can choose the file name yourself) or edit the existing file `/ etc/ udev / rules.d / 70-schedulerset.rules`

and add the necessary lines of the form:

```
ACTION==" add|change ", KERNEL==" sd [a-z] ", ATTR{queue/rotational}==" 0 ",
ATTR{queue/scheduler}="none"
```

Where

`ATTR{queue/scheduler}="none"` name of the desired scheduler for devices named sda , sdb .. sdz .

`ATTR{queue/rotational}==" 0 "` if the driver reports that the device has the same random and sequential read speed.

`ATTR{queue/rotational}==" 1 "` if random access is slower.

If the driver does not set the attribute to 0 or 1, you can specify:

```
TEST!="queue/rotational"
```

In AHCI, there is one queue per port with a depth of 32 commands (i/o request, RQ-SIZE commands `lsblk -td`). In NVMe usually 256 , but can be up to 64000.

Physical sector of the disk

- the smallest unit of storage that a physical storage device can write atomically
- usually has a size of 512KB or 4KB
- for NVMe , Linux uses the value of the Atomic parameter Write Unit Power Fail (AWUPF) if provided by the hardware

Physical sector disk

A physical sector is the smallest unit of storage that a physical storage device can write " atomically " , meaning either completely written or not written at all. Typically, a physical sector is 512K or 4K.

For NVMe , Linux uses the **Atomic** hardware parameter value for the physical sector **Write Unit Power Fail** (AWUPF), if the hardware provides it to the driver.

The logical sector is used to read and write to the storage device at the software level by the operating system. The size of the logical sector may differ from the size of the physical sector. You can view the sizes using the commands:

```
root@tantor :~# fdisk -l | grep size
Sector size ( logical / physical ): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
root@tantor :~/ flashbench -dev# lsblk -td
NAME ALIGNMENT MIN-IO OPT-IO PHY-SEC LOG-SEC ROTA SCHED RQ-SIZE RA
sda 0 512 0 512 512 0 none 32 128
sr0 0 512 0 512 512 1 mq -deadline 2 12 8
nvme0n1 0 512 0 512 512 0 mq -deadline 256 128
```

Some NVMe and SATA drives support changing the reported sector size using standard NVMe (Format NVM from the NVM Command Set Specification) or ATA4 (SET SECTOR CONFIGURATION EXT) commands. For hard drives, this changes the logical sector size to match the physical sector size for optimal performance. For NVMe , both the logical and physical sector values are changed.

At the first level, the N AND chip consists of several targets , each containing several dies . Each die is an independent storage unit, which consists of several layers (planes) . Several layers share the same bus and can be combined into a single unit for multi -layer parallel operations. Each layer consists of several erase units .

erase unit size determines the granularity of the discard /trim at the firmware level of the SSD controller (SoC , system-on-chip). Each erase block consists of several pages. A page is the smallest unit of recording, typically 16 Kilobyte . The size of the erase block varies widely, from 1 MB to tens of MB.

To improve performance, SSDs can use DRAM to cache both data and Flash Translation Layer (FTL) mapping tables. FTL is similar to a journaling file system. FTL writes data by adding records to a journal file. It uses a Logical address mapping table B lock A ddress -Physical Block Address . The problem is garbage collection - cleaning up unnecessary records.

To improve performance, a list is usually used rather than complex hierarchical structures, but the list is large, ~1/1000 of the SSD volume . For a terabyte SSD , this is 1GB. SSDs typically use 4KB sectors, which is optimal for Linux . To reduce costs (save DRAM), some manufacturers use 16KB sectors.

In addition to performance, fault tolerance is important when SoC when the power supply is lost, it must have time to write the accumulated data. This is achieved by having **capacitors in the power supply circuits** of SSDs for industrial use. Garbage collection, wear leveling of erase blocks, redundancy, and data storage increase the internal traffic of the SSD .

The NVMe 1.4 specification includes metrics that a manufacturer can provide: Preferred Write Alignment, Preferred Write Granularity, Deallocate alignment , and granularity statistics.

When analyzing performance, you should not use `sysbench` , since this utility uses test files filled with zeros. For testing, you should use the fio utilities or flashbench .

Testing with `fsync` after each entry:

```
root@tantor :~# fio --filename=/dev/nvme0n1 --name=a -- blocksize =8k -- rw = randrw -- iodepth =32 --runtime=10 -- rwmixread =90 -- fsync = 1
READ: bw =27.1MiB/s (28.4MB/s), 27.1MiB/s-27.1MiB/s (28.4MB/s-28.4MB/s), io =271MiB (284MB), run=10001-10001msec
WRITE: bw =3099KiB/s (3173kB/s), 3099KiB/s-3099KiB/s (3173kB/s-3173kB/s), io =30.3MiB ( 31.7MB ) , run=10001-10001msec
```

Without `fsync` :

```
root@tantor :~# fio --filename=/dev/nvme0n1 --name=b -- blocksize =8k -- rw = randrw -- iodepth =32 --runtime=10 -- rwmixread =90 -- fsync = 0
READ: bw =30.6MiB/s (32.1MB/s), 30.6MiB/s-30.6MiB/s (32.1MB/s-32.1MB/s), io =306MiB (321MB), run=10001-10001msec
WRITE: bw =3523KiB/s (3607kB/s), 3523KiB/s-3523KiB/s (3607kB/s-3607kB/s), io =34.4MiB ( 36.1MB ) , run=10001-10001msec
```

Determining the erase block size by testing with the https://github.com/bradfa/flashbench utility:

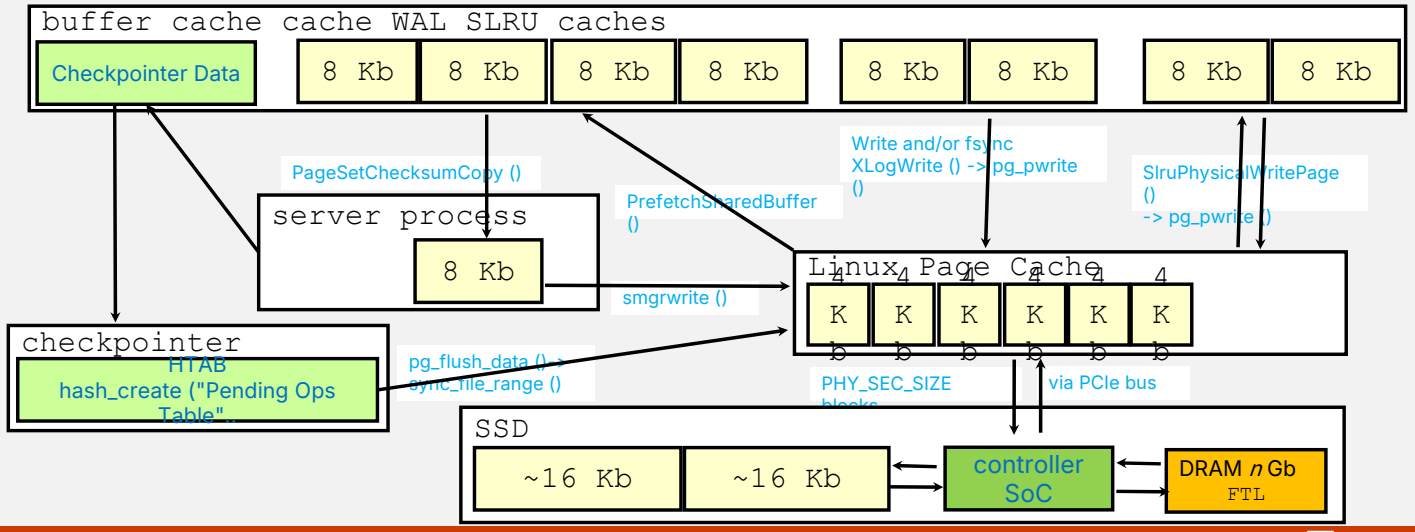
```
root@tantor :~/ flashbench -dev# ./ flashbench -a /dev/nvme0n1 -- blocksize = 1024
align 33554432 pre 213µs on 287µs post 224µs diff 68µs
align 16777216 pre 199µs on 248µs post 210µs diff 43.4µs
align 8388608  pre 159µs on 230µs post 130µs diff 85.7 µs
align 4194304 pre 228µs on 259µs post 257µs diff 16µs
align 2097152 pre 197µs on 236µs post 217µs diff 29.4µs
align 1048576 pre 171µs on 208µs post 210µs diff 17.4µs
```

The utility can be used to determine the optimal RAID stripe size.

Example of checking the recording by fsync when powering off:
https://habr.com/ru/companies/selectel/articles/521168/ with the diskchecker.pl script

Interaction of instance processes with disk

- 8K blocks are read into shared memory via page cache (4K blocks)
- blocks are written to disk via the page cache
- An optimized synchronization algorithm is used for recording



Interaction of instance processes with disk

When reading into the buffer cache, the linux recommendation can be used calculate the next two pages of 4 KB (total PostgreSQL block size BLCKSZ = 8 KB):
`PrefetchSharedBuffer () -> smgrprefetch () -> smgrsw []. smgrprefetch = mdprefetch -> FilePrefetch (,BLCKSZ,) - > posix_fadvise (.., BLCKSZ, POSIX_FADV_WILLNEED)`

To synchronize dirty buffers, `the sync_file_range ( fd , offset, nbytes , SYNC_FILE_RANGE_WRITE) call is used` . The `posix_fadvise ( fd , offset, nbytes , POSIX_FADV_DONTNEED) call` is not used by default, since it has a side effect - in addition to writing changed pages, it removes both the changed and unchanged page from memory.

For each file writeback calls are performed by block ranges. References to blocks for synchronization (in the future) are written to a hash table of 100 blocks, created by the `hash_create function ("Pending Ops Table" or " pending sync hash "..)` **in local memory** checkpointer , sorted by sort_pending_writebacks (..) to arrange the blocks for the block range transfer . **fsync () is executed once for each file (where at least one block has changed) at the end of the checkpoint.**

Synchronization requires remembering all files that have changed since the last checkpoint so that synchronization can be completed before the next checkpoint is complete. Hash table (not linked list) is chosen to eliminate duplication of commands (operations) for writing the same block. Blocks that need to be synchronized are stored in hash tables. For file deletion commands, linked list is used , since there should not be any repeated file deletion commands (operations).

Processes submit operations to the checkpointer process through the `CheckpointerShmemStruct shared memory structure.` named "Checkpointer Data" . The list of shared structures and their sizes are available in the `pg_shmem_allocations view` .

The number of synchronization calls a checkpoint process makes before going to sleep is limited by the constants:

```
/* Intervals for calling AbsorbSyncRequests */
#define FSYNCS_PER_ABSORB   10
#define UNLINKS_PER_ABSORB 1 0
/* interval for calling AbsorbSyncRequests in CheckpointWriteDelay */
#define WRITES_PER_ABSORB   1000
```

Temporary tables are not synchronized because they do not require fault tolerance.
https://medium.com/@hnasr/following-a-database-read-to-the-metal-a187541333c2

Synchronizing data files with disk

- performs mainly checkpointer
- is performed on block ranges by the system call: `sync_file_range ( fd , offset, nbytes , SYNC_FILE_RANGE_WRITE)`
- if a write error is returned for the system call , the instance will be aborted and the blocks will be restored from the WAL logs thanks to their full images (configuration parameter `full_page_writes = on`), if `data_sync_retry =off`

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

```
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
------------------+--------+--------+-----------+-----------+--------+------------
background worker | 7 | 0 | 0 | 43 | 0 | 0
client backend | 7451184 | 83639 | 0 | 15253670 | 0 | 0
 autovacuum worker | 61803 | 705 | 0 | 94784 | 0 | 0
background writer | 2344704 | 26415 | 2344704 | | 0 | 0
 checkpointer | 5595043 | 99818 | 5598779 | | 84627 | 843454
(5 rows)
```


Synchronizing data files with disk

On Linux to perform fsync **for data files** the system call is used:
`sync_file_range` `( fd , offset, nbytes , SYNC_FILE_RANGE_WRITE );`
The call checks that the file's page range has been flushed to disk by the operating system.
The checkpointer process does most of the fsync and only part of the write (copying blocks from the buffer cache to the linux page cache).
If the default checkpoint_flush_after is nonzero, then synchronization by file block ranges is enabled because in the PostgreSQL source code :

```
 /* Default and maximum values for backend_flush_after , bgwriter_flush_after and checkpoint_flush_after ;
measured in blocks. Currently, these are enabled by default if sync_file_range () exists, ie , only on Linux .
Perhaps we could also enable by default if we have mmap and msync (MS_ASYNC)? */
 # ifdef HAVE_SYNC_FILE_RANGE
 #define DEFAULT_ BACKEND _FLUSH_AFTER 0 /* never enabled by default */
 #define DEFAULT_BGWRITER_FLUSH_AFTER 64
 #define DEFAULT_CHECKPOINT_FLUSH_AFTER 32
 #else
 #define DEFAULT_BACKEND_FLUSH_AFTER 0
 #define DEFAULT_BGWRITER_FLUSH_AFTER 0
 #define DEFAULT_CHECKPOINT_FLUSH_AFTER 0
 #endif
```

Usually, server processes do not synchronize data files, this is done by checkpointer and bgwriter . If in the Checkpointer Data shared memory structure (from which checkpointer moves identifiers to a hash table in its local memory " Pending Ops Table "), in which all processes send block identifiers for synchronization, there will be no space, the server process will start to execute fsyncs . An example will be considered in part 2 of the practice for Chapter 14 " Executing fsyncs with a stopped checkpointer " .
If the operating system refuses to make the call with the error " not implemented " , then the diagnostic log displays a warning: " `could not flush dirty data: ` ".
If the operating system fails to perform the synchronization system call and returns a write error, the instance is immediately stopped with a PANIC status and the blocks that linux issued a failure in synchronization will be restored by WAL from the moment of the beginning of the last checkpoint and using a full image of the block (full page image), and therefore are guaranteed to be restored. You can disable a crash with PANIC by setting the parameter data_sync_retry to on (default is off) , but you should not do this, as the data blocks will be damaged.
https://habr.com/ru/articles/803347/

File system block size

- must be equal to the page size - 4Kb
- ext4 file system best choice
- default ext4 parameters are optimal
- list of options with which the file system is mounted: **`cat /proc/ fs /ext4/sda1/options`**

File system block size

On 4Kn disks (4096 byte physical sector size and 4096 byte logical sector size), the `mkfs utility` will use a block size of 4096 bytes.

On 512e (4096-byte physical sector size and 512-byte logical sector size) and 512n (512-byte physical sector size and 512-byte logical sector size) disks, mkfs.ext4 defaults to using 1024-byte blocks for file systems smaller than 512 MB, and 4096-byte blocks for file systems larger than 512 MB.

Linux will only mount filesystems with a block size less than or equal to the memory page size, i.e. 4K.

file system is optimal for the PostgreSQL DBMS .

Mount parameters are specified in:

```
root@tantor :~# cat /etc/ fstab | grep ext4
UUID=aca1a090-eba2-49ba-a8fc-ba12e9e2bf26 / ext4 defaults 1 1
```

defaults means that the parameters are taken from the file system itself (superblock). The parameters can be viewed and changed using the `tune2fs utility` :

```
root@tantor :~# tune2fs -l /dev/sda1 | grep opt
Default mount options: user_xattr acl
```

The parameters with which the file system is mounted:

```
root@tantor :~# mount | grep ext4
/dev/sda1 on / type ext4 ( rw,relatime )
```

Full list of options: **`cat /proc/ fs /ext4/sda1/options`**

default parameters of the ext4 file system being created are optimal, when searching for recommendations, you need to pay attention to whether they are outdated. For example, the recommendation to mount with the `noatime option` deprecated in favor of the faster `realtime option` . Parameter `no barrier` does not significantly increase performance . `The data=ordered` parameter means that data blocks are written before metadata is written to the file system journal. The parameter `data= writeback` causes file contents to be corrupted.

https://www.enterprisedb.com/blog/postgres-vs-file-systems-performance-comparison

wal_sync_method parameter

- a system call that is used to ensure that block records are saved to the WAL file if the fsync parameter is set to on
- fdatasync - default value for linux

```
postgres=# show fsync ;
 fsync
-------
 on
postgres=# show wal_sync_method ;
 wal_sync_method
-----------------
 fdatasync
```


wal_sync_method parameter

The parameter specifies the method that is used to ensure that records are saved in the WAL file if the fsync parameter value is on . Possible values:

fdatasync - fdatasync () is called on each (taking into account group commits, which will be discussed in 6 pages) transaction commit by the server process and on closing the WAL file if it is full. **Default value for Linux .** Space for WAL files is pre-allocated and files are created in advance, so synchronization of file system metadata is not needed, hence fdatasync fault tolerance same as fsync .

open_datasync - the WAL file is opened for writing with the O_DSYNC parameter

fsync - fsync () is called every time a transaction is committed by the server process (taking into account transaction grouping) and when the WAL file is closed if the file is completely full

open_sync - the WAL file is opened for writing with the O_SYNC parameter .

fsync parameter is set to on , the instance does not ensure that the write to the WAL file has been completed. Modified blocks may remain in the operating system cache by default for up to 5 seconds (the commit parameter of the ext4 file system) and will be lost if the operating system suddenly stops or the power is lost.

This parameter does not affect synchronization of data files, only WAL files. When synchronizing data files , fsync system calls are used , fdatasync is not used with data files, since data files can grow in size .

WAL write guarantee

- `fsync ()` or `fdatasync ()` system calls , called after a COMMIT log record is written to the WAL with `wal_sync_method = fdatasync` or `fsync` ensure that the writing of data blocks to the WAL file is completed by the disk controller before the result from these functions is returned.
- before writing to a WAL file, it is created (renamed) in full size and while writing to it, its metadata does not change and the guarantee of writing to the file system journal for writing to WAL does not matter

WAL write guarantee

Calls to the operating system `fsync ()` And `fdatasync ()` guarantee that the writing of data blocks to the WAL file will be performed by the disk controller: " includes writing through or flushing a disk cache if present. The call blocks until the device reports that the transfer has completed" as described in the call description:

https://man7.org/linux/man-pages/man2/fsync.2.html

Since the WAL file is created (renamed) to full size before being written to, the attributes of the WAL file do not change during writing to the WAL file, and the JBD2 log does not play a role while the instance processes are writing to the WAL file .

For reference, let's look at how fault tolerance of writing to file system logs is guaranteed. To ensure that file system log blocks do not linger in the disk cache, cache flush calls and the Force option are used. Unit Access (`O_FUA`) . When using `journal=ordered` (others should not be used) first a write command is sent to the disk controller. Then all changed journal blocks are written to JBD2 . Then a cache flush command is sent to the disk controller - to write everything that is in the disk controller cache to ensure that the data blocks and journal blocks are saved and protected from loss during a power failure. Next, one commit block is written to JBD2 , indicating that the transaction has been successfully completed. This ensures that the transaction is atomic in the log. The commit block is written using the `O_FUA` , which can be set to write only one block (rather than multiple blocks) and is a lightweight operation since it does not perform a full cache flush. Since a transaction in JBD2 consists of at least 3 blocks, a cache flush is performed on each commit. If a transaction in the fastcommit log consists of a single block, then `O_FUA is used when writing to the fastcommit log` , rather than cache flush.

Fast commits of changes in the ext4 file system journal (fast_commit)

- If WALs are on a separate filesystem, then fast commit will not improve performance.
- `commit` parameter (default 5 seconds) sets the frequency of calling sync on the file system
- quick commits work in `data=ordered mode`
- `fsync` execution latency and `fdatasync` but only if the metadata has changed
- writing to the fastcommit log is done by the process that called `fsync`
- writing to the regular log is done by a kernel thread
- fast commit works not only with ext4 but also with other journaling file systems

Fast commits of changes in the ext4 file system journal (fast_commit)

ext4 file system records changes in metadata (what blocks contain what) in a JBD2 -type journal separately for each mounted file system. The write is performed according to transaction logic to avoid damage in case of power failure. The commit parameter (default 5 seconds) specifies the frequency of calling sync . `fsync` writes changes only to the file for which it is called.

By `fsync` flushes all dirty blocks and metadata for all file descriptors. This can create a lot of I/O that is not needed to save changes to a single file via `fsync` .

If you remove the directory `PGDATA/pg_wal` to a separate file system section where nothing but WAL files will be stored, this is not essential for WAL . This plays a role when files are frequently created and deleted. Temporary table files can be frequently created, deleted and resized, which is relevant for 1C applications.

Advantages of quick fixation:

•fast commit writes one 4K block, and committing changes to JBD2 stores at least 3 blocks (6 blocks on average) per commit . The fast commit log contains operations performed since the last standard commit, which is called by default once every 5 seconds. If fast commits cannot handle an operation, writing to JBD2 is used . Increasing the commit frequency may cause fsync will be executed more often by writing to JBD2. The commit parameter also correlates with the `vm.dirty_expire_centisecs parameters` and `vm.dirty_writeback_centisecs`

•Writing to JBD2 requires at least two execution context switches, since writing to JBD2 is done by a kernel thread. Writing to the fastcommit log is done by the process that called fsync . Using schedulers other than none under heavy CPU load can increase the difference.

fsync latency , the latency when using fast commit has less variance than JBD2 writes.

Fast commit works not only with ext4 , but also with file systems using other journal types (xfs , jfs).

Quick Fix Description:

https://www.usenix.org/system/files/atc24-shirwadkar.pdf

pg_test_fsync twice and pgbench -c 3 -t 5 0000 on virtual machines . The number of clients is 3, since there are 4 processors, with a larger number the bottleneck became would be a processor.

fast_commit enabled :

```
postgres@tantor :~$ pgbench -c 3 -t 5 0000
number of transactions actually processed: 15 0000/ 15 0000
latency average = 2.147 ms
initial connection time = 12.987 ms
tps = 1397.598698 (without initial connection time)
root@tantor :~# cat /proc/ fs /jbd2/sda1-8/info
895 transactions (867 requested), each up to 16320 blocks
average:
844ms running transaction ( number of milliseconds the transaction was running )
3447us average transaction commit time
90 handles per transaction ( number of filesystem transaction handles for journal transaction )
3 blocks per transaction ( number of filesystem blocks in the transaction )
  5 logged blocks per transaction (total number of blocks written to the journal for this transaction, including journal overhead)
root@tantor :~# cat /proc/ fs /ext4/sda1/ fc_info
fc stats:
fc stats:
180988 commits
731 ineligible
181026 numblks
596us avg_commit_time
```

fast_commit disabled :

```
postgres@tantor :~$ pgbench -c 3 -t 5 0000
number of transactions actually processed: 15 0000/ 15 0000
latency average = 2.125 ms
initial connection time = 12.188 ms
tps = 1411.806469 (without initial connection time)
root@tantor :~# cat /proc/ fs /jbd2/sda1-8/info
27446 transactions (27430 requested), each up to 16384 blocks
average:
12ms running transaction
4447us average transaction commit time
2 handles per transaction
1 block per transaction
  3 logged blocks per transaction
```

The data provided mostly relates to work pg_test_fsync .

150000 COMMITs called 150000 fdatasync () but they were processed by 22 commits in JBD2 both with fast_commit enabled and disabled , i.e. once every 5 seconds . This is not visible in the example, to measure this you need to reread info before starting pgbench and during pgbench operation , it will be visible that commits in JBD2 occur once every 5 seconds.

fast_commit enabled In JBD2, each transaction wrote slightly more blocks, and also wrote blocks (150,000 out of 181,026 numblks) to the fastcommit log on each COMMIT .

If you enable fsync instead of fdatasync , the difference in TPS will be greater: 970 (with fast_commit enabled) and 1300. In this case , there will be 590 and 28600 fixations in JBD2 .

This applies to logging. Data block writes to the WAL file are guaranteed to be written on every COMMIT.

Fast commits of changes in the ext4 file system journal (fast_commit)

- commit parameter (default 5 seconds) specifies how often sync is called on the file system.
- quick commits work in `data=ordered mode`
- Enable quick commits:

```
root@tantor :~# dumpe2fs /dev/sda1 | grep Fast
dumpe2fs 1.47.0 (5-Feb-2023)
Fast commit length: 0
root@tantor :~# tune2fs -O fast_commit /dev/sda1
tune2fs 1.47.0 (5-Feb-2023)
root@tantor :~# dumpe2fs /dev/sda1 | grep Fast
dumpe2fs 1.47.0 (5-Feb-2023)
Fast commit length: 256
```


Fast commits of changes in the ext4 file system journal (fast_commit)

To support fast commits, a fast commit log is needed for operations.
Quick commits work in `data=ordered mode` .
Check if the quick commit log is available:

```
root@tantor :~# dumpe2fs /dev/ sda1 | grep Fast
Fast commit length: 0
```

Zero means not available and fast commits are not used. To enable logging:

```
root@tantor :~# tune2fs -O + fast_commit /dev/sda1
root@tantor :~# dumpe2fs /dev/ sda1 | grep Fast
Fast commit length: 256
```

A non-zero value means that the fast commit log is available on this file system. The file system must be remounted for fast commits to be used. You can verify that they are in use by looking **at the fast commit statistics:** `cat /proc/ fs /ext4/ sda1 / fc_info`

Also check if the fast_commit log is available you can use the command:

```
root@tantor :~# tune2fs -l /dev/ sda1 | grep fast_commit
Filesystem features: has_journal ext_attr resize_inode dir_index
fast_commit filetype needs_recovery extent 64bit flex_bg sparse_super
large_file huge_file dir_nlink extra_isize metadata_csum
```

Disconnection fast_commit :

```
root@tantor :~# tune2fs -O ^ fast_commit /dev/ sda1
```

Why by default fast commit log not created ? For normal desktop use where fsync `fast_commit` is not called often does not provide any performance gain. Writing to JBD2 occurs regardless of the presence of fast commit and the volume of writing to JBD2 does not decrease.

Astralinux 1.8.1 has version 1.47.0 installed .

`fast_commit` on and off supported since version 1.46 (2021):

" E2fsprogs now supports the fast_commit (COMPAT_FAST_COMMIT) feature. This feature, first available in Linux version 5.10, adds a fine-grained journaling which improves the latency of the fsync (2) system call. It should also improve the performance of ext4 file systems exported via NFS." https://e2fsprogs.sourceforge.net/e2fsprogs-release.html

Utility `pg_test_fsync`

- a command line utility that performs standard tests to evaluate performance when choosing a value for the `wal_sync_method parameter`

```
postgres@tantor :~$ 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 3396.549 ops/sec 294 usecs /op
        fdatasync 3459.610 ops/sec 289 usecs /op
        fsync 3136.642 ops/sec 319 usecs /op
        fsync_writethrough n/a
        open_sync 3275.082 ops/sec 305 usecs /op
Compare file sync methods using two 8kB writes:
```


`pg_test_fsync` utility

A command line utility that runs tests to evaluate the performance when choosing a value for the `wal_sync_method parameter` . The utility writes one or two 8K blocks to a test file, just as the process that saves a transaction commit record to a WAL file does.

The utility gives the number of block writes per second, which can serve as an estimate for the maximum TPS (transactions per second) for small transactions if the bottleneck is writing to WAL files.

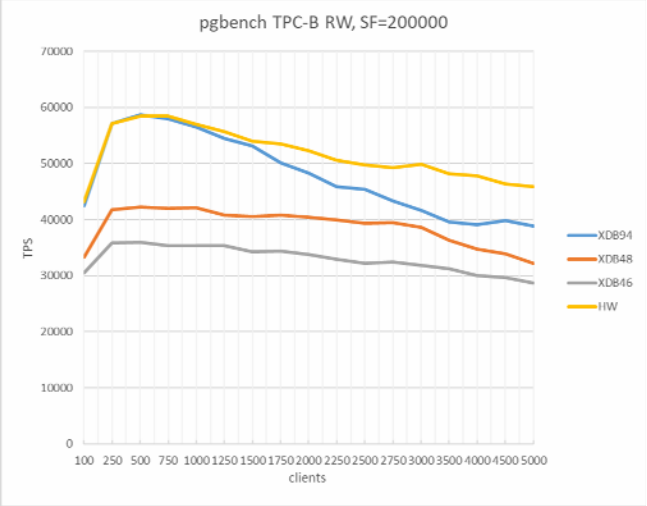
By default, each method is tested for 5 seconds.

pg_test_fsync prints the average time of a file system synchronization operation for each `wal_sync_method` , which can be useful when choosing a value for the parameter `commit_delay` if you plan to use `commit_siblings` . These two parameters can increase TPS if the instance has a large number of concurrent committing transactions and the bottleneck is writing to WAL files.

Support `for O_DIRECT` means that the operating system supports direct I/O and can (but does not need to) use the `debug_io_direct option` .

Group commit transactions

- transactions can be recorded in groups
- When processing a COMMIT , it is not necessary that every server process writes to the WAL file.
- the journal subsystem is usually not a bottleneck, but it guarantees fault tolerance
- the algorithm for working with WAL ensures scalability : with an increase in the number of sessions to ~500, TPS grows




```
HW - physical server Tantor xData 96CPU, 1536GB RAM, 18TB SSD
shared_buffers =384GB

Tantor XDataBox container
94CPU, 1120GB RAM, LVM 5TB, shared_buffers=384GB
48CPU, 560GB RAM, LVM 5TB, shared_buffers =192GB
46CPU, 560GB RAM, LVM 5TB, shared_buffers =192GB
```


Group commit transactions

If writing the log buffers to disk takes time, and during this time other server processes execute COMMIT , then it cannot be said that flushing the buffers will be performed by each of these processes. The largest delay is usually caused by the execution of the call that guarantees writing the buffers to disk (flush cache) - by default `fdatasync` . This is achieved by the algorithm of the processes working with the log buffer and the logic of obtaining locks.

When a process wants to flush its buffers, it must acquire a `WALWriteLock lock` . If the process acquires it immediately, it becomes the " leader " and begins flushing all buffers accumulated in the log buffer, and then gives `fdatasync` (unless another guaranteed write method is used). Other processes that have generated a record of their `COMMIT` and those ready to flush buffers to disk also queue up for the `WALWriteLock lock` . When the lock becomes available, none of the processes in the queue immediately obtains the lock that has become available. Instead, each process checks again whether the buffers have been flushed before and including the journal entry with `COMMIT` their transactions. Often processes detect that the log records have been flushed by the " leader " . In this case, the processes refuse to acquire the lock, write to the WAL segment and confirm to the client that the transaction is committed. With significant delays in the execution `of fdatasync` and high transaction frequency, a small portion of processes (the " leaders ") will flush the buffers. It turns out that transaction commits are processed by a group (batch) . Due to this algorithm, the journal subsystem is usually not a bottleneck.

https://pgeoghegan.blogspot.com/2012/06/towards-14000-write-transactions-on-my.html

https://rutube.ru/video/private/e9a69100951dd2865db96ec49293423c/?p=G5067rhW71F2GeS7U3cJ7Q

commit_delay parameters and commit_siblings

- This configuration parameter sets the maximum delay in microseconds that the server process will wait after creating a log entry with COMMIT. in the WAL buffer before calling the write command to the WAL file if there is commit_siblings open transactions and fsync =on
- the default value is zero, there is no delay
- does not speed up writing to WAL
- can be used to limit TPS so that a bottleneck does not occur in a subsystem other than WAL
- commit_delay value to start with is half the delay reported by pg_test_fsync for the wal_sync_method being used
- change values of commit_delay parameters and commit_siblings does not require instance restart

commit_delay parameters and commit_siblings

commit_delay parameter sets the maximum delay in microseconds that the server process will wait after creating a log record with COMMIT. in the WAL buffer before calling the write command to the WAL file. The default is zero - no delay. The server process will wait if: after the journal entry is generated, the instance has at least commit_siblings open transactions, fsync =on . Moreover, the first server process that is ready to send a command to write (to flush accumulated blocks of the log cache to disk) waits for commit_delay , the processes following it that will form a log record with COMMIT wait only until the first waiting process flushes its log record, their log records (which were created in the buffer while the first process was waiting) and everything that has accumulated in the buffer by the end of the wait. This logic has appeared since version 9.3 of PostgreSQL , the parameter itself appeared in version 7.1 when the flush to disk methods worked inefficiently.

Because fdatasync works fast enough, has a group transaction commit algorithm, commit_delay does not speed up writing to the WAL file. This option can be used if a large number of transactions per second is stressing the instance so that a bottleneck occurs in a system other than the log file that is difficult to resolve other than by reducing the number of transactions per second (TPS) by inserting a small delay into the COMMIT command . The application is expected to encounter a decrease in TPS , will not increase the number of sessions. The heavy load usually occurs not with 5 sessions, but on strands more, therefore the value of commit_siblings should be large enough. An analogy can be drawn: with laminar flow of liquid the throughput of the pipe is high, with turbulent it decreases and causes vibration of the pipe and it is necessary to ensure that the liquid pressure is limited when the flow becomes turbulent somewhere usually in the narrowest place.

a commit_delay value of half the delay reported by pg_test_fsync for the wal_sync_method being used (usually fdatasync).

Typical range of values commit_delay 20 0-1000. For commit_siblings ~500.

Changing the values of the commit_delay parameters and commit_siblings does not require an instance restart. Changing the commit_delay value requires the modify privilege or the superuser attribute , commit_siblings can be changed by any user.

https://postgresqlco.nf/doc/en/param/commit_delay/

https://pgpedia.info/c/commit_delay.html

I/O bus commands discard `/trim`

- important for SSD
- `trim` - for SSD on SATA bus
- `discard` - for SSD on pci bus
- SSD controller does not know that there are files on it and that there are file systems
- the operating system must send commands to clear blocks (physical sectors) whose contents it does not need
- write resource in SLC mode ~ 100000 cycles
- non-SLC write resource ~1000 cycles

I/O bus commands discard `/trim`

When using SSD on any bus - PCIe (NVMe) or SATA, the operating system must inform the device controller that the contents of physical blocks are not needed by the operating system. The SSD controller does not know that there are files and file systems on it. The operating system must send the trim (for the SATA bus) or discard (for the PCI bus) command to clear blocks (physical sectors) .

After disk initialization and partition creation, data about which blocks are used is stored in the file system. It is critical for SSD to receive data from the operating system that some blocks are not used by the file system and can be cleared.

For example, SSD was initialized, partition created, formatted and mounted. SSD device FTL table stores only a few records that several blocks are used by the operating system. The partition starts to fill with files. SSD controller writes them to "SLC cache " (high-speed physical layer with resource of about 100000 write operations) . Maximum SLC size depends on memory chips. Currently common TLC 3 bits (SLC maximum 1/3 minus reserve blocks, in practice less , since it depends on the algorithm by which the controller uses SLC), for cheap QLC 4 bits (SLC maximum 1/4 of the size minus reserve blocks).

While the SLC is filling , the controller (depending on the controller's operating algorithm) can move blocks from the SLC to other layers, writing to which is several times slower, but the most important thing is that the number of records to such layers is small - about 1000. Also, if the SLC cache is full, the controller can write received blocks to other layers at a speed several times slower. Files can be deleted in the file system and space can be freed. The controller will not know about this, since the mark that the file system has stopped using such and such blocks is stored only in its metadata. In order for the SSD to know that the blocks occupied by deleted files are no longer needed, the operating system must, when deleting a file in the file system, send the SSD controller the discard or trim command , specifying the range of blocks that are no longer needed (usually those occupied by the file, if the file system did not combine several files into one block to save space). Whether Linux will send such a command is set in the file system mount properties.

You should not rely on the fact that when initializing an SSD (creating a GPT/MBR partition) or when creating a file system with the mkfs.ext4 command, a discard will be performed on the entire range of blocks. If an SSD was used, it is worth initiating a discard separately .

Support `discard/trim`

- if the controller does not have free blocks, the write speed and lifespan of the SSD will be low
- SSD controller usually clears in blocks of `erase size`

```
root@tantor:~# tune2fs -o +discard /dev/sda1
root@tantor:~# tune2fs -l /dev/sda1 | grep discard
Default mount options: user_xattr acl discard
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
```


`Discard/trim` support

To clean up a previously used SSD, you can use the `blkdiscard utility` .
To clean up blocks not used by a mounted file system, you can use the `fstrim utility` .
SSD controller cleans in blocks of size `erase size` . If the range of blocks does not fit into the boundaries `of erase size` , then `discard` for physical blocks falling on part of the block, the erase size is usually ignored (the behavior of the SSD is completely determined by the controller firmware) and such blocks are not cleared. The `erase size` there could be tens of megabytes.
In the description (issued by the `man command`) `fstrim` utilities It is written that for most desktops and servers it is enough to run it once a week.
The file system can be mounted with the discard option and when a file is deleted from the file system, the `discard/trim command will be sent` . The time it takes for the controller to execute the `trim command` (unless it is given across the entire SSD) may be proportional to the number of blocks. If the file size is large, the controller may suspend receiving commands from the operating system for some time. PostgreSQL does not use large files. In Tantor DBMS, the maximum size of data and log files is 1 GB. However, linux can also serve other applications, for them they came up with a logic according to which the file system is mounted without `discard` , while cleaning is performed by `the fstrim service` . It is better to enable `dicsard` in the `ext4 mount parameters` .
discard property can be set in the superblock ext4 as default mount option: **`tune2fs -o +discard /dev/sda1`** or in `/etc/ fstab` :
`/dev/sda1/ext4 rw,discard 0 0`
For LVM in `/etc/lvm/lvm.conf` : **`devices`** `{ issue_discards = 1 }`
`For` encrypted partitions in `/boot/grub/grub.cfg`
check whether discards were sent to the SSD controller using the following commands:

```
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
```

Zeros in `DISC-GRAN` And `DISC-MAX` means that `discard` not used.

Recommendations for using SSD

- It is not recommended to actively write to the SSD if the percentage of filling more than ~ 80% (depends on SSD controller algorithm)
- In order to always have a part of the SSD free, you can create a partition smaller than the SSD
- status of the service performing discard(trim) on unused blocks on file systems using discard(trim) -capable devices :

```
root@tantor:~# systemctl status fstrim.timer
fstrim.timer - Discard unused blocks once a week
Loaded: loaded (/lib/systemd/system/fstrim.timer; enabled; preset: enabled)
Active: active (waiting) since ..; 7h ago
Trigger: 1 day 6 hours left
Triggers: fstrim.service
Docs: man:fstrim
systemd[1]: Started fstrim.timer - Discard unused blocks once a week.
```


Recommendations for using SSD

When working with SSDs, the `nvme` utility is useful . It is installed in the `nvme-cli package` , which depends on the uuid-runtime package:

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

The number of blocks that can be cleared at the same time may be limited. You can see the limits:

```
/sys/block/ sda /queue/ discard_max_hw_bytes
```

"Many experts recommend limiting SSD usage to only 80% of its total capacity":
https://www.seagate.com/nl/nl/blog/what-are-ssd-trim-and-garbage-collection/
In order to always have a part of the SSD free, you can create a partition smaller than the SSD.
It is recommended to periodically (once a week) clean up unused blocks, this is done by the fstrim service .
https://wiki.archlinux.org/title/Solid_state_drive/Memory_cell_clearing
Under Windows operating system , cleaning can be performed manually using the command in an elevated command prompt:

```
powershell -command optimize-volume - driveletter C - retrim -verbose
```

Disabling the JBD2 journal in the ext file system without losing data is possible with the command:

```
tune2fs -O ^ has_journal /dev/sda1
tune2fs 1.47.0 (5-Feb-2023)
The has_journal feature may only be cleared when the filesystem is
unmounted or mounted read-only.
```

Enabling JBD2 Logging :

```
tune2fs -O + has_journal /dev/sda1
```

https://wiki.astralinux.ru/pages/viewpage.action?pageId=48759308

Parameter max_files_per_process

- default value 1000
- when the value is reached, the server process will frequently close and open files
- a large number of files may be opened if the commands work with a large number of relations or tables consisting of a large number of gigabyte files

```
select setting, min_val,max_val,vartype , short_desc from pg_settings where
name = ' max_files_per_process ';
 setting | min_val | max_val | vartype | short_desc
---------+---------+----------+---------+----------------------------
1000 | 64 | 2147483647 | integer | Sets the maximum number of
simultaneously open files for each server process.
```


Parameter max_files_per_process

At working with a large number of objects, for example, a server process can open many files. Each table has three layers plus a TOAST table and index. With a large amount of data, the main layer can consist of a large number of gigabyte files (up to 32K). The cluster parameter `max_files_per_process` limits the number of files that can be opened by each server process. Shared libraries are not taken into account. The default value is 1000. If the server process must work with a large number of files for a short time, then you can increase the value of the parameter, otherwise the server process will be forced to close and open files frequently. A large number of files can be opened in sessions servicing 1C applications.

The value 1000 is set based on the fact that older versions of operating systems set a limit on the number of open files in `/etc/security/limits.conf` (`nofile parameter`) or in `/etc/systemd/*.conf` (`LimitNOFILE` parameters , `LimitNOFILESoft` , `DefaultLimitNOFILE`) . **Setting these files to values other than infinity causes the kernel to collect statistics on open files, which does not speed up its operation.**

```
root@tantor:~# cat /proc/sys/fs/file-nr
6336 0 9223372036854775807
```

the first number is the total number of open files in linux , the second is the number of open but not currently used files, the third is the maximum possible number of open files.

The maximum value of the `max_files_per_process` parameter is:

```
postgres =# select setting, min_val,max_val,vartype , short_desc
from pg_settings where name = ' max_files_per_process ';
setting | min_val | max_val | vartype | short_desc
---------+---------+----------+---------+----------------------------
1000 | 64 | 2147483647 | integer | Sets the maximum number of
simultaneously open files for each server process.
```

View the number of files opened by the process (PID=854) :

```
root@tantor :~# lsof -p 8 52 | wc -l
66
```

Increase `max_files_per_process` values

- When increasing the value of this parameter, you need to make sure that the operating system does not limit the number of open files:

```
ERROR: could not open relation 5/16550: Too many open files in system
```

- real limitations of running processes:

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

- restrictions on processes launched not through services:

```
sudo -u postgres bash -c ' ulimit -n'
1024
```

- in the file /etc/security/limits.conf add lines:

```
postgres hard nofile infinity
postgres soft nofile infinity
```

- in services files / usr /lib/ systemd /system/ tantor * add after [Service]

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


Increase `max_files_per_process` values

The default value is `max_files_per_process =1000` . When increasing the value of this parameter, you must ensure that the operating system does not limit the number of open files. Example of errors:

```
ERROR: could not open relation 5 / 16550 : Too many open files in system
```

Check the real limits of already running processes named postgres:

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

In the example, the soft `limit is 1024, the hard limit is 524288.` When exceeding `s oft limit` the process will receive a warning. When exceeding `hard limit` The process will not be able to open new files until previously opened files are closed.

`soft limit` for one process of the postgres user you can see it with the command:

```
sudo -u postgres bash -c ' ulimit -n'
1024
```

With the parameter `-n (maximum number of open file descriptors)` you can use parameters - S (soft) or - H (hard) :

```
sudo -u postgres bash -c ' ulimit - Hn '
1048576
```

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

```
postgres hard nofile infinity
postgres soft nofile infinity
```

On those launched via `systemd` instances this will not work. You need to edit the file `/ usr /lib/ systemd /system/tantor-se-server-16.service ,` adding after `[Service]`

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

Setting a value other than infinity forces the kernel to collect statistics on open files, which does not speed up its operation.

after editing update systemd configuration and restart the instance:

```
systemctl daemon reload
systemctl restart tantor-se-server-16.service
```

Alternatively, you can edit with the command `systemctl edit tantor-se-server-16.service ,` paying attention to which file will be edited.

Temporary file system (tmpfs)

- a file system that uses RAM to store files
- Tantor DBMS does not use `tmpfs` for storing files
- don't use `tmpfs`

```
root@tantor :~# mount | grep tmp
udev on /dev type devtmpfs
tmpfs on /run type tmpfs
tmpfs on /dev/ shm type tmpfs
tmpfs on /run/lock type tmpfs
ramfs on /run/credentials/ systemd - tmpfiles -setup- dev.service type
ramfs
ramfs on /run/credentials/ systemd-tmpfiles-setup.service type ramfs
tmpfs on /run/user/102 type tmpfs
tmpfs on /run/user/1000 type tmpfs
```


Temporary file system (tmpfs)

`tmpfs` (temporary file system , temporary file system), previously known as `shmfs` . There is an analogue `of ramfs` . These are file systems that use RAM to store files. After rebooting the operating system, the files in them disappear. If the operating system uses swap , then `tmpfs` blocks from memory can be pushed out to swap partitions or files .

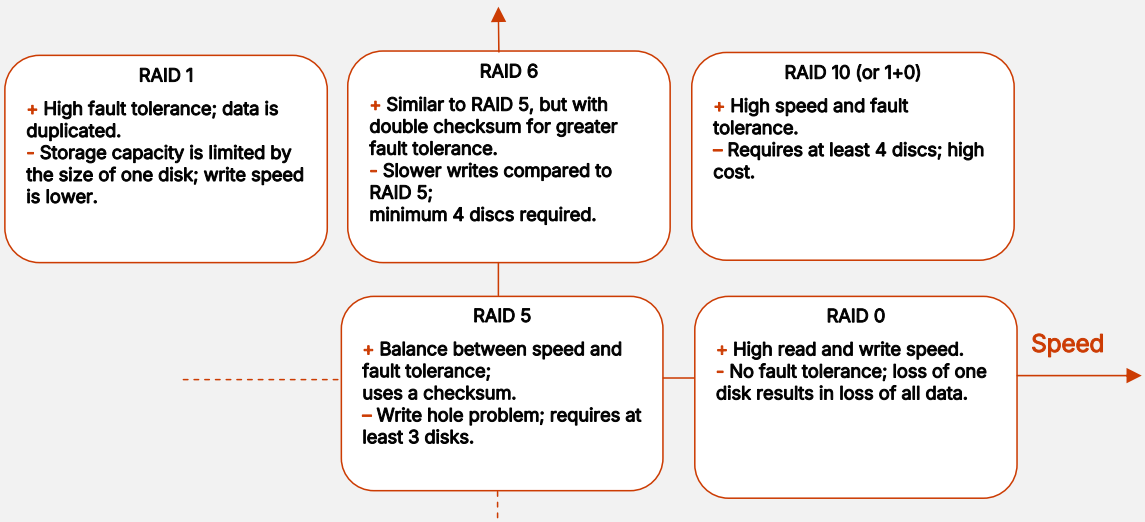
Tantor DBMS does not use `tmpfs` for storing files and do not need to store them in it. In the PostgreSQL DBMS up to version 15, statistics were collected in the directory specified by the `pg_stat_tmp parameter` , by default `PGDATA/ pg_stat_tmp` . The recording was so active that it was necessary to mount this directory in `tmpfs` . In version 15, this unfortunate functionality was changed, but recommendations were written and only confuse. Files related to temporary tables and indexes on temporary tables can be created in table spaces. These files are created in subdirectories named `pgsql_tmp` . For example, `PGDATA/ base / pgsql_tmp` is consonant with `pg_stat_tmp` and the administrator's memory may pop up with a recommendation to mount something using the `tmpfs file system` . The table is temporary, but the files are permanent. These files cannot be stored on `tmpfs and it makes no sense at least because the table can grow to 32 TB and either there will not be enough space (if you limit the size of the tmpfs` file system when mounting), or there will not be enough RAM and the swap partition (or file) will be used , or oom kill will start stopping processes with `the SIGKILL signal .`

Old versions of linux operating systems could mount temporary directories as tmpfs (/ tmp) . This is not used now and is not recommended to use, if only because the speed of writing to RAM of large volumes without using Huge Pages is usually slower than writing to modern NVMe (up to ~ 8 gigabytes per second on a PCI 4x4 bus).

In modern versions of Linux the file system is tmpfs used to store some virtual service directories (`/run` , `/dev/ shm`) with small files, or functionality whose code was written a long time ago, works and there is no point in rewriting.

RAID

- array of " disks " (storage devices)
- used as a volume (single block device)



RAID

Term RAID was born in 1987 as an abbreviation for Redundant Array of Inexpensive Disks (an array of inexpensive redundant disks). It was proposed to use a set of inexpensive unreliable disks, instead of large-capacity disks " SLED" (Single Large Expensive Drive). Later Inexpensive was replaced by Independent , as they started using expensive disks . In types (" levels ") :

RAID 0 (stripe) - stripes are distributed across all disks, no fault tolerance

RAID 1 (mirror) - duplication (mirroring) . High overhead .

RAID 2 , 3, 4 are not used in practice;

RAID 5 (stripe with parity) - stripe with parity data is distributed across all disks, disks are equal. It has become widespread due to small overhead costs: a mirror has costs of half the disks versus one disk. Protects against the loss of one disk. Disadvantages: the speed of writing in random order (Random Write) is 10-25% lower when compared to RAID 0 (stripe alternation without parity control) , since each write operation is replaced by two read operations and two write operations. When replacing a failed disk, initialization of a new disk is accompanied by a heavy load on the existing ones, which can lead to the failure of other disks and complete loss of the array without the possibility of recovery. It is not recommended to use with a DBMS.

RAID 6 (double parity) - resistant to failure of two disks. Disadvantages: recording speed up to ~ 2 times lower than RAID 5; high overhead costs. Minimum 4 disks.

RAID 10 (1+0) mirrored pairs of disks. Overhead as a mirror - half the size of the disks. Minimum 4 disks.

RAID 50 (5+0) minimum number of disks 6. Stripes with parity are duplicated. Overhead: two disks.

RAID support is available in linux at the kernel level. Linux supports software RAID levels 0, 1, 4, 5, 6, 1+0. Using the volume manager, you can combine levels to get 5+0. You can manage RAID devices in Linux using the mdadm utility . Software RAID uses CPU resources. CPUs are usually not a bottleneck for DBMS.

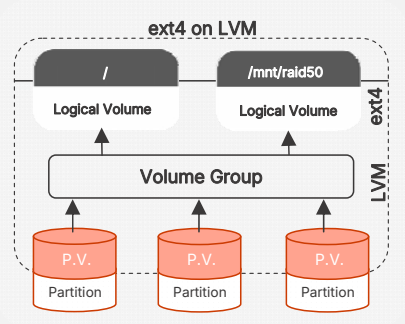
Oracle ASM (the volume manager and cluster file system for storing Oracle Database files) uses a software (not Linux kernel level) RAID (double, triple stripe duplication , or no duplication).

You can also use hardware mirroring and RAID (Intel Matrix).

LVM

- Logical Volume Manager
- combines physical disks into groups - Volume Groups (VG) on which logical partitions can be created Logical Volumes (LV)
- You can resize LVs and add new disks to VGs without moving data or changing file systems
- LVM does not increase performance, it adds an additional level of abstraction, which increases the likelihood of errors when reconfiguring
- used when creating RAID 5+0 :

```
mdadm --create /dev/md0 --level=5 --raid-devices=2 /dev/sd1 /dev/sd2
mdadm --create /dev/md1 --level=5 --raid-devices=2 /dev/sd3 /dev/sd4
pvcreate /dev/md0
pvcreate /dev/md1
vgcreate vg0 /dev/md0 /dev/md1
lvcreate -L 100%FREE -n lv0 vg0
mkfs.ext4 /dev/vg0/lv0
mkdir / mnt /raid50
mount /dev/vg0/lv0 / mnt /raid50
```



LVM

Logical Volume Manager (Logical Volume Manager). Allows at the operating system level (" logically ") combine physical storage devices (" disks ") into groups - Volume Groups (VG) on which logical partitions can be created Logical Volumes (LV) - Logical partitions can be used in the same way as regular hard disk partitions. The advantage is that you can resize LVs and add new disks to a VG without moving data or changing file systems.

LVM does not improve performance and adds an extra layer of abstraction, which increases the likelihood of errors when reconfiguring.

LVM is used when there are no hardware storage arrays and separate devices must be used.

LVM is also used to create RAID 5+0. Example:

1) Partitions are created on disks , block devices of the type `/dev/ sdN appear`

2) Creating two R AID 5:

```
mdadm --create /dev/md0 --level=5 --raid-devices=2 /dev/sd1 /dev/sd2
mdadm --create /dev/md1 --level=5 --raid-devices=2 /dev/sd3 /dev/sd4
```

3) Combining RAID 5 into physical volumes (PV, Physical Volume):

```
pvcreate /dev/md0
pvcreate /dev/md1
```

4) Unification PV in VG:

```
vgcreate vg0 /dev/md0 /dev/md1
```

5) Creating a logical volumes (LV):

```
lvcreate -L 100%FREE -n lv0 vg0
```

6) Format and mount the logical volume :

```
mkfs.ext4 /dev/vg0/lv0
mkdir / mnt /raid50
mount /dev/vg0/lv0 / mnt /raid50
```

For rotational disks (HDD) combined into RAID, you can set the `effective_io_concurrency parameter` in the number of disks across which data is distributed (striping) . The parameter enables or disables (the value is zero) the preliminary fetching of table blocks in the Bitmap Heap Scan index access method . In addition to the usual levels, the parameter can be set in the tablespace properties, like the `maintenance_io_concurrency parameter .`

Practice

- Part 1. Disk subsystem parameters
- Part 2. Installing packages in Astralinux
- Part 3. Working with SSD and testing disk performance with the `fio utility`
- Part 4. Testing the ext4 fastcommit journal
- Part 5. Removing the limit on the number of open files
- Part 6. Example of command files for testing
- Part 7. Example of creating programs for testing

Practice

In the practice, you will compare the performance of SATA and NVME interfaces of a virtual machine on the same physical device and verify that using the NVME interface is 10% faster than SATA .

Find out that NVME devices are formatted by default not in ext4, but in ext2 , and find out whether ext2 provides any advantages.

You will learn how to enable the fast commit log and what parameters to use to evaluate file system performance.

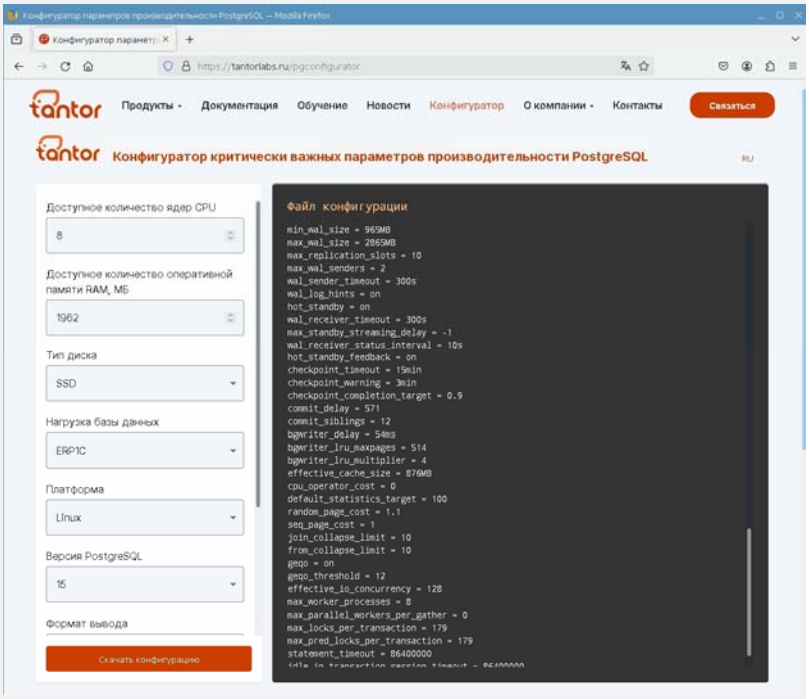
Changes to the limits.conf file to remove the limit on the number of open files do not affect services. You will learn the procedure for removing the limit and commands for checking the actual limits.

5

Initial setup of DBMS

Configurators

- pg_configurator utility
- web version http://tantorlabs.ru/pgconfigurator
- are introduced host characteristics and planned load
- gives configuration parameters



Configurators

The database cluster is created by the initdb utility . The utility creates the postgresql.conf file with default values. These values are designed to service a not very loaded application so that the DBMS can be used on the desktop by an ordinary developer. In the Tantor DBMS, the initdb utility does not change the parameter values compared to PostgreSQL initdb . It is assumed that the parameters for production use will be configured separately.

For initial configuration, you can use the pg_configurator utility created by and supported by Tantor Labs . The utility is available on the website https://tantorlabs.ru/pgconfigurator/ shell in the form of a command line utility https://github.com/TantorLabs/pg_configurator

The utility accepts 7 or ~ 20 parameters and makes recommendations based on them.

There are not many initial configuration utilities. Of the known ones:

1. PGconfigurator www.cybertec-postgresql.com , web version pgconfigurator.cybertec.at makes recommendations based on 13 parameters

2. PG With onfig https://github.com/pgconfig/api, web version www.pgconfig.org gives recommendations based on 8 parameters

3. PGTune github.com/le0pard/pgtune , created by 2ndQuadrant employee , web version pgtune.leopard.in.ua gives recommendations based on 7 parameters

During the operation of the DBMS, the Tantor Platform configurator can recommend configuration parameters. The Platform configurator makes recommendations based on ~25 parameters.

https://tantorlabs.ru/pgconfigurator

Setting up PostgreSQL to work with 1C products :

https://wiki.astralinux.ru/tandocs/nastrojka-postgresql-tantor-dlya-raboty-1s-294394904.html

Next, we consider the parameters whose values are set first. The configurator provides initial values. It is important to understand the meaning of the parameters and what they affect.

Parameters `shared_buffers , temp_buffers , effective_cache_size`

- `shared_buffers` Default is 128MB . Can be set to ~ 1/4 of physical memory size. After changing the value, you need to restart the instance
- `effective_cache_size` by default 4 GB
  - gives the planner an estimate of the size memory that can accommodate table blocks
  - affects the calculation of the I/O cost, which is part of the overall cost of the execution plan
  - with a small value of the parameter, the cost of input/output (cost) increases for any blocks - both tables and indexes, and plans will be selected in which fewer blocks are read
  - can be set to 70-80% of the physical memory size

Parameters `shared_buffers , temp_buffers , effective_cache_size`

When initially setting parameter values, it is assumed that the values of parameters that will not be mentioned are set to default values. If the parameter values differ from the default values, then check whether this reduces fault tolerance. For example, it is worth checking whether the checksum calculation is enabled. If not, then enable it. Check no whether there are long-unused initialized replication slots :

```
select slot_name , pg_current_wal_lsn ()- restart_lsn from pg_replication_slots
;
```

Parameters whose values should be set depending on host parameters and load:

shared_buffers Default is 128MB . Can be set to ~ 1/4 of physical memory size. After changing the value, you need to restart the instance

`effective_cache_size` Default is 4 GB. Gives the scheduler a size estimate. memory that can accommodate table blocks (the operating system cache and possibly the buffer cache if the blocks are not duplicated in them) . Can be set to 70-80% of the physical memory size. The parameter does not affect memory allocation, only the scheduler estimates. The parameter can be set at any level (sessions, transactions, functions, and others), so the values set may not correspond to the memory size, but may be used to influence the scheduler's choice of access methods. The cost of the execution plan consists of two parts: processor and input - output. The parameter affects the calculation of the cost of input-output. **With a small value of this parameter, the cost estimate of the input-output increases** for any blocks: both tables and indexes. **Plans will be selected in which fewer blocks are read** .

`temp_buffer s` specifies the size of the local (in the server process memory) buffer cache, which is used when working with temporary tables. By default, 8 MB. The value can be changed in a session, but only before the first use of temporary tables, after which the value does not change until the end of the session. Memory is allocated gradually as needed, not immediately. In addition to the memory for storing temporary table blocks, **an additional 64 bytes are immediately allocated in the server process memory for each buffer to store buffer descriptors** . If the application actively uses temporary tables, the parameter value can be increased. For 1C applications, it is increased to 256 MB.

Parameters `work_mem` , `hash_mem_multiplier` , `maintenance_work_mem`

- `work_mem` default 4MB
  - Together with the `hash_mem_multiplier parameter` , it affects the memory allocated by each server and parallel process.
  - The initial value is set based on an estimate of the number of sessions executing commands that process large data sets in memory, i.e. it depends on the type of OLTP (single-row inserts and selects) and non- OLTP workload.
- `maintenance_work_mem` default value 64MB
  - Specifies the amount of memory allocated by each process (server, parallel) participating in the execution of `the VACUUM , ANALYZE, CREATE INDEX , ALTER TABLE ADD FOREIGN KEY commands`
  - Initial value ~0.5% of physical memory size

Parameters `work_mem` , `hash_mem_multiplier` , `maintenance_work_mem`

`work_mem` parameter is 4MB by default. Together with the `hash_mem_multiplier parameter` , it affects the memory allocated by each server and parallel process. For example, when joining tables using hashing (Hash Join), the amount of memory allocated to service the JOIN will be `work_mem * hash_mem_multiplier *(Workers + 1)` . The parameter `work_mem` specifies how much memory a process can use to execute one step in an execution plan, rather than the memory for the entire command.

Besides `work_mem` the server process allocates memory for processing single row, but usually table rows are not very long. Therefore, the initial value is set depending on the estimated number of sessions in which commands are executed that process large data sets in memory. That is, the value depends on the type of OLTP (single-row inserts and selections) and non- OLTP workload. If we assume that `shared_buffers` will be set to ~1/4 of the physical memory size, then ~ 3 /4 will be taken up by the operating system cache , clean cache pages can be freed quickly, then memory is usually sufficient.

Rough estimate of memory that the instance will occupy: shared_buffers+2* work_mem * processes .

`maintenance_work_mem` default value is 64MB. Specifies the amount of memory allocated by each process (server, parallel) participating in the execution of `the VACUUM , ANALYZE, CREATE INDEX , ALTER TABLE ADD FOREIGN KEY commands` .

The number of parallel processes is limited by the `max_parallel_maintenance_workers parameter` . Index creation and normal (without FULL) vacuum . When vacuuming only the index processing phase (other phases are not parallelized), one index can be processed by one (rather than several) parallel processes. Whether parallel processes are used depends on the size of the indexes and the configuration parameters. Speed and load generated by commands servicing of database objects ("maintenance") , strongly depends on the allocated memory, in such cases the value is set before executing the command at the session level, but this is not the initial setting, but the setting during operation. The initial value at the cluster level depends on the amount of available (clean pages of the page cache) physical memory, for example, 0.5% of the size of physical memory. It is allocated immediately at the beginning of the command execution. Combining VACUUM and ANALYZE in a single VACUUM (ANALYZE) command will not provide any advantage, since in current versions of PostgreSQL vacuum and analyze are performed independently of each other.

autovacuum_work_mem parameter

- default -1 (uses maintenance_work_mem value)
- is allocated immediately by each working process of the autovacuum
- autovacuum processes do not use parallel processes
- to store tid (tuple identifier), regardless of the value of the maintenance_work_mem parameter

autovacuum_work_mem parameter

autovacuum_work_mem parameter defaults to -1 (i.e. equals maintenance_work_mem). It is allocated **immediately by each** autovacuum worker process. The number of autovacuum worker processes is limited by the autovacuum_max_workers parameter , they are not considered parallel processes (i.e. the max_worker_processes , max_parallel_maintenance_workers parameters they are not affected and do not exhaust the pool of parallel processes). Autovacuum processes **do not** use parallel processes , unlike the VACUUM command (without FULL) , each non-partitioned table with its indexes is processed by only one autovacuum worker process at a time . Memory is used to store row identifiers (tid) that have gone beyond the database visibility horizon from table blocks that may contain outdated row versions, i.e. are not in the visibility map . The amount of memory for such tid depends not on the table size, but on the frequency of vacuuming (the more often it is performed, the fewer blocks are likely to be scanned in the current autovacuum cycle), on the number of row updates or deletions (depends on how the application works with rows), and whether in-page cleaning is effective. If the autovacuum_work_mem memory into which tid are written not enough to record all tid from the table blocks that need to be cleared, then there will be several phases of clearing all indexes in the table and the efficiency of vacuuming will decrease: the duration will increase, but what is worse, the load on resources will increase due to repeated scans and clearing of index blocks.

Parameter value can only be set at the cluster level (it cannot be set at the database or table level) , to change the value it is enough to reread the configuration.

The main feature is that **under storage tid no more than 1GB is used regardless of the parameter values** . Therefore, autovacuum_work_mem should be less than 1GB. The value to set depends on autovacuum_max_workers and free physical memory. For example, if the number of processor cores is 4 and the memory is 32GB, then autovacuum_max_workers = 4 (not more than the number of cores) and autovacuum_work_mem=1 GB .

In version 17 for storing tid instead of a list, an adaptive radix tree is used, and the number of index rescans does not depend linearly on autovacuum_work_mem .

https://pganalyze.com/blog/5mins-postgres-17-faster-vacuum-adaptive-radix-trees

Parameters `temp_file_limit` And `temp_tablespaces`

- `temp_file_limit` sets the maximum amount of disk space that a single process can use for temporary files
- `temp_tablespaces` defines the names of tablespaces in which files will be created temporary objects

Parameters `temp_file_limit` And `temp_tablespaces`

`temp_tablespaces` sets the names of the tablespaces in which files will be created temporary tables and indices on temporary tables ; files created when executing sorts, joining data sets, creating indexes in SQL commands, if the process runs out of `memory` (`work_mem` and `maintenance_work_mem`) . By default, the tablespace list is empty and the default tablespace of the database the instance process is running is used.

The lifespan of temporary files is short - until the end of the command, transaction, truncation or deletion of the temporary table. The operating system uses regular files.

If multiple tablespaces are specified in the `temp_tablespaces parameter` , the process selects a tablespace randomly each time a temporary object is created. If objects are created within a transaction, tablespaces are rotated to reduce latency (to obtain the random value). Names of tablespaces that do not exist or cannot be used (no privileges) are ignored and do not cause an error.

Temporary files can be large in size and may be undesirable in tablespaces with persistent data. Due to the large volumes of writing to temporary files, the use of magnetic disk storage systems (HDD) may be preferable to storage arrays based on memory chips (SSD), since the resource of the latter is determined by the volume of data written.

The size limit for temporary files used by one process can be set using the `temp_file_limit parameter` . By default, there is no limit.

`temp_file_limit` specifies the maximum amount of disk space that a single process can use for temporary files, such as sorting and hashing, or storing a held cursor. If the limit is exceeded, the command will abort, and any transaction that attempts to exceed the limit will be rolled back.

Limits the size of temporary files that are created implicitly when commands are executed. This option does not limit the size of temporary table files explicitly created by the CREATE TEMPORARY TABLE command.

Parameters `max_slot_wall_keep_size` and `transaction_timeout`

- `max_slot_wall_keep_size` default -1 (no limit) maximum size of log files that can remain in the pg_wal directory after a checkpoint for replication slots
- `transaction_timeout` default is zero (timeout) disabled) . Allows you to roll back any transaction or single command that exceeds a specified time period, not just idle ones. Protects against database horizon holdup and file bloat

Parameters `max_slot_wall_keep_size` And `transaction_timeout`

To prevent unlimited space usage, it is worth checking or setting the values of the following parameters.

`max_slot_wall_keep_size` default -1 (no limit) . The maximum size of log files that can remain in the `pg_wal directory` after a checkpoint for replication slots. If the slot is enabled and a client does not connect, the log files are retained. If no limit is set with this parameter, the log files will fill the entire file system and **the instance will crash** . A server process that fails to write data to the log will be terminated:

```
LOG: server process (PID 6 543 ) was terminated by signal 6: Aborted
```

The instance will then attempt to restart :

```
LOG: all server processes terminated; reinitializing
```

To avoid running out of space, it is worth setting a limit. However, a replica that fails to receive logs and they are erased will have to receive log files from somewhere else or the replica will have to be deleted and recreated.

`transaction_timeout` default zero, timeout disabled. Allows you to cancel not only an idle transaction, but also any transaction or single command whose duration exceeds the specified period of time. The effect of the parameter extends to both explicit transactions (started with the BEGIN command) and implicitly started transactions corresponding to a single operator. The parameter appeared in Tantor DBMS version 15.4.

Long-running transactions and single statements hold the database horizon. Holding the database horizon prevents old row versions from being cleaned up and causes object files to bloat .

Parameters `statement_timeout + idle_session_timeout` do not protect against transactions consisting of a series of short commands and short pauses between them (eg a long series of fast UPDATES in a loop). To protect against long SELECT statements, the parameter can be used `old_snapshot_threshold` . It should not be set on physical replicas. In version 17, `old_snapshot_threshold` and `transaction_timeout were removed`. allows you to replace it .

Parameters `max_connections` And `client_connection_check_interval`

- `max_connections` sets the maximum number of simultaneous connections to an instance
  - default 100
  - Changing the value requires restarting the instance, so it is worth setting it first based on the planned number of sessions.
  - `reserved_connections` parameters `and superuser_reserved_connections`
  - when increasing the value, first increase it on physical replicas, otherwise they will stop serving requests
- `client_connection_check_interval` default is zero, checking is disabled. Allows you to interrupt long queries, the result of which the client will not be able to receive.

Parameters `max_connections and client_connection_check_interval`

`max_connections` defaults to 100. The maximum number of simultaneous connections to an instance. When increasing the value, you must first increase it on the physical replicas, otherwise they will pause their work . The change in value is transmitted through logs. Changing the value requires restarting the instance, so it is worth setting in advance based on the planned number of sessions.

`reserved_connections` parameters and `superuser_reserved_connections` , but you can leave them as is, since the second one has a default value of 3. When the number of simultaneous connections `reaches max_connections - superuser_reserved_connections` , The last three connections can only be made by superusers .

Rough estimates for the number of connections (server processes): maximum TPS on high-performance hardware is achieved in the region of ~ 500 **active** (executing commands) sessions, not idle ones. **An instance with default settings can effectively serve up to ~ 4000 sessions. For larger numbers, instance parameters (e.g. SLRU cache sizes) need to be adjusted.**

`client_connection_check_interval` defaults to zero, disabling checking. Interval between checks (during SQL command execution) by polling the socket state when no data is being transmitted. The socket may have been closed by the peer or by the operating system kernel code due to the peer not responding to keepalive packets . Allows to interrupt long queries, the result of which the client will not be able to get. Can be set to ~ 30 s or less . If the unit is not set, milliseconds are used.

The number of object locks and advisory locks (which are not used automatically, their use requires programming) on an instance is determined by the product of `max_locks_per_transaction * max_connections` (it is assumed that `max_prepared_transactions = 0` and does not need to be changed) . The product of these parameters determines the amount of shared memory for storing locks on the instance. If the space in the memory structure is exhausted, then no process on the instance will be able to obtain an object lock or an advisory lock . The `pg_dump` utility using parallel processes (the `--jobs parameter`) sets locks on all objects being unloaded. When unloading a database - **on all database objects** . If you want it not to fail, it makes sense to set `the max_locks_per_transaction` and `max_connections values` so that their product is not less than the number of objects in each cluster database.

Why not just set `the max_connections parameter` to a large value?

This parameter specifies the total size of the PGPROC structures pointed to by the PROC_HDR structure. PROC_HDR (an array of pointers) is used to find free PGPROC slots when spawning a process and to check the status.

The fields in the PGPROC structure are used by all processes for tasks that require coordination and the fields are queried very frequently. For example, the fields of the PGPROC structure of each process are checked when forming a snapshot, that is, at each SELECT or the beginning of a transaction. Also when synchronizing waits/latches. Since PGPROC are read frequently, pointers to PGPROC memory pages should be kept in hardware caches (TLB). **The larger the product of `PGPROC * max_connections` , the more likely it is that some other structure will have less space, the access speed will drop and the work of processes will slow down.**

max_connections parameter also affects the size of the overall lock structure. Details are described in `src / backend / storage / lmgr / lock.c :`

```
 /* Allocate hash table for LOCK structures . This stores per-locked-object
information . */
  info.keysize = sizeof (LOCKTAG);
  info.entrysize = sizeof (LOCK);
```

max_locks_per_transaction parameter

- The number of object locks and advisory locks that applications can use is limited on an instance by the product of `max_locks_per_transaction * ( max_connections + max_prepared_transactions )`
- by default `max_prepared_transaction =0` and there is no need to change it if distributed transactions are not used
- formula sets the amount of shared memory to store locks in instance memory
- `pg_dump` utility using parallel processes (`the --jobs option`) sets locks on all unloaded objects
- 300000 locks will require ~ 48MB of shared memory

max_locks_per_transaction parameter

The number of object locks and advisory locks that applications can use is limited on an instance by the product of `max_locks_per_transaction * ( max_connections + max_prepared_transactions )` . The formula is defined in `src /backend/storage/ lmgr / lock.c :`

```
#define NLOCKENTS() mul_size ( max_locks_per_xact , add_size ( MaxBackends , max_prepared_xacts ))
```

By default `max_prepared_transactions =0` and does not need to be changed unless distributed transactions are used.

The formula determines the amount of shared memory to store locks in the instance memory . More memory is allocated - up to the nearest power of two.

Each lock is allocated ~ 168 bytes of shared memory. If the memory structure is exhausted, no process in the instance will be able to acquire an object lock or an advisory lock .

`pg_dump` utility using parallel processes (`the --jobs parameter`) sets locks on all objects being unloaded. When unloading a database, **on all objects in the database** . If you want the unloading to be possible, it makes sense to set `the max_locks_per_transaction` and `max_connections` values so that their product is not less than the number of objects for each database in the cluster.

The parameters are set only when the instance is started, changes to their values are transmitted via WAL , on physical replicas the values should be no less (better equal) than on the master. That is, changing these parameters is difficult, since it requires restarting the master and replica instances in the correct order - first the replicas, then the master. Therefore, the values of these parameters should be set in advance.

For example, the number of sessions is 1000, objects in the database are 300000, then `max_locks_per_transaction =300` , the size of the shared memory allocated when starting an instance to store locks on objects is ~ 48MB. The amount of allocated memory will be larger: ~73MB . The changes affect memory with the name `name = <anonymous>` in the query output:

```
SELECT name , allocated_size , pg_size_pretty ( allocated_size ) FROM pg_shmem_allocations ORDER BY size DESC;
```

Background Work Processes

- `max_worker_processes` (default 8) maximum number of background processes on an instance
  - parameter measurement requires instance restart
  - On physical replicas, the parameter value must be no less than on the master.
- `max_parallel_workers` (default 8) maximum number of background processes to serve commands
- `max_parallel_workers_per_gather` (default 2) limit the degree of parallelism
- `parallel_leader_participation` (default `on`) the server process will assist background processes

Background Work Processes

max_worker_processes parameter (default 8) sets the maximum number of background processes on an instance. Measuring this parameter requires restarting the instance. On physical replicas, the parameter value must be at least as large as on the master.

Background processes can be used to perform arbitrary tasks: parallelization of commands, tasks sent for execution by the application by the `pg_background extension functions` . Such tasks may not load the central processor cores.

`max_parallel_workers` parameter (default 8) sets the maximum number of background processes that can be used to parallelize command execution. executed by server processes.

Background processes used to parallelize command execution The tasks performed by server processes usually require a large load on the processor core, so the number of background processes correlates with the number of processor cores. Setting more than `max_worker_processes` does not make sense.

`max_parallel_workers_per_gather` parameter (default 2) sets the maximum number of background processes that can be used by a server process to service a single `Gather node`. or `Gather Merge` from the command execution plan. A value of 0 disables the use of parallel plans. Pipelining (a set of processes reads data and passes it on to another set of processes) is not used, so doubling there is no use of processes as in Oracle Database .

The degree of parallelism of service commands (`CREATE INDEX` when building a `btree index` and `VACUUM` without `FULL`) is limited by the `max_parallel_maintenance_workers parameter` (default 2).

`parallel_leader_participation` parameter (defaults `to on`) specifies that the server process will perform the same work as background processes, rather than sitting idle while background processes are running. Optimality depends on the query plan. In general, the higher the degree of parallelism, the longer the query, and to a lesser extent the amount of data returned by the query, the less optimal the true value is.

`max_worker_processes` and `max_parallel_workers` parameters

- `max_worker_processes` by default 8. The maximum number of background workers that can be running on an instance .
  - set only at cluster level
- `max_parallel_workers` by default 8. Maximum number of parallel processes (parallel workers)
  - Effectively limited by `max_worker_processes` , but can be set to a higher value (for convenience)
  - Parallel processes are used to service SQL commands executed by server processes and to transmit changes in logical replication.
  - can be set at the level of the database, role, role in the database
- `max_parallel_maintenance_workers` default 2. The maximum number of worker processes that can be started by a single CREATE INDEX or VACUUM command (without FULL)

Parameters `max_worker_processes` And `max_parallel_workers`

Parallel processes can significantly speed up the processing of a large number of lines. The optimal number depends on the number of CPU cores. The default values are small and do not depend on the number of cores.

`max_worker_processes` by default 8. Maximum number of background (working) processes (background workers) that can be running on the instance. When increasing the value, increase it on the physical replicas first, otherwise they will pause their work . The change in the value is transmitted through the logs. Set only at the cluster level.

`max_parallel_workers` default 8. Maximum number of parallel processes (parallel workers) . Effectively limited by `max_worker_processes` , but can be set to a higher value (for convenience). Parallel processes are used to service SQL commands executed by server processes, as well as to transfer changes in logical replication (`max_logical_replication_workers parameter` limited by the above parameters). Can be set at various levels, such as the database level, the role level, the role connected to a specific database:

```
postgres=# alter role postgres set max_
max_parallel_maintenance_workers  max_parallel_workers_per_gather
max_parallel_workers              max_stack_depth
```

`max_parallel_maintenance_workers` by default 2. The maximum number of worker processes that can be started by one command CREATE INDEX or VACUUM (without FULL). Autovacuum does not use worker processes. The number of processes is limited by `max_worker_processes` and `max_parallel_workers` .

Parameter `max_parallel_workers_per_gather`

- `max_parallel_workers_per_gather` (default 2) sets the maximum number of worker processes that can used by the server process to service normal (non- maintenance) commands
- If the number of free processes is less than the calculated one, they are used and the command is served by a smaller number of processes (the degree of parallelism decreases).
- By default, the server process participates in data processing (plan nodes below Gather) on an equal basis with parallel processes, but will process fewer rows since it services the serial part of the plan and Gather nodes .
- participation of the server process in processing the parallel part of the plan can be disabled `parallel_leader_participation = off` , but usually does not result in increased productivity

Parameter `max_parallel_workers_per_gather`

Parameter `max_parallel_workers_per_gather` The default is 2. The maximum number of worker processes that can used by a server process to service normal (non- maintenance) commands . Parallel processes service those operations that pass data to Gather or Gather nodes Merge command execution plan. Gather operations are processed by the server process itself. It is not worth setting a value greater than the number of CPU cores , since parallel processes also load all resources (processor, memory, disk) as active server processes. The number of processes is calculated automatically for each SQL command . If the number of free processes is less than the calculated one, they are used and the command is served by fewer processes (the degree of parallelism is reduced). By default, the server process participates in data processing (plan nodes below Gather) equally with parallel processes, but will process fewer rows, since it services the sequential part of the plan and Gather nodes . Participation of the server process can be disabled `parallel_leader_participation = off` but usually does not result in increased performance of either the instance or the team.

`min_dynamic_shared_memory` also relates to the work of parallel processes. It is worth considering if there is a lot of physical memory (hundreds of gigabytes).

Storage system parameters

- `random_page_cost` the default value of 4 is only suitable for a single HDD . For SSD, the value should be close to seq_page_cost , which is 1 by default. You can set the value to 1.1
- `effective_io_concurrency` the default value of 1 is suitable only for a single HDD . For multiple HDDs (RAID), it is set to the number of HDD devices across which the recording is distributed (striping) . For a single SSD on the SATA bus , values from 64 are suitable. For NVMe above 128-512
- `maintenance_io_concurrency` default 10 . Used by vacuum and analysis. For SSDs, the recommended values are the same as for `effective_io_concurrency`

Storage system parameters

There is a set of parameters that is determined by the storage system (" disk "). These can be HDD, SSD (NVMe), their RAID device sets . Characteristics for HDD and SSD are significantly different.

`random_page_cost` the default value of 4 is only suitable for a single HDD. For SSDs, the value should be close to `seq_page_cost` , which is 1 by default. For SSDs , the speed of accessing blocks sequentially and randomly does not differ, taking into account the physical characteristics of the SSD (erase size), which is usually greater than 4K. You can set the value to 1.1

`effective_io_concurrency` default value 1 is ok for a single HDD. For rotational disks (HDD) combined in RAID , you can set the number of disks across which data is distributed (striping) . For a single SSD on the SATA bus , values 64-200 are suitable . For NVMe 500-1000. The maximum value is 1000. The parameter is taken into account by the scheduler for the Bitmap Heap Scan index access method. The parameter tells the scheduler how many I/O operations can be expected to be performed simultaneously. The parameter also enables or disables (value zero) block prefetching. In addition to the normal levels for configuration parameters, the parameter can be set in the tablespace properties.

`maintenance_io_concurrency` default is 10 . Used by vacuum and analysis. For SSDs, the recommended values are the same as for `effective_io_concurrency` .

These options can be set at the tablespace level:

```
alter tablespace name set ( <TAB>
EFFECTIVE_IO_CONCURRENCY RANDOM_PAGE_COST
MAINTENANCE_IO_CONCURRENCY SEQ_PAGE_COST
```

Checkpoint parameters

- parameter `full_page_writes =on` (default)
- `checkpoint_completion_target = 0.9` (default)
- `checkpoint_timeout` is recommended to be set to 20-30 minutes
- `max_wall_size` log file size (pg_wal) after which a checkpoint is called earlier than the `checkpoint_timeout parameter specifies`
  - acceptable for bulk loading or data changes: when blocks are filled in seconds and are not changed later
- c statistics in columns `checkpoints_req` and `checkpoints_timed` `pg_stat_bgwriter` views

Checkpoint parameters

When setting up checkpoints, you should first check that `full_page_writes = on` (the default value). Installation to off will reduce the volume of log entries and improve performance, but will increase the likelihood of failure. recover from instance failure. The off value is set temporarily when you have backups (physical replicas) and the ability to redo changes in the event of an instance failure.

`checkpoint_timeout` by default `5 min` . This value is too small and not optimal. It is recommended to set it to 20-30 minutes.

`checkpoint_completion_target` defaults to 0.9, which is optimal.

`max_wal_size` by default 1GB. Specifies the approximate size of WAL segments, upon reaching which a checkpoint " by size " is called . Executing a checkpoint allows you to remove WAL segments that store changes that occurred before the checkpoint. Usually, checkpoints are executed " by time " - with the frequency specified by the `checkpoint_timeout parameter` . **" By size " checkpoints are acceptable for bulk loading or data changes: when blocks fill up in seconds and are not changed later** .

`max_wal_size` is selected so that checkpoints are most often called by time, and not by size, while fitting into the size of the file system on which the pg_wal directory is mounted. If pg_wal has enough free space, you can set `max_wal_size` to a large value. For SSD ~1/4 (depends on the SSD controller algorithm) of the SSD size , so that fit into the size of the SSD SLC cache .

If PGDATA and tablespaces are located on slow HDDs , then setting `s hared_buffers` can roughly give a performance boost of ~ 3.5 times compared to the default value, setting `max_wal_size` by ~ 1.5 times. Checkpoints performed more often than once every ~ 15 minutes allow dirty blocks to stay in memory for less time. Because of this, the same block will be dirty several times and written to disk several times.

Checkpoint Statistics - `checkpoints_req columns` and `checkpoints_timed pg_stat_bgwriter` views .

https://www.enterprisedb.com/blog/tuning-maxwalsize-postgresql

Background Recording Process Parameters **bgwriter**

- bgwriter_delay default 200ms , reduce to 20-40ms
- bgwriter_lru_multiplier by default 2. Usually set the value to more than 4-10.
- bgwriter_lru_maxpages default 100 buffers, increase to 400-500
- bgwriter_flush_after default 512kB
- The values are set based on data from the pg_stat_bgwriter view :

```
select * from pg_stat_bgwriter \ gx
-[ RECORD 1 ]---------+---------
checkpoints_timed | 1461
checkpoints_req | 90
checkpoint_write_time | 3805215
checkpoint_sync_time | 116032
buffers_checkpoint | 130655
buffers_clean | 250785
maxwritten_clean | 1421
buffers_backend | 5243535
buffers_backend_fsync | 0
buffers_alloc | 653441
```


Background Recording Process Parameters **bgwriter**

bgwriter_delay default is 200ms . If there are no dirty buffers left in the shared pool, bgwriter may be inactive longer than specified. The timer resolution on many systems is 10 ms, so setting bgwriter_delay to a value that is not a multiple of 10 will produce the same result as the next multiple of 10. The default value is high, which is suitable for an unloaded cluster. Under load, the process will work inefficiently and the checkpoint er will clean up the buffers. and server processes. A small value can load one of the processor cores. If there are many cores and the instance is loaded, it makes sense to set it to a small value : 20 ms .

bgwriter_lru_multiplier by default 2 . Usually the value is set to 4-7 .

bgwriter_lru_maxpages default 100 buffers . Per cycle bgwriter records no more than was recorded in the previous cycle, multiplied by bgwriter_lru_multiplier , but not more than bgwriter_lru_maxpages . Usually increased to 400-500.

bgwriter_flush_after Default is 512kB . The value range is from 0 (disables flush) to 2MB.

The values are set based on data from the pg_stat_bgwriter view :

```
select * from pg_stat_bgwriter \ gx
-[ RECORD 1 ]--------+------------------------------
checkpoints_timed | 1461
checkpoints_req | 90
checkpoint_write_time | 3805215
checkpoint_sync_time | 116032
buffers_checkpoint | 130655
buffers_clean | 250785
maxwritten_clean | 1421
buffers_backend | 5243535
buffers_backend_fsync | 0
buffers_alloc | 653441
```

I/O statistics for server and background processes are available in the pg_stat_io view .

Practice

- Part 1. Object locks
- Part 2. Monitoring the server process memory
- Part 3. Temporary tables and files
- Part 4. Impact of configuration parameters on shared memory
- Part 5. `max_connections parameter` and performance
- Part 6. Buffer Cache Size and Buffer Release

Practice

Practical examples allow us to compare the data on allocated memory provided by operating system utilities and system directory functions.

You will see:

how memory is allocated for temporary tables and at what point are temporary table files created ;

how changes in configuration parameters affect the actual sizes of memory structures ;

how much table deletion slows down when the buffer cache is increased.

6

6- 1

Storage structures

Tables

- an object in which data is stored
- several types: regular tables (heap tables , rows are stored in an unordered manner) , unlogged , temporary, sectioned
- Extensions can create new ways to store data and access it
- number and order of columns are set when creating a table
- After creating a table, you can add and remove columns. When adding a column, it is added last - after all existing columns
- you can change the column type

Tables

Application data is stored in tables. The DBMS has regular tables (heap tables , rows are stored in an unordered manner) , unlogged , temporary, partitioned. Extensions can create new ways of storing data and methods of accessing it. Tantor SE DBMS has the `pg_columnar` (`pgcolumnar`) extension.

Number and order of columns are specified when the table is created. Each column has a name. After the table is created, you can use the ALTER TABLE command to add and remove columns. When you add a column, it is added after all existing columns.

NULL values by default or are given the values specified by the DEFAULT option. When adding a column, new row versions will not be generated if DEFAULT is set to a static value. If the value uses a volatile function, such as `now()` , then when adding a column, all rows in the table will be updated, which is slow. In this case, it may be more optimal to first add the column without specifying DEFAULT , then update the rows with UPDATE commands setting the value for the added column, then set the DEFAULT value with the ALTER TABLE command table ALTER COLUMN column SET DEFAULT value ;

Dropping a column deletes the values in the fields of each row and the integrity constraints that include the dropped column. If the integrity constraint being dropped is referenced by a FOREIGN KEY , you can drop it in advance or use the CASCADE option.

You can also change the column type using the ALTER TABLE command table ALTER COLUMN column TYPE type(dimension);

You can change the type if all existing (non- NULL) values in the rows can be implicitly cast to the new type or dimension. If there is no implicit cast and you do not want to create one or set it as the default data type cast, you can specify the USING option and set how to get new values from existing ones.

The DEFAULT values (if defined) and integrity constraints that the column is a part of. It is better to remove integrity constraints before modifying the column type and then add the constraints.

To view the contents of a block, the functions of the standard pageinpect extension are used .

https://docs.tantorlabs.ru/tdb/ru/15_6/se/ddl-alter.html

Service columns

- `xmin` - transaction number (xid) that created the row version
- `xmax` - transaction number (xid) that deleted or attempted (transaction was not committed for any reason: rollback was called , server process was interrupted) to delete a row or zero
- `ctid` address of the physical location of the row
- `tableoid` - oid table that physically contains the row. The values are meaningful for partitioned and inherited tables
- `cmin is` the zero-based sequence number of the command within the transaction that created the row version
- `cmax is` the zero-based sequence number of the command within the transaction that deleted or attempted to delete the row

Service columns

When accessing table rows in SQL commands, you can use pseudocolumn names (service, system , virtual). Their set depends on the table type. For regular (heap) tables, the following pseudocolumns are available :

`ctid` is the address of the physical location of the row. Using `ctid` , the scheduler can access a page (block of the primary layer file) of the table without a full scan of all pages. `The ctid` will change if the row is physically moved to a different block.

`tableoid` - oid table that physically contains the row. The values are meaningful for partitioned and inherited tables . A quick way to find out oid tables, as it corresponds to `pg_class.oid` .

`xmin` - the transaction number (xid) that created the row version.

`xmax` - transaction number (xid) that deleted or attempted (the transaction was not committed for any reason: rollback was called , the server process was interrupted) to delete the row.

`cmin is` the zero-based sequence number of the command within the transaction that created the row version. Has no application.

`cmax is` the sequence number of the command within the transaction, starting from zero, that deletes or attempted to delete the row. To support " crooked " code, when the same row is updated several times in one transaction.

`x min , c min , x max , c max` are stored in three physical fields of the row header. `x min` and `x max` are stored in separate fields. `c min , c max , x vac` (VACUUM FULL was used before PostgreSQL version 9) in one physical field. `cmin` and `c max` are only of interest during the lifecycle of a transaction for insertion (`cmin`) and deletion (`c max`). `ctid` calculated based on the address of the row. Physically, the row version stores `t_ctid` stores the address of the next (created as a result of UPDATE) version of the row. Moreover, this is not a " chain " , the connection can be lost, since the vacuum can delete a newer version of the row earlier than the old one (the block processed earlier) and the old version of the row will refer to the missing version. If the version is the latest, then `t _ ctid` stores the address of this version. For partitioned tables, if an UPDATE resulted in the new version moving to a different partition (the value of a column included in the partition key changed), a special value is set. Also During an INSERT , a "speculative insertion token" may be temporarily set instead of the row version address.

https://docs.tantorlabs.ru/tdb/ru/15_6/se/ddl-system-columns.html

pageinspect extension

- standard extension, does not require loading the library
- checksum:
  - calculated and stored at the time of writing a dirty block to the file
  - checked when reading a block from a file into the buffer cache buffer
  - while the block is in the buffer the checksum field in the block header does not change and is not checked

```
create extension pageinspect ;
drop table if exists t;
create table t(s text);
insert into t values ('a');
select * from page_header ( get_raw_page ('t', 0));
    lsn | checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6EC12C0| 0 | 0 | 28 | 8144| 8176 | 8192 | 5 | 0
vacuum full t;
select * from page_header ( get_raw_page ('t', 0));
    lsn | checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6ED5458| -11261 | 0 | 28 | 8144| 8176 | 8192 | 5 | 0
```


pageinspect extension

Standard extension. Can be installed into the production database with the command:

```
create extension pageinspect ;
drop table if exists t;
create table t(s text);
insert into t values ('a');
select * from page_header ( get_raw_page ('t', 0));
    lsn   | checksum |flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6EC12C0| 0 | 0 | 28 | 8144| 8176 | 8192 | 5 | 0
```

Why is the checksum zero ?

Checksum:

1) is calculated and saved at the moment of writing a dirty block to the file

2) is checked when reading a block from a file into the buffer cache buffer

3) while the block is in the buffer, the checksum field in the block header does not change and is not checked

When creating a table, the block in the file is empty: it consists of zero bytes, including a zero checksum. While the block is in the buffer, the checksum in it remains the same as it was at the time of reading from the file. In memory, if the buffer is not cleared and not read from the file again, the checksum is the same - at the time of reading from the file into the buffer.

After restarting the instance or a complete vacuum, the checksum field will be updated:

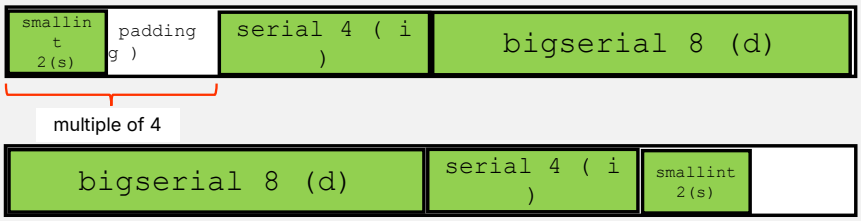
```
vacuum full t;
select * from page_header ( get_raw_page ('t', 0));
    lsn | checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6ED5458| -11261 | 0 | 28 | 8144| 8176 | 8192 | 5 | 0
```

When a block changes while it is in the buffer cache , the value in the checksum field will not be updated :

```
delete from t;
   lsn | checksum|flags|lower|upper|special|pagesize|version|prune_xid
----------+--------+-----+-----+-----+-------+--------+-------+---------
6/B6ED7008| -11261 | 0 | 28 | 8144| 8176 | 8192 | 5 | 1026
```

Padding and aligning

- alignment (align) - the field length is a multiple of the number of bytes
- specified in the pg_type.typalign columns and pg_attribute.attalign
- The following alignment values are possible:
  - s (short) 2 bytes
  - i (int) 4 bytes
  - d (double) 8 bytes
  - c (char) unaligned (byte by byte)
- padding - adding unused space to make alignment



Padding and aligning

In the `pg_type.typalign columns` And `pg_attribute.attalign` the alignment is specified when storing a field for a data type in normal (heap) tables. All types:

```
select distinct typname , typalign from pg_type where typname not like 'pg_%' and typname not like '\_%' order by typname ;
```

All types used in columns:

```
select distinct atttypid :: regtype , attalign from pg_attribute order by attalign ;
```

Alignment can be:

`c ( char ), 1 byte` , i.e. without alignment
`s ( short ), 2 bytes`
`i ( int ), 4 bytes`
`d ( double ) 8 bytes`
`x` for 64 -bit `xid` type in Tantor SE and SE1C.

Padding - adding unused space to perform alignment .

For example, two tables are created with different column orders:

```
create table t1(c1 varchar (1), c2 bigserial , c3 date, c4 timestamp);
create table t2(c1 bigserial , c2 timestamp, c3 date, c4 varchar (1));
```

Rows with the same values are inserted:

```
insert into t1 values('A', 1, now(), current_timestamp );
insert into t2 values(1, current_timestamp , now(), 'A');
```

Strings will be stored as a sequence of bytes in HEX :

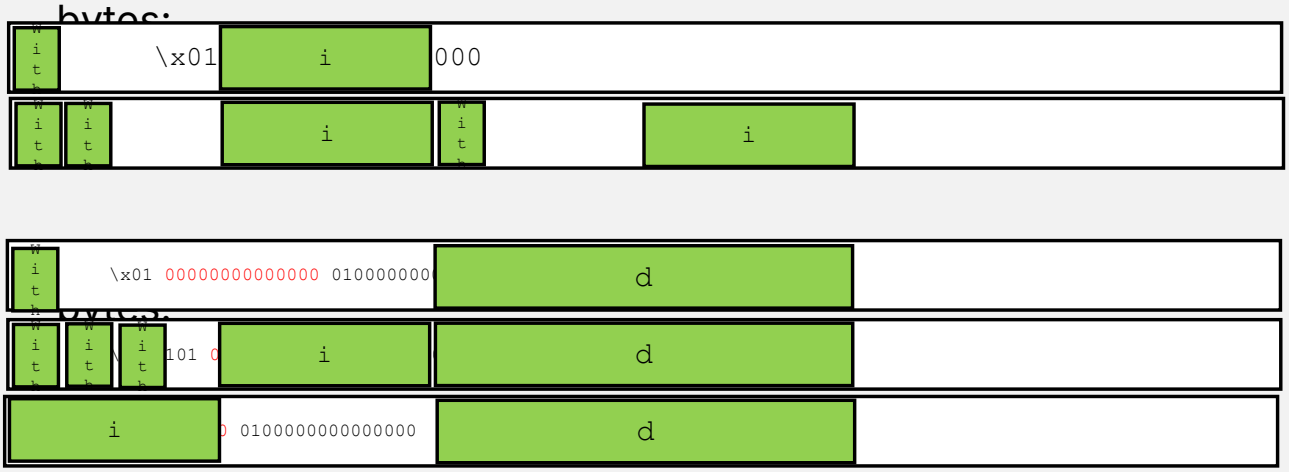
```
create extension pageinspect ;
select t_data , lp_len , t_hoff from heap_page_items ( get_raw_page ('t1','main',0));
                          t_data                                | lp_len|t_hoff
----------------------------------------------------------------- - -+------+------
0541 000000000000 0100000000000000 36230000 00000000 8a31e17666c40200  56   24
select t_data , lp_len , t_hoff from heap_page_items ( get_raw_page ('t 2 ','main',0));
                          t_data | lp_len|t_hoff
-----------------------------------------------------------------+------+------
0100000000000000 c93ae17666c40200 36230000 0541        46    24
```

The difference is 8 bytes per row version. Length of the string version 56 And 46 bytes, row version header size 24 bytes for Tantor SE. For PostgreSQL, the row version length is 64 and 54 bytes , the row version header size is 32 bytes (stored in one byte t_hoff , the value is a multiple of 8) . https://docs.tantorlabs.ru/tdb/ru/16_4/se/storage-page-layout.html

Alignment

- " c " and variable types up to 127 bytes long are not aligned:

- " i " - the beginning of the field can only be located in 1,5,9,13... bytes:



Alignment

Fields of variable length types (text, numeric) up to 127 bytes long are not aligned , from 127 bytes are aligned by pg_type.typalign (if the value is " i " , then by 4 bytes) .

types in the typlen column pg_type tables value -1:

```
select typname , typlen , typalign from pg_type where typname like '% bool %';
 typename | typen | typealign
----------+--------+-----------
 bool | 1 | c
_ bool | -1 | i
(2 rows)
```

Example for 2 pictures on slide:

```
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
--------------------+--------+--------
\x 01 000000 01000000 | 32 | 24
```

Example for 6 pictures on slide:

```
drop table if exists t;
create table t( a int4 , b int8 );
insert into t values (1, 1);
select t_data , lp_len , t_hoff from heap_page_items ( get_raw_page ('t','main',0));
               t_data | lp_len | t_hoff
--------------------------+--------+--------
\x 01 00000000000000 0100000000000000 | 40 | 24
```

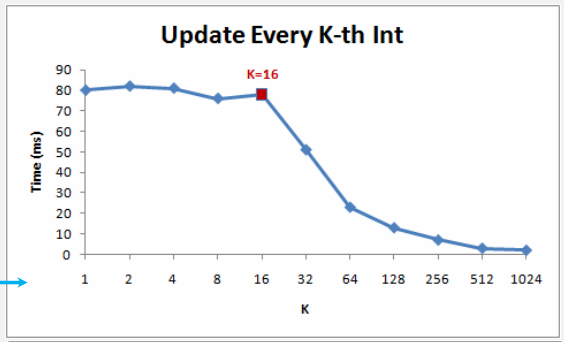
Example for numeric:

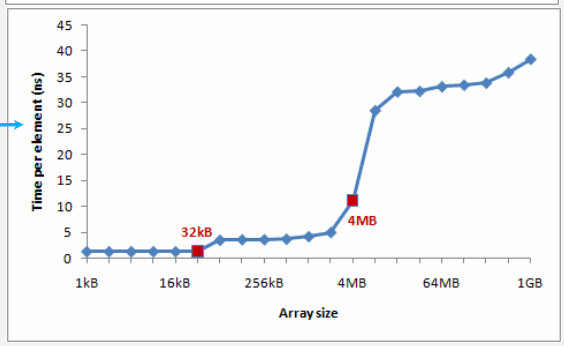
```
drop table if exists t;
create table t( a numeric , b numeric , c int4 );
insert into t values (1, 1, 1);
select t_data , lp_len , t_hoff from heap_page_items ( get_raw_page ('t','main',0));
               t_data | lp_len | t_hoff
--------------------------+--------+--------
\x 0b00800100 0b008001000 000 01000000 | 40 | 24
(1 row)
```

The value 1 in the numeric fields takes up 5 bytes, but it is aligned to 1 byte, so there is no padding between columns a and b . The int4 field is located from 13 to 16 bytes. The padding (unused bytes) is marked in red.

cache line

- alignment significantly speeds up data processing, so it is used whenever possible where it does not lead to a significant increase in storage space
- up to 16*8=64 bytes (K=16) the time to access memory is the same
- time to process data if it fits in the cache





cache line

Alignment significantly speeds up data processing, so it is used whenever possible where it does not lead to a significant increase in storage space.

Example of two cycles:

```
int [] arr = new int [64 * 1024 * 1024];
for( int i = 0; i < arr.length ; i += 1) arr [ i ] *= 3;
for( int i = 0; i < arr.length ; i += 16) arr [ i ] *= 3;
```

In the array of integers of 4 bytes, all numbers are changed in the first cycle (arithmetic operation of multiplying by 3). In the second cycle, every sixteenth number is changed. The execution time of the cycles will be the same: the second cycle is faster by 2.5%. Most of the time is spent on accessing memory. Just as files on HDD and SDD are read in blocks (512 bytes or 4K or other size), the processor also works with the main (not cache) physical memory in pieces called cache lines . Modern processors have a **cache line size (LineSize)** of 64 bytes. How the processor physically reads bytes or sets of bytes does not matter, what is important is that the time to read from the main memory from 1 to 64 bytes into the cache and the subsequent writing to the main memory from the cache is the same. The time to process from 1 to 64 bytes in the cache is ~ 60 times faster. In cycles, 16 numbers of type int (4 bytes = 32 bits) occupy 64 bytes - cache line.

The graph on the slide shows the time it takes to process an array depending on the step of reading int numbers .

If you complicate the cycle and repeatedly return to processing values, then the speed of processing values stored in caches will be higher up to the cache boundaries , and then fall. Example:

```
int steps = 64 * 1024 * 1024;
int lengthMod = arr.length - 1;
for( int i = 0; i < steps; i ++) arr [( i * 16) % lengthMod ]++;
```

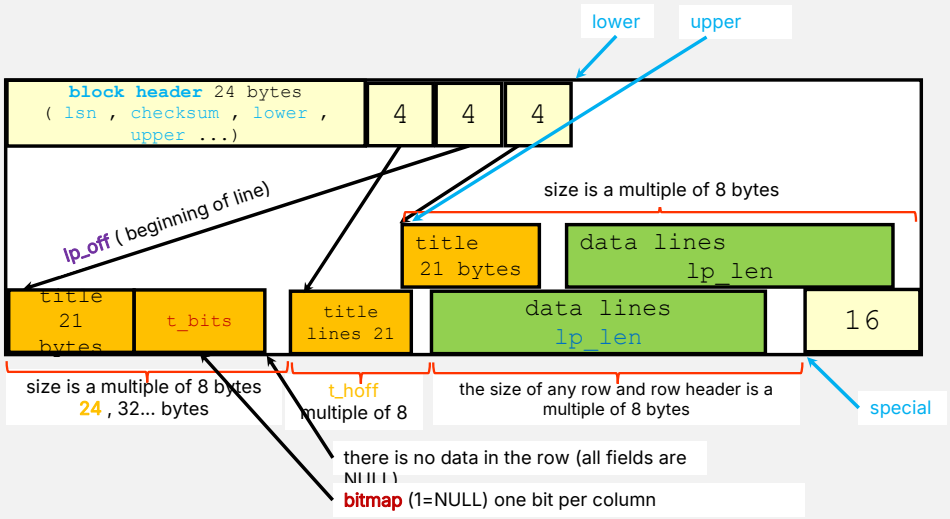
The graph on the slide shows an example for the core cache sizes L1 = 32 KB, L2 = 4 MB.

PostgreSQL shared memory structures are 64-byte aligned. Each core can have its own cache. If the same memory addresses are loaded into the mapped caches of the cores, then when the value in the cache changes , in addition to writing the entire cache line to main memory, invalidated cache line (64 bytes) in the cache of all other cores and if the cores simultaneously work with bytes within 64 bytes, the slowdown is ~ 15 times.

https://igoro.com/archive/gallery-of-processor-cache-effects/

Cache line contention (Andres Freund): https://m.youtube.com/watch?v=dLrqQOCRFOU

Data block structure




```
select * from page_header (
get_raw_page ('t','main',0));
-[ RECORD 1 ]---------
lsn | 0/110DCF10
checksum | 0
flags | 0
lower | 928
upper | 944
special | 8176
pagesize | 8192
version | 5
prune_xid | 0
```

```
select * from heap_page_items ( get_raw_page ('t','main',0)) limit 1;
lp| lp_off |lp_flags| lp_len |t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask| t_hoff | t_bits | t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
18 | 776 | 1| 38 | 43339| 0 | 369 |(0, 18 )| 6 | 2307 | 24 | 01111100 |
```


Data block structure

The structure of the heap table block is given . The block size is 8Kb. At the beginning of the block there is a service structure of a fixed size of 24 bytes. They contain: LSN indicating the beginning of the journal record following the journal record of which the block was changed. This LSN is needed so that the block is not sent for writing if the journal record has not been written to disk (implementations write ahead log rules) . Also used in log recovery.

Tantor SE uses a 64-bit (8 bytes) transaction counter and at the end of a block of regular tables there is a " special space " of 16 bytes , TOAST has 8 bytes . PostgreSQL does not have a special area for tables, index blocks have one.

`prune_xid` - default 0. xmax oldest unpurged row. It is used as a hint (it's like a hint-bit , a 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 look for a place in the block to try to prune the block (First check whether there's any chance there's something to prune) . It is requested in the heap_page_prune_opt (..) function which is called by the heapgetpage (..) function, which on occasion (opt - opportunity) may prune the block.

After the fixed area there are pointers (line pointers) to the beginning of records (lines) in this block (`itemid.h`) . For each line, 4 bytes are used for the pointer . Why so much ? The pointer contains the offset (" **off** set") in bytes to the beginning of **the** line (l p_off 15 bits, line pointer **off** set), 2 bits (l p_flags) , 15 bits **lengths per** line (l p_len) **.** Two bits indicate four possible states of the pointer: 1 - points to a string, free, and two more states that implement HOT (heap only tuple) optimizations : dead and redirect.

pd_flags) in the block header . Bits (flags) can be used in the process code as hints, or they may not be used :

```
PD_HAS_FREE_LINES 0x0001 /* there may be Unused pointers in the header */
PD_PAGE_FULL 0x0002 /* there may not be room for new lines */
PD_ALL_VISIBLE 0x0004 /* all rows are visible to all transactions, no dead ones */
PD_VALID_FLAG_BITS 0x0007 /* used as checksum for these bits */
```

Although the address (offset to) the start of a row for table blocks is a multiple of 8 bytes, the offset stores the address with byte precision.

```
select * from page_header ( get_raw_page ('t','main',0));
lsn | checksum|flags | lower | upper | special | pagesize | version | prune_xid
----+--------+-----+-------+-------+---------+--------+---------+----------
0/50| 0| 0| 928 | 944 | 8176 | 8192| 5 | 0
```

The request does not require specifying the layer name 'main' :

```
select * from page_header ( get_raw_page ('t', 0));
  lsn | checksum| flags |lower | upper | special|pagesize|version|prune_xid
-----------+--------+-----+-----+-------+-------+--------+------ +---------
6/C0D 8 | 0 | 2 | 40 | 1264 | 8176 | 8192 | 5 | 966
```

In this example, the `PD_PAGE_FULL flag is set ( flags =2 )` , meaning that UPDATE did not find space in the block and was forced to insert a new version of the row into another block. When accessing a block with such a flag (even SELECT), the process will try to clean up space in the block (perform a HOT cleanup) . The block also has the hint `prune_xid =966` .

```
select ctid ,* from heap_page ('t',0);
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
------+--------+------+--------+------+------+-----+-----+--------+------
(0,1) | 6448 | (0,1) | normal | 963c | 966c | t | | (0.3) | f
(0,2) | 4720 | (0,2) | normal | 964c | 967c | t | | (0.4) | f
(0.3) | 2992 | (0.3) | normal | 966c | 969 | | t | (1,1) | f
(0.4) | 1264 | (0.4) | normal | 967c | 968c | | t | (0.4) | f
select ctid from t;
 ctid
-------
(1,1)
```

The process that executed the previous select cleared the block:

```
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
------+-------+------+-------+------+------+-----+-----+--------+------
(0,1) | 0 | (0,1) | dead | | | | | f
(0,2) | 0 | (0,2) | dead | | | | | f
  lsn | checksum|flags|lower|upper|special|pagesize | version | prune_xid
- -----------+--------+-----+-----+-----+-------+--------+---------+----------
6/DE30| 0| 0| 32|8176 | 8176 | 8192 | 5 | 966
```

`PD_PAGE_FULL` flag was cleared, but prune_xid remained.

```
vacuum t;
```

Vacuum cleared line references in the block header and set `the PD_HAS_FREE_LINES and PD_ALL_VISIBLE flags` (**`flag=5`**):

```
 ctid | lp_off | ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
------+--------+------+--------+------+------+-----+-----+--------+------
(0,1) | 0 | (0,1) | unused | | | | | f
 lsn | checksum|flags|lower|upper | special | pagesize | version | prune_xid
---- - -+--------+-----+-----+-----+---------+----------+---------+----------
6/0648| 0| 5 | 28| 8176| 8176 | 8192 | 5 | 0
```

The row header is 24, 32, ... bytes in size and is a multiple of 8 bytes. It stores t_hoff - offset to the beginning of the row data. At the end of the header there will be a bitmap t_bits (size is a multiple of a byte), if at least one field of the row is NULL . One bit - one column, zero - NULL , 1 - the field is not empty . The presence of the map (the presence of NULL in any field) is indicated by one of the bits t_infomask .

```
select * from heap_page_items ( get_raw_page ('t','main',0)) limit 1;
lp| lp_off |lp_flags| lp_len |t_xmin|t_xmax|t_field3|t_ctid|t_infomask2|t_infomask| t_hoff | t_bits | t_oid
--+------+--------+------+------+------+--------+------+-----------+----------+------+--------+-----
18 | 776 | 1| 38 | 43339| 0 | 369 |(0, 18 )| 6 | 2307 | 24 | 01111100 |
```

Number of rows in a block

- no more than 291 for lines of length zero (all fields are empty)
- the maximum number of non-empty lines that can fit in a block is: **226, 185, 156 (157), 135 (136), 1 19 (120) , 107, 97, 88, 81, 75, 70, 65,** ... since the line length is a multiple of 8 bytes
- 4 bytes are added to the block header for each line
- at the beginning of the block 24 bytes are occupied by service data
- in Tantor SE, SE1C at the end of the table block 16 bytes are used for service data
- if there are no more than 8 columns in the table, then the row header is 24 bytes
- row header is a multiple of 8 bytes and can be 24, 32,... bytes in size
- in the row header, space is allocated for a bitmap of empty values in the row fields if the columns can contain NULL
- empty fields do not take up space in the data area
- NULLs do not take up space in the data area, using NULLs is efficient in terms of storage compactness
- FILLFACTOR (fill percentage) for tables by default is 100%

Number of rows in a block

To visualize the distribution of space in a table page, let's look at an example. A table **with two columns, uuid and boolean** . We insert rows by filling uuid with the `uuid_generate_v1()` value , and boolean with the true value . These data types have a possible storage method of PLAIN . **FILLFACTOR for PostgreSQL tables is 100% by default** , for btree indexes 90 %. For such a table, 156 rows will fit in a Tantor SE page . In a postgresql page from Astralinux 1.8.1 136 lines will fit . If boolean leave empty (null) , the page will fit 185 and 157 lines. If both fields are NULL, then 291 and 226 rows.

The page header is 648 bytes and 568 bytes (`page_header.lower`) . **The block header is 24 bytes plus 4 bytes for each row in the block.**

In the Tantor SE block, 16 bytes are used at the end of the block to store service data. In postgresql the data area reaches the end of the block, there is no special area at the end of the blocks in the table.

The length of the line in Tantor SE 4 is 1 byte (lp_len) , but the line takes up 48 bytes (lp_off) , because: all fields are aligned ; header is 8 bytes ; header + data is 8 bytes .

At postgresql Astralinux the total length of the string is 56 bytes and lp_len = 49. Tantor SE Row Header - 24 bytes and postgresql Astralinux - 32 bytes (stores t_infomaskpgac , t_hasmac , t_maclabel).

Example of a table with columns:

```
create table t1 ( c1 varchar (1), c2 bigserial , c3 date, c4 timestamp );
create table t2 ( c1 bigserial , c2 timestamp , c3 date , c4 varchar ( 1 ) );
```

Number of lines in block: Tantor SE 1 35 (non-optimal column order) and **15 6** , in postgresql Astralinux : 120 and 136. Rearranging columns reduced overhead by ~10%.

The block header size in Tantor SE is 564 bytes and 648 byte, in postgresql Astralinux : 504 and 568 .

Storage overhead is ~13% , of which block header is ~7% , row headers are ~6%.

The maximum number of rows in an 8-kilobyte PostgreSQL block is 291. However, completely empty rows (all NULL fields) almost never occur. Therefore, you can focus on the fact that **the maximum number of non-empty lines that can fit into a block is: 226 , then 185, then 156** (PostgreSQL has 1 more line due to 16 bytes at the end of the block in Tantor SE) **, then 135 , then 120.** For example, for tables: `t 1 (c bigserial ); t 2 (c serial); t 3 (c smallint );` the number of lines that fit into the block will be the same: 226. Why ? The row header is 24 bytes, the row is aligned to 8 bytes and the data area can be 0,8,16,24... bytes.

```
 9 | 672 (the line will still fit in 10% of the block size) for hot cleanup )
 8 | 984 (size exceeds 10% of block size)
 7 | 1136
 6 | 1328
 5 | 1600
 4 | 2008
 3 | 2688
 2 | 4048
 1 | 8120
(60 rows)
```

By default FILLFACTOR=100, but HOT cleanup is triggered if there is less than 10% free space in the block. Is 90% a lot or a little ? If the block fits more than 9 lines, then the freed space is enough to fit the line. The exact statistics of cases when during UPDATE no space was found in the block and the new version of the row was inserted into another block is shown by `pg_stat_all_tables.n_tup_newpage_upd` .

the most optimal full row size is a multiple of 64 bytes (cache line size) . Number of rows and row data area size:

```
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
...
```

If there are no more than 8 columns in the table, the row header is 24 bytes. If there are more than 8 columns, the row header is 32 bytes. Starting with 73 columns, the row header will be 40 bytes.

The order of columns in a table

- the first columns should be d (aligned by 8 byte) , then i , then s (smallint), then c (boolean , char, uuid)
- variable length types (text, bytea, numeric and many others) have i alignment or d . However, **fields of these types will only be aligned if they store more than 126 bytes.**
- for variable length types in the typlen column of the pg_type table negative number: -1 denotes varlena type
- columns with a large number of NULL values should be placed after fixed-width fields and before varlena (if they will store more than 126 bytes)

The order of columns in a table

The first columns should be d (aligned by 8 byte) , then i , then s (smallint), then c (boolean , char, uuid) . Is it possible to put i first , and then d? It is possible. For example, " iid " - there will be no loss of space. "id" - there will be a loss of 4 bytes of space, since d must be located starting from the 8th byte.

Variable length types (text, bytea, numeric and many others) have i alignment or d . However, **fields of such types will only be aligned if they store more than 126 bytes** . For variable-length types, in the typlen column of the pg_type table negative number: -1 denotes the type varlena . There is also a value " -2 " (cstring and unknown) are pseudotypes (`pg_type.typtype ='p'`) . Pseudotypes cannot be used as a column data type, they are used as parameters of routines to pass values.

Columns with a large number of NULL values should be placed last . NULL is passed directly from the bitmap preceding the first field, so the location of the field with the NULL value does not affect the cost of access, but the dimensionality of the data type of these columns affects both the alignment and access to the data of the columns that follow them.

```
create table t1 (c1 serial, c2 timestamp);
create table t2(c2 timestamp, c1 serial);
insert into t1 (c2) values( current_timestamp );
insert into t2 (c2) values( current_timestamp );
select t_data , lp_len , t_hoff from heap_page_items ( get_raw_page ('t1','main',0))
union
select t_data , lp_len , t_hoff from heap_page_items ( get_raw_page ('t2','main',0));
               t_data | lp_len | t_hoff
--------------------------+--------+--------
\x0100000000000000b51ff02b6fc40200 | 40 | 24
\x8523f02b6fc4020001000000         | 36 | 24
```

On lp_len cannot be oriented, since in addition to padding , the total size of the line located in the block is aligned . This means that if lp_len is not divisible by 8, then 1-7 bytes (with a value of zero) will be added to the end of the string to make the size a multiple of 8 bytes. lp_len stored in the block header and used to determine the end of the last field.

The numbers 36 and 40 are multiples of 8, so in this example there will be a saving of 4 bytes per row.

Column Order and Performance

- The columns included in the PRIMARY KEY should be listed first as they do not have null values and are the most frequently used.
- next are fields with fixed-width data types that do not have empty values
- Next are columns with fixed-width data types that rarely contain null values
- columns that are less common in queries should come after columns that are more common
- columns with larger fields are best placed last
- take into account the algorithm for removing fields in TOAST
- **if the table has no more than 8 columns, then the row header is 24 bytes**
- If there are more than 8 columns, then NULL in any of the row fields increases the row header by 8 bytes, but on the other hand the data area decreases by the size of the field.

Column Order and Performance

Empty fields occupy zero bytes in the data area. If the table has a large number of columns and NULL appears in at least one field, the t_bits bitmap can jump (by 8 bytes) to increase the row header from 24 to 32 bytes. In the bitmap, 1 column is 1 bit, its size is a multiple of a byte.

The row header without a bitmap takes 21 bytes. Physically, the header is aligned to 24, 32, 40... bytes. To prevent the header from growing from 24 to 32 bytes, the bitmap must fit into 3 bytes, i.e. **if the table has no more than 8 columns, the header** is 24 bytes.

When the row size exceeds ~ 200 0 bytes (after compression of variable width fields, which strategy allows To TOAST , variable-width fields are pushed out, starting with the longest in the row, until the row fits in ~ 2,000 bytes . If there are no more push-out fields left, the row will exceed this limit. Fixed-width fields cannot be stored in TOAST and cannot be compressed .

Example: a table with 25 int4 columns . If all fields are non-empty, the row header is 24 bytes. If at least one field is empty, the row header is 32 bytes. In both cases, the block will fit 61 rows because the length of the row without NULL is 24+25*4= 124 padded with 4 bytes to be a multiple of 8. Length of a string with NULL: 32 + **24** *4 = 128 (one column less, since NULL takes up zero bytes in the data area)

```
lp | lp_off | lp_len | t_ctid | t_hoff
61 | 368 | 124 | (0, 61 ) | 24
61 | 368 | 128 | (0, 61 ) | 32
```

If the first column is made int8 , and the second or subsequent column is NULL, then t_hoff = 32 , lp_len = 132 the block will fit 58 lines, **5%** less:

```
58 | 288 | 132 | (0, 58 ) | 32
```

Most of the columns can be in TOAST and the difference will be bigger.

The total optimization effect from bad order to better order (no variable width columns before fixed width columns) if the row fits in the block is ~ 27%.

The reason for using aligning is to improve performance when processing a set of bytes, otherwise padding would not be used.

https://gitlab.com/dhyannataraj/tuple-internals-presentation

Practice

- Part 1. Free space map
- Part 2. Changing the order of columns
- Part 3. Table Block Contents
- Part 4. Aligning fields in table rows
- Part 5. Aligning Rows in Table Blocks
- Part 6. Storing empty (NULL) values in table rows
- Part 7. Number of rows in a table block

Practice

In practice you will see:
how does the order of columns affect the space occupied by a table ;
that the presence of an index slows down insertion into a table by an order of magnitude ;
how much the row header is increased if any field in the row contains an empty value ;
how to view the contents of table blocks .

7

Indexes

String Access Methods

```
\ dA
List of access
methods
Name | Type
----------+-------
bloom | Index
 brin | Index
 btree | Index
columnar | table
gin | Index
gist | Index
hash | index
heap | table
 spgist | Index
(9 rows)
```

- two types of " methods " (ways) of accessing table rows: tabular and index
- Access methods can be added by extensions:
  - `create extension pg_columnar ;`
  - `create extension bloom;`
- Table access methods define how data is stored in tables and typically read all rows.
- Index methods typically read a portion of the table blocks.
- For index methods (methods) you need to create an auxiliary object called an index
- Indexes are created on one or more columns of a table :

```
create table t (id int8, d date, s text);
create index t_idx on t using btree ( id int8_ops, d date_ops );
create index t_idx1 on t using btree ( s text_ops );
create index if not exists t _idx2 on t ( id , d );
```


String Access Methods

There are two types of " methods " (ways) to access table rows: tabular and index.

List of available access methods: **`\ dA`** or request:

```
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
```

Access methods can be added by extensions:

```
create extension pg_columnar ;
create extension bloom;
```

Extensions will be added to the `pg_am table` access methods:

```
 2425358 | columnar | columnar.columnar_handler | t
 2425512 | bloom | blhandler | i
```

Table access methods define how data is stored in tables. In order for the planner to use an index access method, you must create a helper object called an index . **" Index type " and " index access method " are synonyms** .

Indexes are created on one or more columns of a table:

```
create table t(id int8, s text);
create index t_idx on t using btree (id int8_ops) include (s) with ( fillfactor
= 90, deduplicate_items = off);
```

When creating an index, you specify the table name and the column or columns (a " composite index ") whose values will be indexed. Option INCLUDE allows you to store column values in the index structure, expressions cannot be used. Operator classes are not required for such column data types. The purpose of including columns is to force the planner to use Index Only Scan.

You can create multiple identical indexes, but with different names.

The operator class name is usually not specified because there is a default class for the column type. The default index type is btree .

https://docs.tantorlabs.ru/tdb/ru/16_4/se/sql-createindex.html

Operator class for index

- When creating an index, you can specify an operator class separately for each index column

```
create table t(id int8 , s text);
create index t_idx on t using btree (id int8_ops , s text_pattern_ops ) ;
```

- If you do not specify an operator class, the default class for the column type is used :

```
\ dA c + btree integer
List of operator classes
  AM | Input type | Storage type | Operator class | Default? | Operator family | Owner
-------+-----------+--------------+----------------+----------+-----------------+----------
 btree | integer | | int4_ops | yes | integer_ops | postgres
```

- list of functions , used by families of operators :

```
\ dA p + 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 | btint8 cmp ( bigint,bi ..
btree | integer_ops | bigint | bigint | 2 | btint8 sortsupport (in..
```


Operator class for index

When creating an index, you can specify an operator class for each index column. In the class includes operators that will be used by the index for comparisons, sorting, and ordering of column values. If you do not specify an operator class, the default operator class for the column type and access method is used. Multiple classes for the same data type allow you to perform calculations and order data differently.

Operator classes for similar types (`int8, int4`) are included in an operator family. Operators are a way to write an expression more compactly than using functions. When creating an operator, the name of the function is specified. For example, " + " is equivalent to `the sum ( a,b ) function` . Functions are data processing algorithms. Algorithms can be universal: work with different data types. In order not to create a large number of functions, universal functions are created. Based on these functions, " universal " operators are created that work with many data types. Operators are combined into operator families. Operator families allow you to create execution plans with expressions of different types without using explicit type casting.

List of functions, used by families of operators : **\ dA p + 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 | btint8 cmp ( bigint,bi ..
btree | integer_ops | bigint | bigint | 2 | btint8 sortsupport (in..
btree | integer_ops | bigint | bigint | 3 | in_range ( bigint , ..
btree | integer_ops | bigint | bigint | 4 | bt equalimage ( oid )
```

List of classes default : **\ dA c + btree integer**

```
List of operator classes
  AM | Input type | Storage type | Operator class | Default? | Operator family | Owner
-------+-----------+--------------+----------------+----------+-----------------+----------
 btree | integer | | int4_ops | yes | integer_ops | postgres
...
```

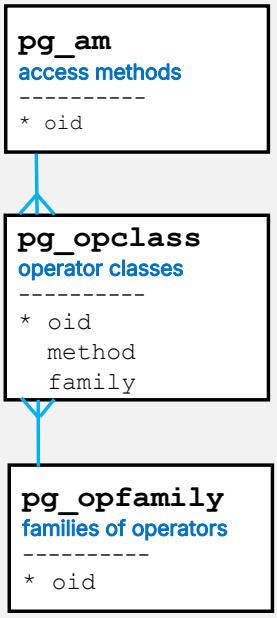
A request similar to the command **\ dAc +** :

```
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
ORDER BY 1, 2, 4;
```

In the `text_pattern_ops class` Other operators are included that implement pattern and regular expression searches.

Families and classes of operators

- The operator class binds operators that will play roles (strategies) in the methods that the index logic uses to order (compare, sort, measure distances, associate) data. The operator class specifies the names of the " supporting " functions that the index method will use to search or sort data.
- list of operators:




```
\ dAo btree
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
...
```


Families and classes of operators

An operator class is created for an access method and data type ; it has functions implementing the operators that the method supports. An operator class is associated with a data type and index type, and can optionally be included in an operator family. If you do not specify a family, the class is placed in a family with the class name. **If there is no family with such a name, it is created** . It may seem that index types (access methods) are associated (depend on availability) with families:

List of operators: **\ dAo btree**

```
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
...
```

vice versa: **an operator family is associated (depending on the presence) with an access method. Access methods are associated only with operator classes** .

Operator families allow you to create execution plans with expressions of different types without using explicit type casts.

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. The operator class specifies the names of the " supporting " (supporting , (supporting) functions that will use the index method when searching or sorting data.

Operators and support functions are defined in an operator class when it is created and the set does not change (strong coupling). If an index is created that uses an operator class, the class cannot be dropped without dropping the index. Operators and functions that are required for an index are included in the operator class, not the family. Operators and functions can be added to and removed from the family (weak coupling). When operators and support functions are added to a family using ALTER OPERATOR FAMILY, they do not become part of any operator class in the family, but are considered " loosely coupled " to the family. These operators and functions are compatible with the family, but are not required for the correct operation of the indexes .

https://docs.tantorlabs.ru/tdb/ru/16_4/se/sql-createopclass.html

Lists of strategies: https://docs.tantorlabs.ru/tdb/ru/16_4/se/xindex.html

Support functions for the index

- For indexing using the btree method :
  - it is enough that the operator class contains a function (BTORDER_PROC) compare (compare) two values
  - for efficient sorting (ORDER BY) it is desirable to have a second function for quick sorting of values (BT SORTSUPPORT _PROC)
  - to allow the planner to use the index in the "RANGE" expressions of window functions need third function (BT INRANGE _PROC)
  - To support deduplication, a fourth function is needed (BT EQUALIMAGE _PROC).
- List of functions :

```
\ dA p + 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 | btint8 cmp ( bigint,bi ..
btree | integer_ops | bigint | bigint | 2 | btint8 sortsupport (in..
btree | integer_ops | bigint | bigint | 3 | in_range ( bigint , ..
btree | integer_ops | bigint | bigint | 4 | bt equalimage ( oid )
```


Support functions for the index

btree is the most commonly used index type in DBMS. For indexing by the method btree it is enough that the data type is comparable. For this purpose, the operator class has a supporting (" reference ") function (BTORDER_PROC) that could compare two values of the data type for which the operator class was created. The result of the function is a negative value, a positive value, or zero if the values are equal.

For efficient sorting (ORDER BY) it is desirable that the operator class have a second (Number 2) function for quickly sorting values (BT SORTSUPPORT _PROC).

To enable the planner to use the index in the "RANGE" expressions of window functions the third (Number 3) function (BT INRANGE _PROC) is needed.

To support deduplication the fourth (Number 4) function (BT EQUALIMAGE _PROC) is needed.

List of functions, used by families and classes of operators of the same name :

```
\ dA p + btree integer_ops
List of support functions of operator families
 AM | Operator family | Registered|Registered| Number | Function
      |                 | left type|right type|
-----------+-----------------+---------+----------+--------+-----------------
btree | integer_ops | bigint | bigint | 1 | btint8 cmp ( bigint,bigint )
btree | integer_ops | bigint | bigint | 2 | btint8 sortsupport (internal)
btree | integer_ops | bigint | bigint | 3 | in_range ( bigint,bigint , bigint,boolean,boolean )
btree | integer_ops | bigint | bigint | 4 | bt equalimage ( oid )
...
btree | integer_ops | smallint | integer | 1 | btint24 cmp ( smallint,smallint )
btree | integer_ops | smallint | integer | 3 | in_range ( smallint,smallint,smallint,smallint )
.. .
btree | text_ops | text | text | 1 | btnametext cmp ( text,text )
...
btree | text_pattern_ops | text | text | 4 | bt equalimage ( oid )
...
( 22 rows)
```

When an operator class is created, it is included in a family with the same name. If there is no family with such a name, it is created. Therefore, **the names of families and classes coincide** .

Indexes for integrity constraints

- For PRIMARY KEY and UNIQUE integrity constraints , **a unique btree** index is required on the columns that are part of the integrity constraint.
- other indexes can be used to speed up queries (" analytical indexes "), full-text search
- indexes speed up row searching and slow down adding, changing, and deleting rows
- indexes use disk space, size is comparable to table size
- example of replacing an index with another index:

```
create table t3 (n int4 primary key, m int4);
Indexes:
"t3_pkey" PRIMARY KEY, btree ( n )
create unique index concurrently t3_pkey1 on t3 ( m,n );
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"
Indexes:
" t3_pkey " PRIMARY KEY, btree ( m, n )
```


Indexes for integrity constraints

If you do not specify an index type in the CREATE INDEX command , an index of type btree is created . btree the most common type of index in relational databases, working with many types of data.

PRIMARY KEY (PK) and UNIQUE (UK) integrity constraints, btree indexes are required . For other integrity constraints, they are optional and are created if: they speed up queries, do not significantly slow down data modification , and the space used by the indexes is not critical .

When creating PRIMARY KEY (PK) and UNIQUE (UK) integrity constraints, unique btree indexes are created . The rules for using indexes with integrity constraints differ from Oracle Database.

For example, in PostgreSQL without a unique PK constraint index and UK cannot exist:

```
ERROR: PRIMARY KEY constraints cannot be marked NOT VALID
```

and cannot use non-unique indexes:

```
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;
```

In Oracle Database there is an enabled and disabled state of integrity constraints, an index is created when an integrity constraint is enabled , and non-unique indexes can be used. Such differences do not provide advantages or disadvantages, but it is useful to know about the differences when operating and maintaining tables if you have experience working with DBMSs other than PostgreSQL .

In PostgreSQL only index type btree supports the UNIQUE property (can be unique) :

```
select amname , pg_indexam_has_property (a.oid, ' can_unique ') as p from pg_am a where amtype = ' i ' and pg_indexam_has_property (a.oid, ' can_unique ') = true order by 1;
 amname | p
--------+---
 btree | t
```

Indexes can be used to speed up queries ("analytical indexes"), full-text search. Indexes speed up the search for rows and slow down the addition, modification, and deletion of rows. The size of indexes is comparable to the size of the table.

You can create a new index and assign it to replace the old one, but it must be unique:

```
create table t3 (n int4 primary key, m int4 );
\d t3
Table "public.t3"
Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+---------
n | integer | | not null |
m | integer | | |
Indexes:
"t3_pkey" PRIMARY KEY, btree ( n )
create unique index concurrently t3_pkey1 on t3 ( m,n );
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"
\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 )
```

The old index was dropped. The integrity constraint columns are defined by the index columns. You cannot drop an index without dropping the PRIMARY KEY integrity constraint .

PRIMARY KEY differs from UNIQUE in that it is unique per table and adds NOT NULL integrity constraints. on all columns. In the example, not null appeared on both columns .

When you drop a PRIMARY KEY , NOT NULL constraints are not dropped because they may have been added separately.

A FOREIGN KEY integrity constraint can only be created on columns with a PRIMARY KEY or UNIQUE , otherwise an error will be returned :

```
ERROR: there is no unique constraint matching given keys for referenced table "t3"
```

When adding an integrity constraint :

```
alter table t3 add constraint fk foreign key (m) references t3(n) not valid;
\d t3
Table "public.t3"
Column | Type | Collation | Nullable | Default
--------+---------+-----------+----------+---------
n | integer | | not null |
m | integer | | |
Indexes:
"t3_pkey" PRIMARY KEY, btree (n)
Foreign key constraints:
" fk " FOREIGN KEY (m) REFERENCES t3(n) NOT VALID
Referenced by:
TABLE "t3" CONSTRAINT " fk " FOREIGN KEY (m) REFERENCES t3(n) NOT VALID
alter table t3 validate constraint fk ;
```

ShareRowExclusive lock requested during row validation on both tables (with FOREIGN KEY and the key it refers to). In the example, there is one table (self -referencing key). When adding other types of integrity constraints, AccessExclusive is usually requested .

Do I need to create an index on the FOREIGN KEY column? If the table with FK has few rows and a full scan is faster than an index scan, then it is not necessary. An index on FK is created if:

1) in the master table (where PK is) the PK column value is frequently updated or rows are deleted. These actions are undesirable and are avoided when designing applications.

PK-FK table joins . This is used very often, because that is what FK is for: it defines **the connection (join)** between tables. For example:

```
select master.m , detail.n from t3 master join t3 detail on ( master.n = detail.m ) ;
```

https://www.cybertec-postgresql.com/en/index-your-foreign-key/

btree index

- For a numeric column (int2, int4, int8) in the internal (intermediate) block at 70% filling, 286 links fit, at 100%, 407 links fit
- If you specify more than one column in the index:
  - the order of indexed columns is important
  - one can imagine that the result of concatenating the column values is indexed
  - the first column is called the leading column
  - the index is called composite
- The index record must fit into 1/3 of the block (2704 bytes) :

```
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 ));
ERROR: index row size 2720 exceeds btree version 4 maximum 2704 for index " t_idx "
DETAIL: Index row references tuple (0,2) 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.
```


btree index

For a numeric column (int2, int4, int8) in the intermediate (internal) block at 70% filling, 286 links fit, at 100% filling, 407 links fit. The number of index levels (level starting from zero) per numeric column with monotonous filling by sequence will not exceed 5: 286 ^ 5 = 1913507486176 references to rows. If the row size is 18 bytes, then this is 32 TB, which is the maximum table size. **For 4 levels, the number of rows is 6690585616** .

To search for a row in the index, the process reads at least a level+1 block and possibly one or more leaf blocks to the right. If Index Only Scan is not used or the block may contain (checked by the vm visibility map) irrelevant rows, then it will also have to read the table block. Each block reading is a search by the Buffer Mapping Table , block pinning, and an increase in usagecount . Therefore, adding a new level to the index reduces performance. If rebuilding the index reduces the number of levels, then the performance of index search (index access) increases.

The intermediate index blocks store the value of at least the first indexed column. If the values are long, the number of records in the intermediate blocks may be large, the number of levels will be large, and the size of the index will also be large. Therefore, **it is not advisable to index the first columns with long values .** The suffix truncation optimization may exclude column values from non-leaf blocks. INCLUDE column values are present only in leaf blocks.

The index record must fit into 1/3 of the block (2704 bytes) :

```
create table t (id int8, s text storage plain);
create index t_idx on t ( s text_ops ) include (id) ;
insert into t values (repeat('a', 2700 ));
ERROR: index row size 27 20 exceeds btree version 4 maximum 2704 for index " t_idx "
DETAIL: Index row references tuple (2,1) 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.
```

Pageinspect extension functions for btree

- `bt_metap ( relname )` gives out information from the index metadata block. This is always the first block of the first index file (block number zero)
- `bt_page_stats (.. )` and `bt_multi_page_stats (..)` numbers of neighboring blocks to the left (btpo_prev) and right (btpo_next) at the same level

```
CREATE TABLE t(s text storage plain) with ( autovacuum_enabled =off, fillfactor =40);
create index t_idx on t (s);
INSERT INTO t VALUES (repeat('a',2500));INSERT INTO t VALUES (repeat('b',2500));
INSERT INTO t VALUES (repeat('c',2500));INSERT INTO t VALUES (repeat('d',2500));
select * from bt_multi_page_stats ('t_idx',1,-1);
 blkno|type|live_items|dead_items|avg_item_size|pagesize|freesize|btpo_prev|btpo_next|btpo_level|btpo_flag
s
 -----+----+----------+----------+-------------+--------+--------+---------+---------+----------+---------
-
 1 | l | 2 | 0 | 2512 | 8192 | 3116 | 0 | 2 | 0 | 1
 2 | l | 3 | 0 | 2512 | 8192 | 600 | 1 | 0 | 0 | 1
 3 | r | 2 | 0 | 1260 | 8192 | 5620 | 0 | 0 | 1 | 2
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , tids , data from bt_page_items
('t_idx',1);
  itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | substring
 ------------+-------+---------+-------+------+------+-------+------+-----------
 1 | (1,1) | 2512 | f | t | | | | 20 27 00 00 62 62 62 62
 2 | (0,1) | 2512 | f | t | f | (0,1) | | 20 27 00 00 61 61 61 61
```


Pageinspect extension functions for btree

To view index data, use the standard pageinspect extension :

`create extension if not exists pageinspect ;`

The list of extension functions can be obtained using the command : **`\ dx + pageinspect`**

`bt_metap ( relname )` gives out information from the index metadata block. This is always the first block of the first index file (block number zero).

`bt_page_stats ( relname , blkno )` takes an index name and a block number, and produces one line:

1) type index block type: **`l`** leaf **sheet** , **`i`** internal **internal** (between root and leaf) , **`r`** **root** root , d deleted leaf (free) , D deleted internal (free) , e ignored

2) `avg_item_size` - average size of a record in a block

3) `free_size` how many bytes are free in the block

4) `btpo_prev btpo_next` numbers of blocks to the left and right (siblings) at the same level. Zero means that the block is the outermost at its level. In leaf blocks, they are used for navigation when searching by range, sorting. `btpo_next` is requested by the server process to find the correct leaf block in case the block split while the process was descending to it from the upper level. To check for such a situation, the first record of a leaf or intermediate block (except for the rightmost at its level) is used, which always stores the value of key columns greater than or equal to which there is no value in this block (called High key).

By `btpo_prev , btpo_next , bt_page_items () .ctid you` can navigate the index tree and even draw the tree.

5) `dead_items` - the number of "killed" (LP_DEAD flag for the index entry) entries as a result of simple deletion index block

6) `live_items` - number of used entries in the index

`bt_multi_page_stats ( relname , blkno , blk_count bigint )` produces the same as the previous function, only for the range of index blocks. A negative number in `blk_count` means the last page. `blkno` is the block number from which to produce data.

`bt_page_items ( relname , blkno )` returns the contents of the index block entries in a human-readable form.

https://docs.tantorlabs.ru/tdb/ru/16_4/se/pageinspect.html#PAGEINSPECT-B-TREE-FUNCS

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.

If the block is sheet and tids not empty, then deduplication and tids are used stored in sorted (by block) and slot) ctid link format on the rows in the table. In htid the first tid is saved from tids . ctid at the same time, it stores not references to blocks, but service data about tids . For example, ctid of intermediate blocks will store the block number in the index, referring to the lower level, and the second part of ctid - the number of elements in tids .

If tids empty, then deduplication is not used in this index entry. In ctid leaf block stores a reference to the indexed row of the table. The htid (heap tuple id) field stores the same value as ctid . Why is the value duplicated in ctid and htid ? Btree operation algorithm optimized for operation under conditions of minimum blocking and to minimize block splits. During the process of reading index blocks, other processes can change its structure. When navigating through blocks, there is no single picture (" read integrity ") and additional fields are used to detect contradictions.

In intermediate blocks in `ctid` the reference to the index block is stored, 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` empty (interpreted as " minus infinity " , that is, the boundary is unknown) and this link leads to the leftmost child block .

The index uses the suffix truncation optimization and 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 remains an empty space, which is treated as minus infinity .

data field of the current and next (itemoffset+1) records specifies the range that must contain the value by which the search is performed in the index :

```
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , tids , data from bt_page_items ('t_idx',1);
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | substring
------------+-------+---------+-------+------+------+-------+------+-----------
1 | (1,1) | 2512 | f | t | | | | 20 27 00 00 62 62 62 62
2 | (0,1) | 2512 | f | t | f | (0,1) | | 20 27 00 00 61 61 61 61
```

data value is included in the range and is the first value in the block the record points to. The next record 's data value is not included in the range. The next record is the first record in its range.

The sizes of indexes can be obtained with the command **\ di + * index_name *** .

For a more detailed selection, you can use the following query as a template:

```
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 ;
table| index | indexes_size|table_size | total | index_size |rows | ddl
----------+----------+----------+-----+----------+---------+--------+--------------------
test | test_id_idx | 6876692480|11097309184|17977090048|6876692480|299916064| CREATE INDEX test_id_idx
ON public.test USING btree (id)
t4 |t4_pkey | 679936| 1187840| 1892352| 679936| -1 |CREATE INDEX t4_pkey
ON public.t4 USING btree (id) WITH ( fillfactor ='100')
```

In the column indexdef pg_indexes views The index creation command is stored .

If rows=-1 This means that the table does not have statistics. You can collect statistics using the ANALYZE command.

https://raw.githubusercontent.com/postgres/postgres/refs/heads/master/src/backend/access/nbtree/README

Indexes with deduplication in leaf blocks

- if the index is not unique, then with a large number of duplicates they are stored compactly due to deduplication in leaf blocks
- Not all data types support deduplication.
  - do not support: `numeric, jsonb , float4, float8`
  - do not support: arrays, composite, range types
  - INCLUDE indexes do not support deduplication
- The values of indexed columns are stored in index entries, and references to table rows are stored in d in the tid column (tuple id , table row identifier) as a sorted array ctid
- example of two records with deduplication and one without deduplication .

```
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , data, tids [0:3] from bt_page_items
('td_idx',1);
 itemoffset | ctid | itemlen|nulls|vars|dead|htid | data | tids
----------+---------+-------+-----+----+----+-------+----------------------+-------------------------
1 |(16.8414)| 1352 | f | f | f |(0,1) |01 00 00 00 00 00 00 00| {" (0.1) "," (0.2) "," (0.3) "
2 |(16.8377)| 1128 | f | f | f |(10,13)|01 00 00 00 00 00 00 00| {" (10,13) "," (10,14) "," (1
        3 | (19.9) | 16 | f | f | f |(19.9) |01 00 00 00 00 00 00 00|
```


Indexes with deduplication in leaf blocks

If the index is not unique, then with a large number of duplicates they are stored compactly due to deduplication in leaf blocks. Not all data types support deduplication . To support deduplication , function number 4 (BTEQUALIMAGE_PROC) must be defined in the operator class for the data type being indexed .

You can check whether deduplication is supported in the index using the following query:

```
select * from bt_metap (' t_idx ');
magic|version|root|level|fastroot|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples| allequalimage
----------+-------+----+-----+- - ------+---------+----------+----------------------+-----------
340322| 4 | 3 | 1 | 3 | 1 | 0 | -1 | t
```

If allequalimage = t that is supported . Deduplication appeared in PostgreSQL version 13.

The same query shows the number of index levels: level=1. The numbering of levels starts from zero. The magic and version fields are used to check that the object is a btree index. supported version. Starting with PostgreSQL 1 2 version 4 index version is used, minimally supported in PostgreSQL 17 version version= 2. If the version is less than 4 , then the index can be used, but the innovations are not supported. To support innovations it is enough to rebuild the index (REINDEX) . " Magic " number 340322 (0x0531162). The value of the number is chosen randomly, it should be in the zero block at the " correct " offset.

The values of indexed columns are stored in records index, and references to table rows are stored in the tid s column (tuple ids , table row identifiers) as a sorted array values of type ctid .

```
drop table td; create table td(id serial) with ( autovacuum_enabled =off); create index td_idx on td(id);
insert into td select 1 from generate_series (1, 408 );
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , data, tids [0:3] from bt_page_items
('td_idx',1);
select * from bt_multi_page_stats ('td_idx',1,-1);
 itemoffset | ctid | itemlen|nulls|vars|dead|htid | data | tids
----------+---------+-------+-----+----+----+-------+----------------------+-------------------------
1 |(16.8414)| 1352 | f | f | f |(0,1) |01 00 00 00 00 00 00 00| {" (0.1) "," (0.2) "," (0.3) "
2 |(16.8377)| 1128 | f | f | f |(10,13)|01 00 00 00 00 00 00 00| {" (10,13) "," (10,14) "," (1
        3 | (19.9) | 16 | f | f | f | (19.9) |01 00 00 00 00 00 00 00|
```

In line itemoffset = 3 no deduplication (tids empty) and the table row is pointed to by ctid = (19,9) . tids point to the table rows .

Deduplication occurs when there is no space in a block or the fillfactor is exceeded .

https://docs.tantorlabs.ru/tdb/ru/16_4/se/btree-implementation.html#BTREE-DEDUPLICATION

Check if deduplication is supported

- deduplication is not supported with data types: `numeric , jsonb , float4, float8` , arrays, composite, range types
- Indexes with INCLUDE columns do not support deduplication
- Examples of a query to check if deduplication is supported:

```
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 ');
 allequalimage
---------------
 t
create index td1_idx on td(n) include (n1);
select allequalimage from bt_metap ('td1_idx');
 allequalimage
---------------
 f
create table td(id int8 [] );
create index td_idx on td(id);
select allequalimage from bt_metap (' td_idx ');
 allequalimage
---------------
 f
```


Check if deduplication is supported

You can check whether deduplication is supported in the index using the following query:

```
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 ');
 allequalimage
---------------
t
```

If `allequalimage = t` then deduplication is supported. A composite index supports deduplication if the key data types support it.

```
create index td1_idx on td(n) include (n1);
select allequalimage from bt_metap ('td1_idx');
 allequalimage
---------------
 f
```

Indexes with INCLUDE columns do not support deduplication even if the data types do.

```
create table td(id int8 [] );
create index td_idx on td(id);
select allequalimage from bt_metap (' td_idx ');
 allequalimage
---------------
 f
```

Index creation parameters and their impact on performance

- by default the index is built in ascending order
- When creating an index, you can specify the reverse order: DESC
- Right blocks are optimized for inserts
- by default empty values are stored in the right blocks
- inserting rows with NULLs will be slower when using NULLS FIRST
- vacuum reads all index blocks in physical order from first to last block
- include column values in an index without including them in key values increases the size of the index

```
create unique index concurrently if not exists t1_idx1 ON t1 using btree
(c2 desc nulls first , upper(c1) ) include (c3,c4) with ( fillfactor =100,
deduplicate_items =off) WHERE c2>0 ;
```


Index creation parameters and their impact on performance

The leaves of the tree repeat the values of the rows of the original table. They are presented in ascending or descending order.

By default, the index is built in ascending order, i.e. smaller values are on the left, larger values are on the right. Leaf blocks and blocks of each level have pointers to neighboring blocks in both directions.

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 predominantly in the right leaf block of the index. " Right blocks " always remain " right " in the index structure. regardless of ASC or DESC .

By default, empty values are saved. on the right, in the " right blocks " . This can be overridden. having indicated NULLS FIRST . When overriding, it is usually assumed that the default is to output first when sorting. However, using NULLS FIRST may 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, so as NULL will be in the leftmost leaf block, and fastpath works only with right** . Inserting rows with NULLs will be slower when using NULLS FIRST .

" fastpath " optimization is similar to " fastroot " , but these are different optimizations.

You can index not only columns, but also expressions . This is a useful option, but you should not create a large number of indexes.

You can include column values in an index without including them in key values. You cannot include expressions , only columns. The include option increases the size of the index. Columns are included to make the index " covering " queries - so that Index Only Scan is used , but the column cannot be included in the key columns, since the column data type does not support the operator class. index.

```
create unique index concurrently if not exists t1_idx1 ON t1 using btree (c2
desc nulls first , upper(c1) ) include (c3,c4) with ( fillfactor =100,
deduplicate_items =off) WHERE c2>0;
```

Partial indices

- are created based on a portion of the table rows
- The WHERE predicate is specified when creating an index and determines the rows to be indexed.
- are useful because they allow you to avoid indexing the most frequently occurring values
- partial index can be unique
- the size of a partial index is usually smaller
- Example of creating a partial index:

```
create unique index t1_idx1 ON t1 (c2 desc nulls first , upper(c1))
include (c3,c4) WHERE c2>0 ;
```


Partial indices

Partial indexes are created on a part of the table rows. The part of the rows is defined by the WHERE predicate, which is specified when creating the index and makes the index partial .

The index size can be significantly reduced and vacuuming will be faster, since vacuuming scans all index blocks. Partials can be created indexes. This is useful if the application does not work with unindexed rows . When creating an index, you can specify a WHERE condition . The size of the index can be significantly reduced and vacuuming will be faster, since vacuuming scans all index blocks.

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, and 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.

https://docs.tantorlabs.ru/tdb/ru/16_6/se/indexes-partial.html

Evolution of indexes: creation, deletion, rebuilding

- `create /drop/ reindex` commands `index index_name` sets a SHARE lock that is incompatible with making changes to table rows
- these commands can be executed simultaneously, they are compatible with themselves, but are not compatible with concurrently
- autovacuum is not compatible with either `concurrently` , not without
- for temporary indexes on temporary tables, there is no need to use concurrently , since there are no locks on temporary objects
- `create/ reindex concurrently` scans the table twice , without `concurrently` once
- `concurrently` allows the execution of `SELECT, WITH, INSERT, UPDATE, DELETE, MERGE commands` and allows using the fastpath for blocking objects (tables, indexes, sections)

Evolution of indexes: creation, deletion, rebuilding

Creating, deleting, rebuilding an index without specifying CONCURRENTLY:

```
create index name ..;
drop index index_name ;
reindex index index_name ;
```

SHARE lock , incompatible with making changes to table rows. The SHARE lock only allows the following commands to work:

1) SELECT and any query that only reads the table (i.e. sets an ACCESS SHARE lock)

2) SELECT FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, FOR KEY SHARE (set ROW SHARE lock)

3) CREATE/DROP/REINDEX INDEX (**without** CONCURRENTLY). **You can simultaneously create , drop, and rebuild multiple indexes on a single table** because the SHARE lock is compatible with itself. CONCURRENTLY is not compatible with SHARE .

" Not compatible " means that either the command will wait, or will fail immediately, or will fail after the timeout specified by `the lock_timeout parameter` .

For temporary indexes on temporary tables, you do not need to use CONCURRENTLY , since there are no locks on temporary objects, only one process has access to them, even parallel processes do not have access.

`create index` **`concurrently`** `name ..;` sets SHARE UPDATE EXCLUSIVE lock , which allows execution of SELECT, WITH, INSERT, UPDATE, DELETE, MERGE commands and enables use of the fastpath for locking objects by processes .

SHARE UPDATE EXCLUSIVE lock is also set by the DROP INDEX CONCURRENTLY commands, REINDEX CONCURRENTLY , as well as VACUUM (without FULL), ANALYZE, CREATE STATISTICS, COMMENT ON , some types ALTER INDEX and ALTER TABLE , autovacuum and autoanalysis . These commands cannot work with one table at a time . Autovacuum skips tables if it cannot immediately obtain a lock. **Autovacuum is incompatible with creating, deleting, recreating indexes.**

CONCURRENTLY has a significant drawback. Without CONCURRENTLY the table is scanned once, with CONCURRENTLY the table is scanned twice and three transactions are used.

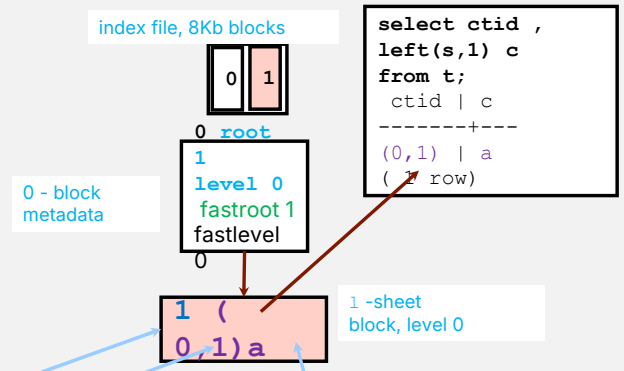
CONCURRENTLY waits for all existing transactions that could potentially modify and use the index to complete. The first transaction creates the index definition and adds it to the system catalog as invalid . Next, the process waits for all transactions that modified the table during the first transaction to complete. The second transaction begins, in which the table is scanned and the index structure is built. Next, the process waits for all transactions that modified the table during the second transaction to complete. The third transaction begins, in which the table is scanned again and the index structure is updated. The process waits for all transactions that received a snapshot before the second transaction to complete.

If a problem occurs while scanning a table, such as a deadlock or a uniqueness violation in a unique index, the CREATE INDEX CONCURRENTLY command will fail and leave the index in an invalid state.

How do I recover from a failed command ? Either execute the REINDEX INDEX CONCURRENTLY command or delete the index with the DROP INDEX or DROP INDEX CONCURRENTLY command. After deleting the index, you can repeat the command or create the index without CONCURRENTLY.

btree index

- block types: metadata, root, internal, leaf
- the path from the root to the leaf blocks is of equal depth (balanced)
- in leaf blocks references to table rows (block, line in block)
- in non-leaf blocks references to index blocks (block, row in block)




```
create table t(s text storage plain) with ( fillfactor =10);
create index t_idx on t (s) with ( fillfactor =10);
insert into t values (repeat('a', 2500 ));
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , tids , left(data,24) data, chr
( nullif (('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items ('t_idx', 1
);
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data | c
------------+-------+---------+-------+------+------+-------+------+----------+---
          1 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61 | a
```


btree index

To visualize the structure of a btree index , let's create a table and an index:

```
drop table 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);
```

An index file with one block was created. The first (0) block of the index contains metadata :

```
select * from bt_metap (' t_idx ');
magic|version| root | level |fastroot|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples|allequalimage
-----------+-------+----+-----+--------+---------+-------------------------+-----------------------+------------
340322|  4 |    0 |     0 |        0 |         0 |                       -1 |                       | t
```

1) level - the number of levels in the index tree. Levels are numbered from zero and from leaf blocks because the index grows from the bottom up and a new level is added above the root block.

2) root - the number of the root block. The root block can change when the number of index levels increases

When inserting a row, block number 1 is added to the index . It is formally leaf (type= `l`) , there is no root or internal block:

```
insert into t values (repeat('a',2500));
select * from bt_page_stats ('t_idx', 1 );
blkno|type|live_items|dead_items|avg_item_size|pagesize|freesize|btpo_prev|btpo_next| btpo_level |btpo_flags
-----+----+ - --------+---------+-------------+--------+--------+---------+---------+----------+----------
    1 | l  |     1 |         0 |        2512 |   8192 |   5632 |       0 |       0 |        0 |          3
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , tids , left(data,24) data, chr ( nullif
(('0x0'||substring(data from 13 for 2))::integer,0)) c from bt_page_items ('t_idx',1);
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data | c
------------+-------+---------+-------+------+------+-------+------+----------+---
          1 | (0,1) |    2512 | f     | t    | f    | (0,1) |      | 20 27 00 00 61 61 61 61 | a
```

The letter " a " has a hexadecimal code of " 61 ".

The link to block number 1 is inserted into the root fields and fastroot metadata block:

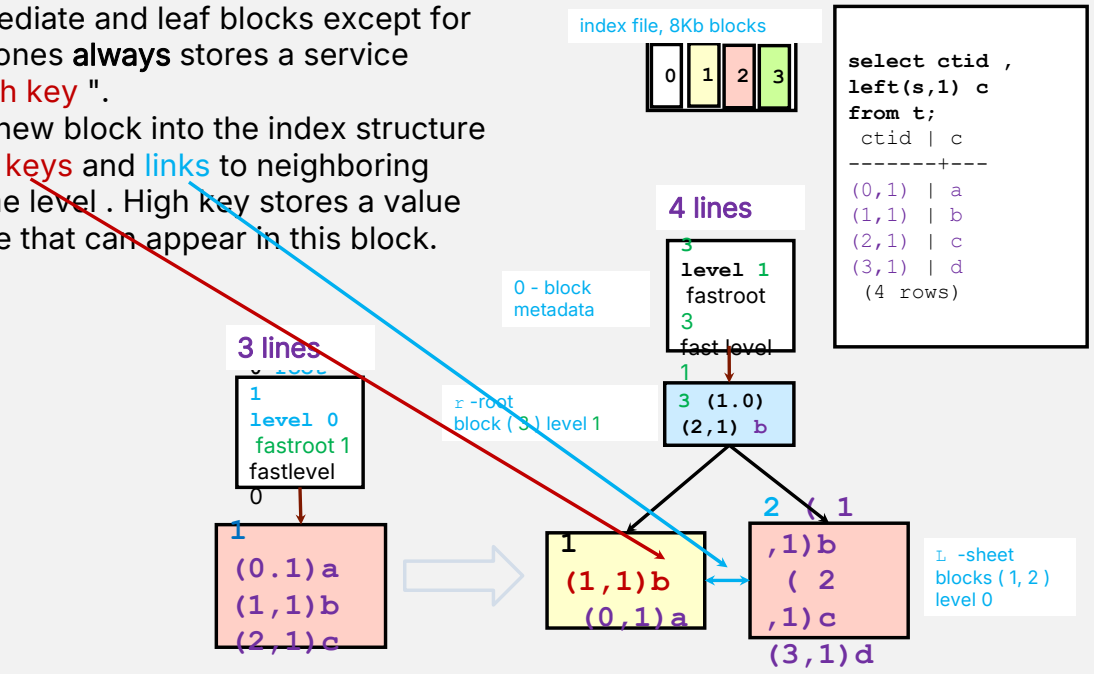
```
magic|version| root | level | fastroot
|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples|allequalimage
-----------+-------+----+-----+--------+---------+-------------------------+-----------------------+------------
340322|  4 |    1 |     0 |        1 |         0 |                       -1 |                       | t
```

Number of levels is currently 0 . Line length `itemlen = 2512` bytes.

Using itemlen you can evaluate whether it is worth indexing this column. In the example, the length is significant: the row is 2500 bytes in size and the index will be larger than the table.

High Key in index structure

- first line in intermediate and leaf blocks except for the very " right " ones **always** stores a service value called " High key ".
- when inserting a new block into the index structure are updated High keys and links to neighboring blocks of the same level . High key stores a value greater than those that can appear in this block.



High key in index structure

The second and third lines will fit into the same block:

```
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+-------+---------+-------+------+------+-------+------+-----------
1 | ( 0 ,1) | 2512 | f | t | f | (0,1) | | 20 27 00 00 61 61 61 61
2 | ( 1 ,1) | 2512 | f | t | f | ( 1 ,1) | | 20 27 00 00 6 2 6 2 6 2 6 2
3 | ( 2 ,1) | 2512 | f | t | f | ( 2 ,1) | | 20 27 00 00 6 3 6 3 6 3 6 3
```

In the ctid field index records indicate references to the address of the indexed row in the table files: (0 , 1) (1,1) (2, 1) - rows with values "a..", "b..", "c.." in 0 , 1 and 2 (block number in the main layer of the table) table blocks. Each **table** block contains one row; the length of a row in **a table block** is 2504+24 (row header in the table)=252 8 bytes:

```
select pg_column_size (s) from t limit 1;
2504
```

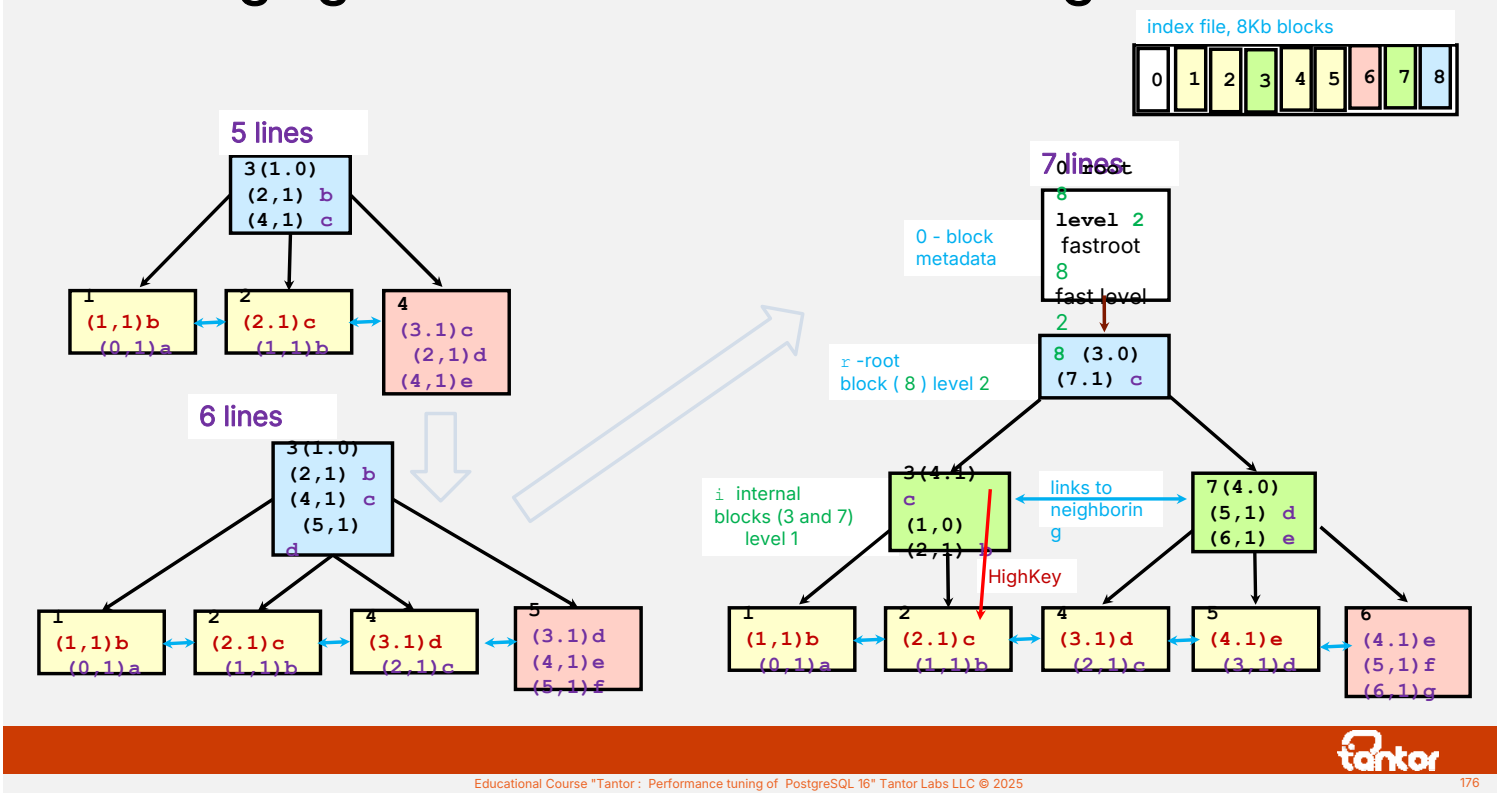
After inserting the fourth row, there will be three blocks in the index tree (plus the metadata block): 1,2,3. Blocks 1 and 2 will be leaf blocks, and 3 will be the root block. The references to the rows will be redistributed: the first leaf block will retain references to a :

```
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+-------+---------+-------+------+------+-------+------+-----------
  1 | ( 1 ,1) | 2512 | f | t | | | | 20 27 00 00 6 2 6 2 6 2 6 2
2 | ( 0 ,1) | 2512 | f | t | f | (0,1) | | 20 27 00 00 61 61 61 61
```

in the second sheet block references to lines b , c, d :

```
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
------------+-------+---------+-------+------+------+-------+------+-----------
1 | ( 1 ,1) | 2512 | f | t | f | ( 1 ,1) | | 20 27 00 00 6 2 6 2 6 2 6 2
2 | ( 2 ,1) | 2512 | f | t | f | ( 2 ,1) | | 20 27 00 00 6 3 6 3 6 3 6 3
3 | ( 3 ,1) | 2512 | f | t | f | ( 3 ,1) | | 20 27 00 00 6 4 6 4 6 4 6 4
```

The first line (itemoffset =1) in blocks **except the most " right " always** stores a service value called "High key". The root is " right " . When inserting a new block into the index structure are updated High keys and links to adjacent blocks of the same level . High key stores a value greater than those that can be found in this block. High key 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 descend and redistributed links to table rows, which means that the sought value is in the block (or even blocks if there were several splits) to the right of the block to which they descended. If the High key value of the block to which they went via the rightmost link **differs** from the HighKey value block from where you descended, then you need to move to the right along the sheet level and check just in case whether the desired values are there.



Changing index structure when adding rows

The index grows from the bottom up. There is not enough space in the leaf block, it is " divided " : an empty block is added, the rows are redistributed, a reference to the new block is inserted into the upper block. If there is no space in the upper block, it is " divided " . The index structure is returned by the pageinspect extension function :

```
select * from bt_ multi _page_stats ('t_idx', 1 , -1 );
blkno|type|live_|dead_|avg_item|page_|free_|btpo_|btpo_|btpo_|btpo_
| | items|items | _size| size| size| prev | next|level|flags
-----+----+-----+-----+--------+-----+-----+-----+-----+-----+----
   1 | l | 2 | 0 | 2512 |8192 |3116 | 0 | 2 | 0 | 1
2 | l | 2 | 0 | 2512 |8192 |3116 | 1 | 4 | 0 | 1
3 | i | 3 | 0 | 1677 |8192 |3104 | 0 | 7 | 1 | 0
4 | l | 2 | 0 | 2512 |8192 |3116 | 2 | 5 | 0 | 1
5 | l | 2 | 0 | 2512 |8192 |3116 | 4 | 6 | 0 | 1
6 | l | 3 | 0 | 2512 |8192 | 600 | 5 | 0 | 0 | 1
7 | i | 3 | 0 | 1677 |8192 |3104 | 3 | 0 | 1 | 0
   8 | r | 2 | 0 | 1260 |8192 |5620 | 0 | 0 | 2 | 2
```

`btpo_prev` stores the block number to the left of the current one (blkno) at the same level. Value 0 means that there is no block on the left, this block is the leftmost one.

`btpo_next` The block number to the right of the current block. Value 0: the current block is the rightmost.

`type` block type : r - root ; i - internal ; l - leaf , e - ignored , d - deleted leaf , D - deleted internal

btpo_flags bitmap:

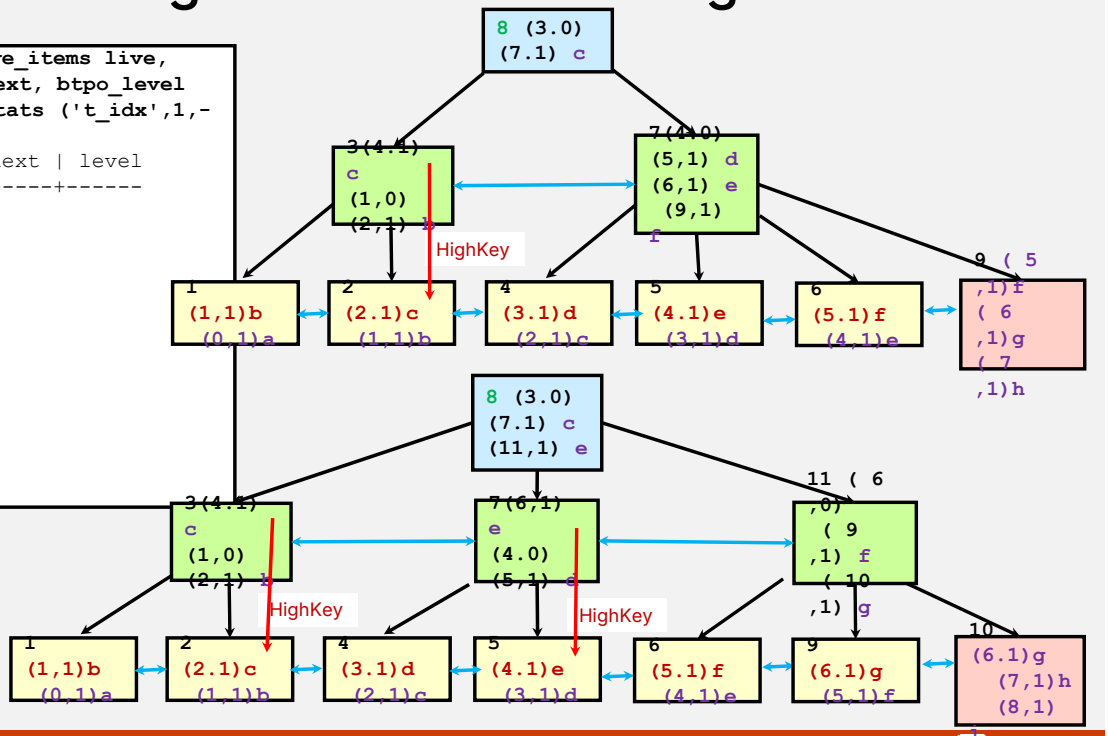
1 - leaf block, 2 - root block, 4 - free block, was removed from the index structure , 8 - metadata block, 16 - empty block, but in the tree structure (half-dead) . The remaining bits are used during vacuuming to track changes in the index structure : 256 is the deleted block flag (BTDeletedPageData).

It can be seen that the block numbers correspond to the order in which the block is added to the index structure. The root block changes as the number of levels increases (level) .

The index grows from the bottom up. There is not enough space in a leaf block, it is "divided": an empty block is added, the rows are redistributed, a reference to the new block is inserted into the block above. If there is no space in the block above, it is also "divided". Why is the word "divided" used? Because the rows in the block are redistributed between it and the one added to the index structure.

Example of index growth when inserting rows

```
 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);
 blk | type | live | prev | next | level
 ----+------+------+------+------+------
 1 | l | 2 | 0 | 2 | 0
 2 | l | 2 | 1 | 4 | 0
 3 | i | 3 | 0 | 7 | 1
 4 | l | 2 | 2 | 5 | 0
 5 | l | 2 | 4 | 6 | 0
 6 | l | 2 | 5 | 9 | 0
 7 | i | 3 | 3 | 11 | 1
 8 | r | 3 | 0 | 0 | 2
 9 | l | 2 | 6 | 10 | 0
 10 | l | 3 | 9 | 0 | 0
 11 | i | 3 | 7 | 0 | 1
 (11 rows)
```



Example of index growth when inserting rows

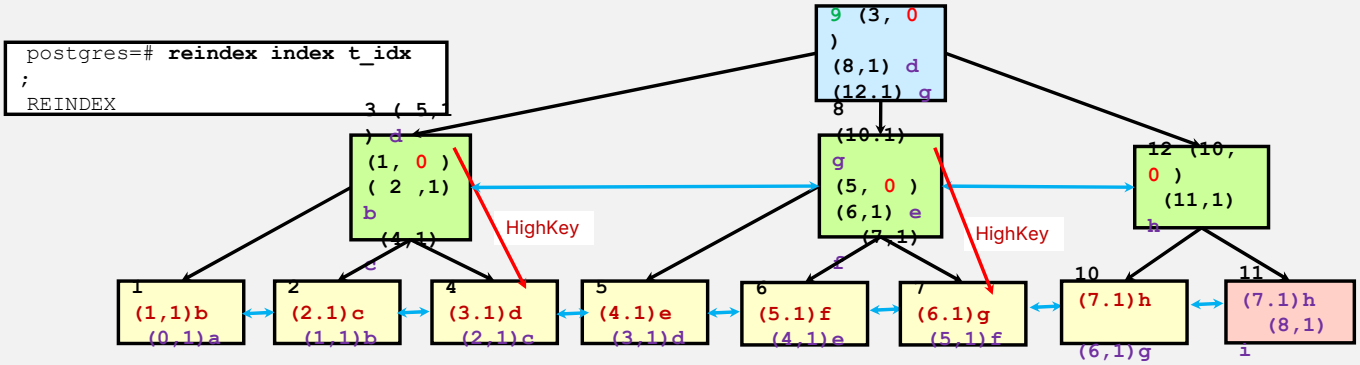
The content of the index blocks is shown by the query , an example is given for 9 rows:

```
 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 ', block_number );
```

```
 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
 ---+-------+---------+-------+--------------------+---
 1 | (2,1) | 2512 | | 20 27 00 00 63 63 | c
 2 | (1,1) | 2512 | (1,1) | 20 27 00 00 62 62 | b
 ---+-------+---------+-------+--------------------+---
 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
 ---+-------+---------+-------+--------------------+---
 1 | (3.1) | 2512 | | 20 27 00 00 64 64 | d
 2 | (2,1) | 2512 | (2,1) | 20 27 00 00 63 63 | 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
 ---+-------+---------+-------+--------------------+---
 1 | (5,1) | 2512 | | 20 27 00 00 66 66 | f
 2 | (4.1) | 2512 | (4.1) | 20 27 00 00 65 65 | e
 ---+-------+---------+-------+--------------------+---
 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
 ---+-------+---------+-------+--------------------+---
 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
 ---+-------+---------+-------+--------------------+---
 1 | (6,1) | 2512 | | 20 27 00 00 67 67 | g
 2 | (5,1) | 2512 | (5,1) | 20 27 00 00 66 66 | f
 ---+-------+---------+-------+--------------------+---
 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
 ---+-------+---------+-------+--------------------+---
 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
```

The structure of the index after its rebuilding

- The HighKey of the parent block must be present in the rightmost lower block referenced by the parent block. If the value is different, the process must read the lower blocks to the right of the rightmost one, since it is likely no longer the right one.
- in the rightmost blocks at their level HighKey is not stored, since there are no blocks to the right of them and checking is not necessary
- index rebuild command:



The structure of the index after its rebuilding

Contents of index blocks after rebuilding with the command:
`reindex index t_idx ;`
The example on the slide is given for 10 rows with values: `a,b,c,d,e,f,g,h,i .`
The red arrows show the HighKey connection . The HighKey value of the upper block must be present in the rightmost lower block referenced in the upper block. If the value is different from c я, the process must read the lower-level blocks to the right of the rightmost one, since it is most likely no longer the right one.
In the rightmost blocks on their own HighKey level are not stored, since there are no blocks to the right of them and checking is not necessary.
Index files may include blocks that are excluded from the index structure. 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 autovacuum and reduces the number of empty index blocks that autovacuum is forced to load into the buffer cache.
In intermediate blocks in `ctid` the reference to the index block is stored, 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` empty (interpreted as " minus infinity " , i.e. the boundary is unknown) and this link leads to the leftmost child block.
In the example on the slide the second number is zero `ctid : (3, 0 ) (1, 0 ) (5, 0 ) (10, 0 )` and they have a `data field` empty.

FILLFACTOR in btree indexes

- for indexes on the column being filled fillfactor =100 should be set in ascending order
- in case of setting `fillfactor =100` the left block when dividing the rightmost leaf block will be filled to the maximum, the data will be stored more compactly, the division of the right block will occur less frequently
- In btree indexes fillfactor is set to the values:
  - 90% for sheet blocks
  - 70% for non-leaf blocks and it doesn't change
  - 96% when splitting any leaf block that is completely filled with duplicates (same value)
- fillfactor sheet blocks the following is used:
  - during index building
  - when splitting the rightmost page of both the sheet and intermediate levels

FILLFACTOR in indexes type btree

If, when inserting a row into a table, the rightmost page at a level is divided (the last page of the level), then **the division is not equal, the left page is filled up to fillfactor** , and the right one remains almost free. This is useful for indexes on auto-incrementing columns or filled ones. increasing sequence, since insertions will always go to the rightmost leaf block and it will be divided. For such indexes, it is worth setting fillfactor =100 , otherwise the indexes will be larger , space in leaf blocks will be wasted . In case of setting `fillfactor =100` the left block will be filled to the maximum when dividing the rightmost leaf block, the data will be stored more compactly, and the right block will be divided less frequently. The intermediate blocks of such indexes will be filled to 70% regardless of the fillfactor value . However, the intermediate blocks make a small contribution to the index size. **For a numeric column (int2, int4, int8), the intermediate block will fit 286 links at 70% fill, and 407 links at 100%.**

This is not shown in the examples, quite the opposite: there are more lines in the right blocks. This is due to the line size and fillfactor , which is exceeded even by one line.

If the blocks could hold more lines, then when dividing the right block (the block is divided if there is no space in it, regardless of fillfactor), the lines up to fillfactor would go to the left block, and the rest would remain in the right block.

btree indexes fillfactor set to values:

1) 90% for leaf blocks (`BTREE_DEFAULT_FILLFACTOR=90`)

2) 70% for non-leaf blocks and this does not change (`BTREE_NONLEAF_FILLFACTOR=70`) .

fillfactor sheet blocks the following is used:

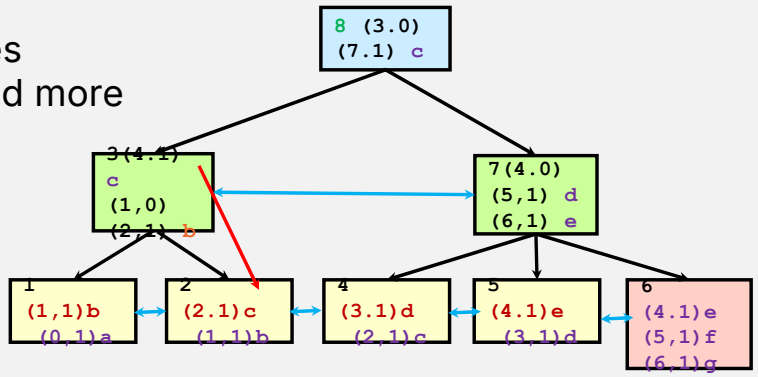
1) during index construction

2) when dividing the rightmost page of both the sheet and intermediate levels.

When splitting pages that are not the rightmost, the data is redistributed equally. A link to the second block is inserted into the parent block. If there is no space in the parent block, it is divided and so on up to the root block. If there is no space in the root block, a new level is added. When splitting any (including the right) leaf block that is completely filled with the same value (duplicates), fillfactor = 96 (`BTREE_SINGLEVAL_FILLFACTOR=96`) .

Fastpath for inserting into indexes

- the process that inserted into the right leaf block remembers the reference to it during subsequent insertions if the new value is greater than the previous one (or empty) and does not pass the path from the root to the leaf block
- speeds up insertions:
- into integer columns filled with a sequence
- datetime (filled with DEFAULT)
- when inserting mostly NULL values
- used when the number level=2 and more



Fastpath inserts into indexes

The primary key column is usually made auto-incremental , that is, a monotonically increasing sequence of integers is used.

In this case, when inserting (INSERT) a new row, it will always be inserted into the rightmost leaf block and when inserting, there is no point in going through the blocks starting from the root. Also, when inserting into a datetime column filled by DEFAULT insertion time or filled in another way. Also when inserting an empty value, that is, if rows are inserted sequentially, and NULL is inserted into the indexed column.

The optimization of row insertion is called fastpath . The process that inserted into the right leaf block remembers the reference to it during subsequent insertions if the new value is greater than the previous one (or empty) and does not pass the path from the root to the leaf block.

The process forgets the block address and starts searching again from the root if for some reason it has performed an insertion (entries are only inserted into the index, they do not change, and are only deleted by vacuum) not into the rightmost block.

Fastpath is used when the number of levels in the index starts from 2 (`BTREE_FASTPATH_MIN_LEVEL`). The optimization works well if one process inserts rows into the table. If the process encounters a lock on the right leaf block with

another process (checking that the buffer is not pinned by other processes), this process stops using fastpath . The other process can continue,

since it has not collided. It turns out that the process is "sent to a penalty lap": it will read several blocks from the root to the right leaf block, and during this time it will not collide with other processes.

It is not worth parallelizing row insertions into a table, assuming that this is always faster: processes may encounter a pinned buffer.

Value in the third internal block b points to the leftmost (in the sort order with which the index is created) value , which is b in the second leaf block. Leaf blocks other than the rightmost leaf block store one value and a HighKey .

In-page cleaning in indexes

- performed during index scan
- If a row in a table is deleted, the index entry for the row or version chain is marked with the LP_DEAD flag.
- the mark can be set by the SELECT command during index scan
- no journal entry is created but the block gets dirty
- The marked index entry will be cleared when executing commands that change data in the table.

```
create table t (id int primary key, c text) with ( autovacuum_enabled = off);
insert into t SELECT i , 'simple delete ' || i from generate_series (1, 10000) as i ;
delete from t where id between 100 and 9000;
analyze t;
explain (analyze, settings, buffers, costs off) select * from t where id between 1 and 9000
;
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , tids , left(data,8) from
bt_page_items ('t_pkey',20) where dead=true limit 2;
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | substring
------------+---------+---------+-------+------+------+---------+------+-----------
2 | (42.37) | 16 | f | f | t | (42.37) | | bd 19 00
3 | (42.38) | 16 | f | f | t | (42.38) | | be 19 00
```


In-page cleaning in indexes

If at Index Scan the server process detects that a row (or a chain of rows referenced by an index entry) has been deleted and has gone beyond the database horizon, then in the index entry of the leaf block (leaf page) in lp_flags the LP_DEAD hint bit is set (**called known dead, killed tuple**). The bit can be viewed in the `dead column` , returned by **`the bt_page_items (' t_idx ', block )` function** .

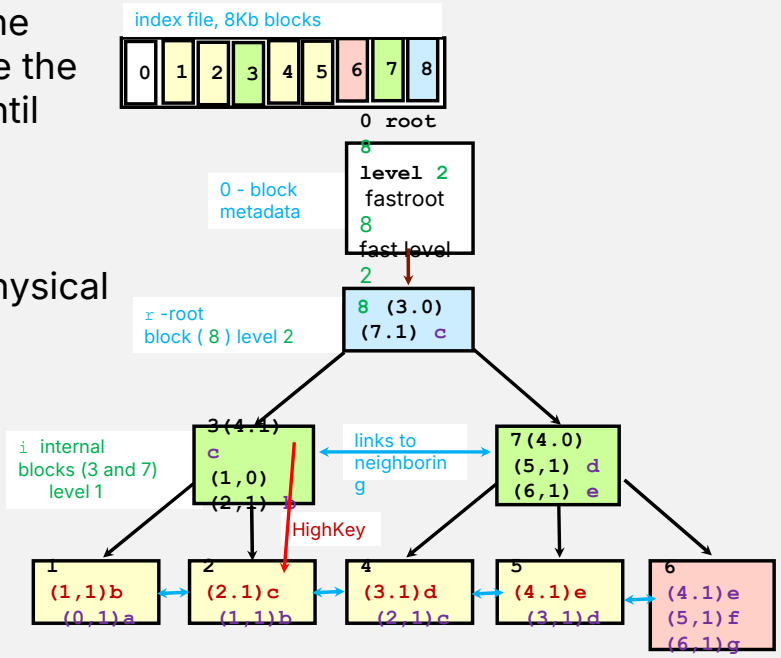
Not installed for Bitmap Index Scan and Seq Scan (non-index access) . A row marked with this flag will be deleted later when a command is executed that makes changes to the index block.

Returning to the block and setting a flag in it adds overhead and increases the execution time of the command, but it is done once. However, subsequent commands will be able to ignore the index entry and will not access the table block.

```
drop table t;
create table t (id int primary key, c text) with ( autovacuum_enabled = off);
insert into t SELECT i , 'simple delete ' || i from generate_series (1, 10000) as i ;
delete from t where id between 100 and 9000;
analyze t;
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , tids , left(data,8) from bt_page_items
('t_pkey',20) limit 2;
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | substring
------------+---------+---------+-------+------+------+---------+------+-----------
1 | (44.1) | 16 | f | f | | | | 2b 1b 00
2 | (42.37) | 16 | f | f | f | (42.37) | | bd 19 00
explain (analyze, settings, buffers, costs off) select * from t where id between 1 and 9000;
                            QUERY PLAN
------------------------------------------------------------------------
 Index Scan using t_pkey on t (actual time=0.013..1.480 rows=99 loops=1)
Index Cond : ((id >= 1) AND (id <= 9000))
select itemoffset , ctid , itemlen , nulls, vars , dead , htid , tids , left(data,8) from bt_page_items
('t_pkey',20) where dead=true limit 2;
 itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | substring
------------+---------+---------+-------+------+------+---------+------+-----------
2 | (42.37) | 16 | f | f | t | (42.37) | | bd 19 00
3 | (42.38) | 16 | f | f | t | (42.38) | | be 19 00
```

Impact of Row Deletion on Indexes

- when deleting a row in a table, the index record is not deleted, since the row in the table block remains until vacuuming
- in-page cleaning in indexes can remove records from indexes
- vacuum scans index block in physical order: 1,2,3,4,5,6,7,8



Impact of Row Deletion on Indexes

When a row in a table is deleted, the index record is not deleted because the row remains in the table block until vacuuming . Vacuum and in-page cleaning in indexes can remove records from indexes.

When a record is deleted (in a leaf block), no changes are made to the parent block, since the parent block stores a reference to the first record of the child block, which stores High key .

High key in the rightmost leaf block , because there are no blocks to the right of it. There are no rows with High key in the intermediate (internal) and root blocks, all rows are used. In the example on the slide, in the 3rd intermediate block, the first row " `3 (2,1) b` " refers to the 2nd leaf block. In the 7th intermediate block, the first row "(4,0)" refers to the 4th leaf block. In the intermediate blocks, the row stores an indexed **value** and a link to a lower block that contains this value or more (High key of leaf blocks are not taken into account). In one of the rows of the intermediate block, the value is empty, this means that this link is " the leftmost " .

If all records in a block are deleted, it is marked with the flag `btpo_flags +=16` (means empty block), but the block is still in the tree structure (half-dead), only references to the block are removed from neighboring blocks at the same level.

Later, the block is removed from the index structure (the reference to it from the upper block is removed), the block is marked with the flag `btpo_flags+=4` as a free block (removed from the index structure). At the same time, the block remains in the index file. The block will not be used for inserting new rows until it is processed by vacuum. The block will be able to return to the index structure, since vacuum marks it in the fsm free space map as free. Server processes search for free blocks in the visibility map and only then expand the index file. Splitting into two steps is necessary because other processes may have locks on the block. Waiting for the locks to be released or checking for them is less productive than splitting the process of removing a block from the index structure into two steps.

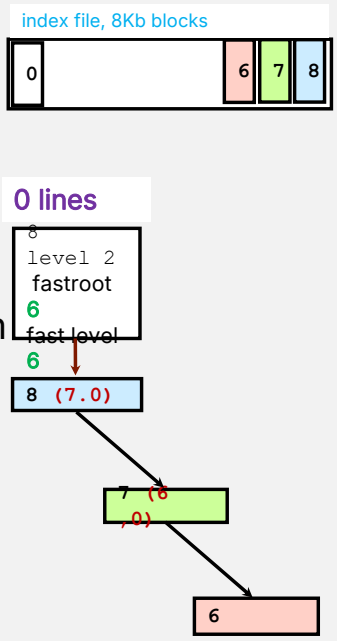
As a conclusion, processing the index with vacuum (autovacuum) is important.

Vacuum scans index blocks not in logical order (leaf by links to neighboring ones), but in physical order : 0,1,2,3,4,5 , 6,7,8.

That is why vacuum can find unused (removed from index structure) blocks even if fsm files are missing. Vacuum updates or creates fsm index files. btree index does not have vm file .

Excluding blocks from the index structure

- after deleting rows in a table, vacuum can exclude blocks from the index structure
  - blocks remain in index files
  - are marked as free in the free space map (fsm) index
  - can be reused by being embedded into the index structure and elsewhere in the tree
- the number of levels is not reduced by vacuum
- " right " block of each index level cannot be removed from the tree
- if the root and internal (intermediate) blocks have one heir, then it becomes " fastroot " - fast root
  - its address and its level are stored in the metadata block in the fastroot fields and fastlevel
  - index search starts with fastroot , not root



Excluding blocks from the index structure

The first (0) block of the index contains metadata :

3) fastroot and fastlevel . After deleting all or a large number of rows in a table, vacuum can exclude blocks from the index structure. Such blocks remain in the index files, in the free space map (fsm) index are marked as free and can be reused by being embedded in the index structure and elsewhere in the tree. But even if all rows are deleted, the number of levels to which the index has grown does not decrease. This is explained by the fact that the " rightmost " block of each index level cannot be deleted from the tree. In the example on the slide , the undeletable blocks are: 8, 7, 6. If this happens (many rows are inserted into the table, the number of levels becomes large, then all rows are deleted and the level does not decrease), then the " fastroot " optimization is used . If the root and intermediate (internal) blocks have one heir, then it becomes a " fastroot " - a fast root. Its address and its level are stored in the metadata block in the fastroot fields and fastlevel . And the index search starts with fastroot , not root. This significantly speeds up the search.

```
select * from bt_ multi _page_stats ('t_idx', 1 , -1 );
blkno|type|live_|dead_|avg_item|page_|free_|btpo_|btpo_|btpo_|btpo_
| | items|items | _size| size| size| prev | next|level|flags
-----+----+-----+-----+--------+-----+-----+-----+-----+-----+----
   1 | d | 0 | 0 | 2512 |8192 |8140 | 0 | 2 | 0 |261
2 | d | 0 | 0 | 2512 |8192 |8140 | 0 | 4 | 0 |261
3 | D | 0 | 0 | 1677 |8192 |8140 | 0 | 7 | 1 |260
4 | d | 0 | 0 | 2512 |8192 |8140 | 0 | 5 | 0 |261
5 | d | 0 | 0 | 2512 |8192 |8140 | 0 | 6 | 0 |261
6 | l | 0 | 0 | 2512 |8192 |8148 | 0 | 0 | 0 | 1
7 | i | 3 | 0 | 8 |8192 |8136 | 0 | 0 | 1 | 0
8 | r | 2 | 0 | 8 |8192 |8136 | 0 | 0 | 2 | 2
```

261=1+4+256, which means leaf, deleted, sign of a deleted block.

In the function arguments:, 1 from the first block -1 until the last block.

```
select * from bt_metap (' t_idx ');
magic|version| root | level |fastroot|fastlevel|last_cleanup_num_delpages|last_cleanup_num_tuples|allequalimag
-----------+-------+----+-----+--------+---------+-------------------------+-----------------------+-----------
340322| 4 |  8 | 2 | 6 | 0 |                       5 | -1 | t
select itemoffset , ctid , itemlen , nulls, vars , dead, htid , tids , left(data,24) data from bt_page_items ('
t_idx ', block_number );
 block | itemoffset | ctid | itemlen | nulls | vars | dead | htid | tids | data
-----------------+------------+-------+---------+-------+------+------+-------+------+------
   7 | 1 | ( 6 , 0 ) | 8 | f | t | | | |
   8 | 1 | ( 7 , 0 ) | 8 | f | t | |     | |
```

Number of blocks excluded from the index structure

- query to get the number of blocks excluded from the index structure:

```
select type, count(*) from bt_multi_page_stats ('t_pkey',1,-1) group by
type;
 type | count
------+-------
  D | 6
 r | 1
 l | 552
  i | 4
 d | 2181
```

- type: d - deleted leaf (deleted leaf) block, D - deleted internal (deleted internal) block
- the second column in the query shows the number of blocks
- LP_DEAD changes are not transmitted to the replica, on the replica the LP_DEAD bits are not set when scanning by index
- When vacuuming, all blocks in index files are read
  - the fewer blocks in the index files, the faster the autovacuum will work

Number of blocks excluded from the index structure

Query to get the number of blocks excluded from the index structure:

```
select type, count(*) from bt_multi_page_stats ('t_pkey',1,-1) group by type;
type | count
------+-------
D | 6
r | 1
l | 552
 i | 4
d | 2181
(5 rows)
```

The first column shows the block types: d - deleted leaf (deleted leaf) block, D - deleted internal (deleted internal) block.

The second column in the query shows the number of blocks.

Blocks in which all records have been marked as deleted (during vacuum or earlier) are excluded from the index structure by vacuum. Excluded blocks are not scanned by queries, but they are read during any vacuum , since indexes do not have a visibility map and all blocks are read. Vacuum reads all index blocks sequentially and in physical order. Blocks excluded from the index structure are returned to the index structure when index blocks are split (i.e., the index structure grows).

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.

LP_DEAD hint bits are not logged or propagated to the replica.

https://www.postgresql.org/message-id/flat/7067.1529246768@sss.pgh.pa.us#d9e2e570ba34fc96c4300a362cbe8c38

Practice

- Part 1. Access methods
- Part 2. Using Indexes with Integrity Constraints
- Part 3. Characteristics of btree indexes
- Part 4. Navigating the btree index structure
- Part 5. Deduplication in btree indexes
- Part 6. Indexes in descending order
- Part 7. Covering indices and Index Only Scan
- Part 8. Partial indices
- btree index
- Part 10. Cleaning index blocks during its scanning
- Part 11. Slow query execution on replica due to lack of index block cleaning
- Part 12. Determining the number of deleted lines
- Part 13. Search by btree index structure

Practice

In practice, all details of using b-tree indexes are considered .
In Part 11 of the practice, you will create a physical replica and see how cleaning blocks of old row versions can affect queries running on the replica.

8-1

TOAST

TOAST (The Oversized-Attribute Storage Technique)

- TOAST is a technique for storing large " attributes " (fields)
- Large fields are fields that do not fit into the block.
- allows you to store fields up to 1GB in size
- if the table has a deliberately large field or a row with a large field is inserted, then a toast table and an index on the toast table are created
- individual row fields are taken out into TOAST
- the extracted fields are divided into parts (chunk) of 199 6 byte:

```
postgres@tantor :~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk: 1996
postgres@tantor :~$ / usr /lib/ postgresql /15/bin/ pg_controldata
-D / var /lib/ postgresql /15/main| grep TOAST
Maximum size of a TOAST chunk: 1988
postgres@vanilla- x32 :~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk: 2000
```


TOAST (The Oversized-Attribute Storage Technique)

TOAST (The Oversized- Attribute Storage Technique) is used not only for storing individual fields in a TOAST table. The PostgreSQL core code is used when processing long values in memory. Not all built-in data types support the TOAST technique . They do not support fixed-length data types, since their length is small and the same for any values (" fixed ") , for example, 1,2,4,8 bytes.

The size of a type that supports TOAST is limited to 1 Gigabyte. This limitation follows from the fact that 30 bits (2 ^ 30 = 1 Gb) of 1 or 4 bytes are allocated for the length at the beginning of the field in the block (3 2 bits). Two bits in these bytes are used to indicate: 00 - the value is short, not TOAST , the remaining bits specify the length of the field together with this byte ; 01 - the field length is stored in one byte, the remaining bits specify the length of the field in bytes and these 6 bits can store a length from 1 to 126 bytes (2 ^ 6 = 64 , but this is for the range from zero) ; 10 - the value is compressed, the remaining bits specify the length of the field in compressed form. Values with one field header byte are not aligned. Values with four field header bytes are aligned on a `pg_type.typealign boundary` .

taken out to TOAST are divided into parts - " chunks " (after compression, if applied) of size 1996 bytes (the value is specified by the constant TOAST_MAX_CHUNK_SIZE), which are located in the rows of the TOAST table with a size of 2032 bytes (the value is specified by the constant TOAST_TUPLE_THRESHOLD). **The values are chosen so that four rows fit into a TOAST table block .** Since the table field size is not a multiple of 1996 bytes, the last chunk of the field can be smaller.

The TOAST_MAX_CHUNK_SIZE value is stored in the cluster control file and can be viewed using the **`pg_controldata utility`** .

The TOAST table has three columns: chunk_id (OID type, unique for the field taken out to TOAST , size 4 bytes), chunk_seq (chunk ordinal number , size 4 bytes), chunk_data (field data, bytea type , size of raw data plus 1 or 4 bytes for storing the size). For quick access to chunks , a composite unique index is created on the TOAST table by chunk_id and chunk_seq . A pointer to the first chunk of the field and other data remains in the table block . **The total size of the remaining part of the field in the table is always 18 bytes** .

In 32-bit PostgreSQL, the chunk size is 4 bytes larger: 2000 bytes.

In AstraLinux PostgreSQL the chunk size is 8 bytes smaller: 1988 bytes.

https://docs.tantorlabs.ru/tdb/ru/15_6/se/storage-toast.html

Variable length fields

- the line must fit into one block
- varlena fields that do not fit into the block are moved to the TOAST table
- fixed width data types are not compressed or TOASTed
- The storage strategy can be set with the command `A LTER TABLE name ALTER COLUMN name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT } .`
- compression is supported for MAIN and EXTENDED
- **1 byte at the beginning of the variable width field stores the length of a field up to 127 bytes long**
- **4 bytes in at the beginning of a variable width field stores the field length for fields longer than 126 bytes**

Variable length fields

A row (record) of a table must fit into one block of 8Kb and cannot be located in multiple blocks of table files. However, rows can be larger than 8Kb. TOAST is used to store them .

btree index entry cannot exceed approximately one third of a block (after compression of indexed columns, if applied to the table).

TOAST supports `varlena data types ( pg_type.typlen = -1)` . Fixed-length fields cannot be stored outside the table block, since there is no code written for these data types that implements storage outside the table block (in a TOAST table). In this case, the row must fit in one block and the actual number of columns in the table will be less than the limit of 1600 columns (`MaxHeapAttributeNumber` in `htup_details.h`) .

To support TOAST, **the first byte or first 4 bytes of a varlena field 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 with the data, i.e. stored in 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 3 (typealign = ' i ') or up to 7 bytes (typealign = 'd') .

In other words, a storage scheme designer **is better off specifying char(126) and less than char(127) and greater** .

Varlena fields with one byte of length **are not aligned** , and **fields with 4 bytes of length are aligned to pg_type.typealign .** For most types of variable lengths, alignment is up to 4 bytes (`pg_type.typalign = i`). The lack of alignment gives a gain in storage space, which is noticeable for short values. But you should always remember about aligning **the entire string to 8 bytes, which is always done** .

Compression is supported only for variable-length data types . Compression is performed only if the column storage mode is set to MAIN or EXTENDED. If a field is stored in TOAST and the UPDATE command does not affect that field, the field will not be specially compressed or decompressed.

For most variable length types, the default mode is EXTENDED , except for types:

```
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)
```

For each column, in addition to the mode, you can also set the compression algorithm (CREATE or ALTER TABLE). If not set, the algorithm from the `default_toast_compression parameter is used, which is set to pglz` by default .

The storage mode (strategy) can be set with the command **`A LTER TABLE name ALTER COLUMN name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT }`** .

EXTERNAL is similar to EXTENDED, but without compression and not set by default on standard types. If the `pglz algorithm` cannot compress the first kilobyte of data, it stops trying to compress.

Field displacement in TOAST

- Tantor and PostgreSQL DBMS, fields of strings longer than 2032 bytes are removed
  - in this case, the fields are cut into parts of 1996 bytes
- algorithm (order) of displacement depends on the order of the columns
- when accessing each displaced field, an additional 2-3 TOAST index blocks are read

```
select reltoastrelid , reltoastrelid ::
regclass from pg_class where relname ='t';
 reltoastrelid | reltoastrelid
---------------+------------------------
74295 | pg_toast.pg_toast_74292
\d+ pg_toast.pg_toast_74292
TOAST table "pg_toast.pg_toast_74292"
Column | Type | Storage
------------+---------+---------
 chunk_id | oid | plain
 chunk_seq | integer | plain
 chunk_data | bytea | plain
Owning table: " public.t "
Indexes:
"pg_toast_74292_index" PRIMARY KEY,
            btree ( chunk_id , chunk_seq )
Access method: heap
select chunk_id , chunk_seq , length(
chunk_data ) from pg_toast.pg_toast_74292;
 chunk_id | chunk_seq | length
----------+-----------+--------
74297 | 0 | 1996
74297 | 1 | 9
```


Field displacement in TOAST

Storage method for regular tables (heap tables) allows compression of individual field values. Compression algorithms are less effective on small data. Access to individual columns is not very effective because the server process needs to find the block that stores the part of the row that fits in the block, then separately determine for each row whether it is necessary to access the TOAST table rows, read its blocks and glue together the parts of the fields (chunk) that are stored in it as rows of this table.

One table can have only one associated TOAST table and one TOAST index (unique btree index on `the chunk_id` and `chunk_seq columns`). The OID of a TOAST table is stored in the `pg_class.reltoastrelid field` .

When accessing each evicted field, 2-3 TOAST index blocks are additionally read, which reduces performance even if the blocks are in the buffer cache . The main slowdown is in obtaining a lock to read each extra block. Any shared resources (that which is not in the local memory of the process) require obtaining a lock even to read the resource.

Fields after compression (if any) are divided into parts (chunk) 1996 bytes :

```
postgres@tantor :~$ pg_controldata | grep TOAST
Maximum size of a TOAST chunk: 1996
```

In Tantor and PostgreSQL DBMS , a row is considered for placing part of its fields in TOAST if the row size is greater than 20 32 bytes. Fields will be compressed and considered for TOAST storage until the row fits into 20 32 bytes or `toast_tuple_target` byte if the value was set by the command:

```
ALTER TABLE t SET ( toast_tuple_target = 2032 );
```

The rest of the line must fit into one block (8Kb) in any case.

For postgresql from Astralinux 1.8.1 :

```
Maximum size of a TOAST chunk: 1988
```

therefore, fields longer than 1988+8=1996 bytes will be taken out, and not 2004. In this case, a field 1997 bytes long will also generate 2 chunks , the second chunk 9 bytes in size, the first 1988 bytes in size.

In 32- bit PostgreSQL 9.6 - 2009 bytes (and the maximum chunk size is 2000).

Field displacement algorithm in TOAST

- when inserting a row into a table, it is completely placed in the server process memory in a 1GB (or 2GB) string buffer
- when updating a row, processing is performed on the fields affected by the command within the string buffer. Fields not affected by the command are represented in the buffer by an 18-byte header.
- after processing (compression or (takeout) each field is checked for the size of the string. If the size does not exceed `toast_tuple_target` (default 2032 bytes) the string is saved to the buffer and processing of the string is finished
- processing starts with the EXTENDED and EXTERNAL fields from the largest size to less
- compression and removal of MAIN fields is performed only after removal of all EXTENDED and EXTERNAL fields

Field displacement algorithm in TOAST

When a row is inserted into a table, it is completely placed in the server process memory in a 1GB (or 2GB) string buffer for sessions that have the configuration parameter `enable_large_allocations =on set`).

Displacement algorithm in four passes:

EXTENDED and EXTERNAL fields are selected in turn from largest to smallest . After each field is processed, the row size is checked and if the size is less than or equal to `toast_tuple_target` (default 2032 bytes), then the eviction is stopped and the row is saved in the table block.

EXTENDED or EXTERNAL field is taken . EXTENDED is compressed. If the size of the row with the field in compressed form exceeds 2032, the field is displaced in TOAST. The EXTERNAL field is displaced without compressing.

2) If the row size is still greater than 2032, the second pass flushes the remaining already compressed EXTENDED and EXTERNAL in turn until the row size is less than 2032.

3) If the row size is not less than 2032, the MAIN fields are compressed in order of size. After each field is compressed, the row size is checked.

4) If the row size has not become less than 2032, the MAIN compressed in the 3rd pass are evicted one by one.

5) If the string size does not fit into the block, an error is generated :

```
row is too big: size ..., maximum size ...
```

When updating a row, processing is performed on the fields affected by the command. within the string buffer. Fields not affected by the command are represented in the buffer by an 18-byte header.

Toast chunk

- when using EXTERNAL , fields larger than 1997 bytes create a second a small chunk size due to which only 3 large chunks fit into a TOAST block
- EXTERNAL fields with a size from 1997 to ~ 2300 bytes there is a possibility of less dense storage

```
select reltoastrelid , reltoastrelid ::
regclass from pg_class where relname ='t';
 reltoastrelid | reltoastrelid
---------------+------------------------
74295 | pg_toast.pg_toast_74292
\d+ pg_toast.pg_toast_74292
TOAST table "pg_toast.pg_toast_74292"
Column | Type | Storage
-----------+---------+---------
 chunk_id | oid | plain
 chunk_seq | integer | plain
 chunk_data | bytea | plain
Owning table: " public.t "
Indexes:
"pg_toast_74292_index" PRIMARY KEY,
            btree ( chunk_id , chunk_seq )
Access method: heap
select chunk_id , chunk_seq , length(
chunk_data ) from pg_toast.pg_toast_74292;
 chunk_id | chunk_seq | length
----------+-----------+--------
74297 | 0 | 1996
74297 | 1 | 9
```


Toast chunk

The field is TOASTed if the row size is greater than 2032 bytes, and the field will be cut into parts of 1996 bytes. Because of this, for a field larger than 1996 bytes , a small chunk that will be inserted by the server process into a block with a large chunk . For example, insert 4 rows into the table:

```
drop table if exists t;
create table t (c text);
alter table t alter column c set 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));
```

TOAST block will fit 3 long chunks :

```
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));
 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
```

Full size of a line with a long chunk **2032** bytes (6104 - 4072).

```
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
```

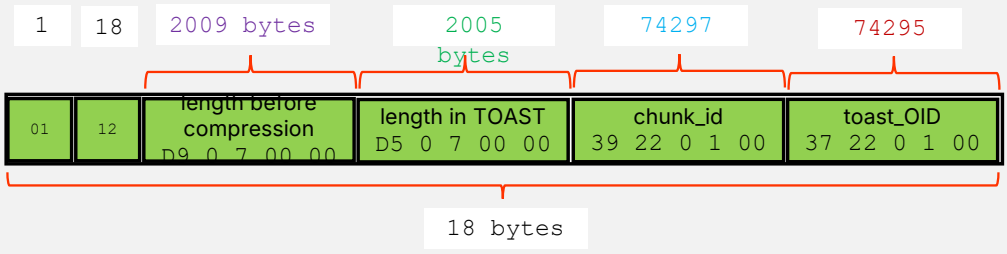
Example of how to calculate block space for 4 lines of 2032 bytes (with 4 chunks):

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.

TOAST Limitations

- in any table in TOAST no more than 2 ^32 fields can be taken out (~4 billion)
- for a field in TOAST , 18 bytes are stored in the table block , they are not aligned, but the entire row is aligned
- the value remaining in the table block after the field has been TOASTed:

```
select lp_len , t_data from heap_page_items ( get_raw_page ( 't'
, 0 ) );
 lp_len | t_data
--------+----------------------------------------
42 | \x0112 d9070000 d5070000 39220100 37220100
```



TOAST Limitations

In PostgreSQL, the special service area is at the end of table blocks No:

```
48 | 1952 | 8192 | 8192
```

In 32-bit PostgreSQL :

```
Maximum size of a TOAST chunk: 2000
```

When using EXTENDED , the field will most likely be compressed and there will be no small chunk .

https://eax.me/postgresql-toast/

Each field is stored in a TOAST table as a set of rows (chunk) is stored as a single row in a TOAST table.

The main table field stores a pointer to the first chunk **18 bytes in size** (regardless of the field size). These 18 bytes store the `varatt_external` structure , described in `varatt.h` :

the first byte has the value 0 x0 1 , this is a sign that the field is TOASTed ;

the second byte is the length of this record (value 0x12 = 18 bytes) ;

4 bytes 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 the TOAST table);

4 bytes - oid toast tables (`pg_class.reltoastrelid`)

In the chunk_id column (type oid 4 bytes) can be 4 billion. (2 to the power of 32) values. This means that only 4 billion **fields** (not even rows) can be TOASTed in a single table . This significantly limits the number of rows in the original table and monitoring is probably desirable. Partitioning can work around this limitation.

MAIN mode is used for storage inside block in compressed form, EXTERNAL - for storing in TOAST in uncompressed form, EXTENDED for storing in TOAST in compressed form. If the values are poorly compressed or you plan to process field values (for example, text fields with the substr , upper functions), then using the EXTERNAL mode will be effective. For fixed-width types, the PLAIN mode is set, which cannot be changed by the ALTER TABLE command, an error " `ERROR: column data type type can only have storage PLAIN` ".

https://habr.com/ru/companies/postgrespro/articles/710104/

TOAST -Erased Fields

- both individual fields and **the entire line are aligned**
- each line is aligned to 8 bytes

Example:

Two tables:

```
create table t3 (c1 text);
create table t4 ( c1 serial , c2 smallint , c3 text );
```

insert rows with a large field length so that all fields are TOASTed :

```
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 ;
```

then the same number of rows are placed in the block of both tables:

```
select count(*) from heap_page_items ( get_raw_page ('t3','main',0)) where ( t_ctid ::text::point)[0]=0
union select count(*) from heap_page_items ( get_raw_page ('t4','main',0)) where ( t_ctid
::text::point)[0]=0;
count
-------
156
( 1 row)
```

The space occupied by the tables is also the same.

- in the table t4 you can save two columns at the expense of space that could not be used in the first table

TOAST -Erased Fields

In the table block for a field placed in a TOAST table, a pointer of always 18 bytes is stored.

Alignment of individual fields is known, but alignment of the entire row is often forgotten. Is the row with the header or the data area aligned ? Both. The row header is always aligned to 8 bytes and can be 24, 32, 40 ... bytes in size. If you say that the data area is aligned, then the data area with the header (the entire row) will automatically be aligned. If you say that the entire row and header are aligned, then it automatically follows that the data area will be aligned. They are **all** aligned to 8 bytes, since the Tantor DBMS only works on 64-bit operating systems (x86-64 and ARM64) . If you create two tables:

```
create table t3 (c1 text);
create table t4 ( c1 serial , c2 smallint , c3 text );
```

insert 200 rows with a long field so that the long field value is TOASTed :

```
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 ;
```

are placed in the block of both tables :

```
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;
 156
 156
```

In the second table Can store two columns in the same place that in the first table could not be used (6 bytes) due to padding of the entire row.

The size of the table files is also the same:

```
select pg_total_relation_size ('t3'), pg_total_relation_size ('t4');
5070848               | 5070848
```

If you replace serial and smallint on bigserial , then the size will be different (156 and 135 lines in each block). bigserial is useless, since the tables will be able to store no more than 4 billion (2 ^ 32) lines with a long field .

toast_tuple_target and default_toast_compression parameters

- two macros affect the eviction: TOAST_TUPLE_THRESHOLD and TOAST_TUPLE_TARGET
  - by default equal to **2032**
- if the row size is greater than TOAST_TUPLE_THRESHOLD , then compression and / or displacement of row fields begins
- fields will be compressed and considered for storage in TOAST , until the remaining fields fit into TOAST_TUPLE_TARGET
- TOAST_TUPLE_TARGET can be overridden only **at table level** :
- `ALTER TABLE t SET ( toast_tuple_target = 2032 ) ;`
- **TOAST_TUPLE_THRESHOLD is not overridden**
- parameter `default_toast_compression` configuration sets the compression algorithm:
- `ALTER SYSTEM SET default_toast_compression = pglz ;`

toast_tuple_target and default_toast_compression parameters

The eviction is affected by two macros defined in the source code (`heaptoast.h`):
TOAST_TUPLE_THRESHOLD and TOAST_TUPLE_TARGET , which have the same value. If the row size is greater than TOAST_TUPLE_THRESHOLD , then compression and / or eviction of row fields begins .
Fields will be compressed and considered for storage in TOAST , until the rest of the row (complete: with row header) fits into TOAST_TUPLE_TARGET . The value can be overridden at the table level:

```
ALTER TABLE t SET ( toast_tuple_target = 2032 );
```

TOAST_TUPLE_THRESHOLD is not overridden.
There is also a parameter that sets the compression algorithm. pglz or lz4:

```
default_toast_compression
```

The constants are defined in the source code:

```
#define MaximumBytesPerTuple ( tuplesPerPage ) MAXALIGN_DOWN((BLCKSZ - MAXALIGN(
SizeOfPageHeaderData + ( tuplesPerPage ) * sizeof ( ItemIdData )))/( tuplesPerPage ))
#define TOAST_TUPLES_PER_PAGE 4
#define TOAST_TUPLE_THRESHOLD MaximumBytesPerTuple (TOAST_TUPLES_PER_PAGE)
#define TOAST_TUPLE_TARGET TOAST_TUPLE_THRESHOLD
```

Block header parameters:

```
ItemIdData = 4 bytes
SizeOfPageHeaderData = 24 bytes
```

If you substitute the values, you get:
TOAST_TUPLE_TARGET = TOAST_TUPLE_THRESHOLD = MAXALIGN_DOWN((BLCKSZ - MAXALIGN(24 + (4) * sizeof (4)))/(4)) = MAXALIGN_DOWN((BLCKSZ - MAXALIGN(24 + 4 * 4))/ 4) = MAXALIGN_DOWN((8192 - MAXALIGN(40))/ 4) = MAXALIGN_DOWN((8192 - 40)/ 4)=MAXALIGN_DOWN(2038) = **203 2** .
TOAST_TUPLE_TARGET also defines the maximum size of TOAST table rows. The header of a row of a regular and TOAST table is 24 bytes. The size of the data area of a TOAST table row is 2032-24=2008 bytes. There are three fields in a row: oid (4 bytes) , int4 (4 bytes) , bytea. In bytea , the first byte at the beginning of a variable-width field stores the length of a field up to 127 bytes long, the first 4 bytes at the beginning of a variable-width field store the length of a field for bytea longer than 126 bytes. Alignment is 4 bytes. 2008-4-4-4=1996.

Heap Only Tuple Optimization

- When updating (`UPDATE`) rows, changes may not be made to indexes
- changes do not go beyond the table block (heap only)
- conditions:
  - only fields that are not included in any of the indexes (except for brin type indexes) on the table are changed
  - the new version of the line is placed in the same block as the previous version
- if the new version of the row is located in a block different from the one in which the previous version of the row is located, then:
  - the previous version will become the last in the chain of HOT versions
  - new records will be created in all indexes on the table pointing to the new row version

Heap Only Tuple Optimization

When updating a row, a new row is created inside the table block. If the fields that were TOASTed were not changed, then the contents of the fields that reference TOAST will be copied without changes and there will be no changes in TOAST .

If indexes are created on any columns of the table, the index entries point to ctid previous version of the row. Index records point to a field in the block header.

If **only fields that are not mentioned in any index (except brin indexes) are changed , then no changes** are made to the indexes . Partial index :

```
create index t5_idx on t5 (c1) where c1 is not null;
```

does not allow HOT to be performed , if the UPDATE command mentions column c1 even if the UPDATE contains the condition WHERE c1 is null .

Similarly, a partial covering index:

```
create index t5_idx1 on t5 (c1) include (c2) where c1 is not null;
```

does not allow HOT to be performed , if the UPDATE command mentions columns c1 and c2 .

When HOT , the t_ctid field refers to the new version of the row . The old row header has the HEAP_HOT_UPDATED bit set in t_infomask2, and the new row version has the HEAP_ONLY_TUPLE bit set.

From the index, the server process gets to the old version of the row, sees the HEAP_HOT_UPDATED bit, and goes to the t_ctid field to a new version of the string (taking into account the visibility rules, if it can see this version, then the server process stops on it), checks the same bit, if it is set, then moves on to a newer version of the string. Such versions of the string are called a HOT chain of versions (HOT chain). Taking into account the visibility rules, the server process can reach the most recent version of the string, on which the HEAP_ONLY_TUPLE bit is set, and stop on it.

If the new version of the row is located in a different block than the old version of the row, then HOT does not apply. The t_ctid field of the old version will refer to the newer version in a different block , but the HEAP_HOT_UPDATED bit will not be set . The old version will become the last one in the chain. HOT versions. New records will be created in all indexes on the table pointing to the new version of the row.

https://www.cybertec-postgresql.com/en/hot-updates-in-postgresql-for-better-performance/

HOT update monitoring

- HOT statistics are available in the `pg_stat_all_tables` and `pg_stat_user_tables views.`
- **HOT update counter is collected for each table and is reflected in the column: `n_tup_hot_upd`**
- all updates are reflected in the `n_tup_upd column`
- **cases when during the update there was no space for the new version of the line and the HOT chain was broken, and the new version was inserted into another block shows `n_tup_newpage_upd`**
- statistics for the database are reset by calling the `pg_stat_reset () function ;`
  - After calling the function, it is recommended to perform ANALYZE on the entire

```
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
```


HOT Monitoring update

HOT statistics are available in two views **`pg_stat_all_tables`** and **`pg_stat_user_tables`** :

```
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
```

Statistics are accumulated since the last call to the `pg_stat_reset () function .` `pg_stat_reset ()` resets the cumulative statistics counters for the current database, but does not reset the cluster-level counters. **Zeroing counters zeroes the counters by which autovacuum decides when to run vacuuming and analysis. After calling the function, it is recommended to run ANALYZE on the entire database.** Cluster-level statistics accumulated in `the pg_stat _* views` are zeroed (" reset ") by calls functions:

```
select pg_stat_reset_shared (' recovery_prefetch ');
select pg_stat_reset_shared (' bgwriter ');
select pg_stat_reset_shared (' archiver ');
select pg_stat_reset_shared (' io ');
select pg_stat_reset_shared (' wal ');
```

Since version 17 `pg_stat_reset_shared (null)` resets all these caches , in version 16 it does nothing.

How to perform monitoring ? For example, created an additional index or increased the number of sections of a partitioned table, it is worth checking how the percentage of HOT updates has changed . n_tup_hot_upd - HOT update counter , n_tup_upd - in all updates.

Approximate estimate of the number of dead lines:

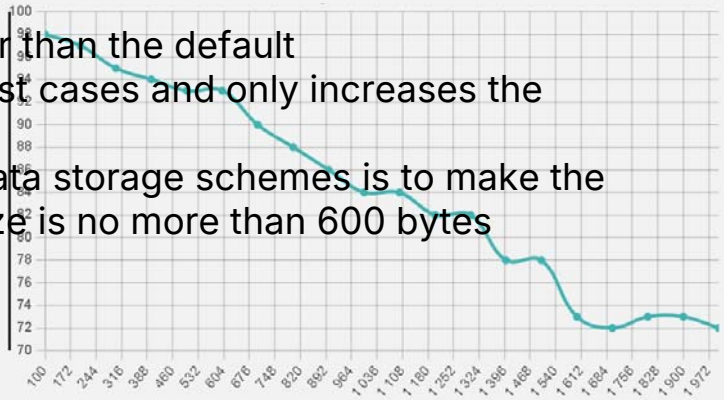
```
select relname , n_live_tup , n_dead_tup from pg_stat_all_tables where
n_dead_tup <>0 order by 3 desc ;
```

The cleanup code is implemented in heapam.c and pruneheap.c : "We prune when a previous UPDATE failed to find enough space on the page for a new tuple version, or when free space falls below the relation's fill-factor target (but not less than 10%)" .

https://postgres.ai/blog/20211029-how-partial-and-covering-indexes-affect-update-performance-in-postgresql

Impact of FILLFACTOR on HOT cleanup

- For HOT cleanup to be executed, the previous UPDATE of the row must exceed the `min(90%, FILLFACTOR) boundary`
- if the size of the rows in the table is such that less than 9 rows fit into the block, then the eighth row will not exceed the 90% limit, and the ninth row will be more than 11% of the block size and will not fit into the block
- 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 most effective way to design data storage schemes is to make the row size small. A reasonable row size is no more than 600 bytes



Impact of FILLFACTOR on HOT cleanup

Quick cleanup (HOT cleanup) is important and in many cases actively works. If the HOT conditions are met, then when updating rows in a block, the new version looks for a place in the block and a chain is created versions. If the inserted new version of the row fits into the block and the fill percentage exceeds the min(90%, FILLFACTOR) boundary , then the block header will display a flag that the block can be cleared. The next update of the block row will perform a HOT cleanup - it will clear the block of rows in the version chain that have gone beyond the base horizon, and the new version of the row will most likely fit into the block.

But if the fill percentage has not exceeded the min(90%, FILLFACTOR) boundary , and the new version does not fit into the remaining space in the block, then fast cleanup is not performed, the row version is inserted into another block, the HOT chain is interrupted, and a flag is inserted into the block header that there is no space in the block. This will happen if the block has less than 9 rows and FILLFACTOR=100% (the default value). In this case, it may be worth setting FILLFACTOR to a value of, where the new version of the row fits in the block and crosses the FILLFACTOR boundary. You should not design tables so that the row size is so large that less than 6 rows fit in a block.

```
create table t(s text storage plain) with ( autovacuum_enabled =off);
insert into t values (repeat('a',2010));
update t set s=(repeat('c',2010)) where ctid ::text = '(0,1)';
update t set s=(repeat('c',2010)) where ctid ::text = '(0,2)';
update t set s=(repeat('c',2010)) where ctid ::text = '(0,3)';
select ctid ,* from heap_page ('t',0);
 ctid  | lp_off | ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid | multi
-------+--------+-------+--------+-------+-------+-----+-----+--------+-------
 (0,1) |   6136 | (0,1) | normal | 1001c | 1002c | t   |     | (0,2)  | f
 (0,2) |   4096 | (0,2) | normal | 1002c | 1003c | t   | t   | (0.3)  | f
 (0.3) |   2056 | (0.3) | normal | 1003c | 1004  |     | t   | (1,1)  | f
(3 rows)
select ctid from t;
 ctid
-------
 ( 1 ,1)
```

The fourth version of the string was inserted into the second block.

The graph on the slide will be discussed in the practice for this chapter.

In-page clearing in tables

- When performing SELECT and UPDATE , a server process can remove dead tuples (row versions that have crossed the database visibility horizon , xmin horizon) by reorganizing the row versions within the block
- In-page cleanup is compatible with HOT and can pre-free up space that will be used by new versions rows resulting from UPDATE
- is fulfilled if:
  - the block is more than 90% full or FILLFACTOR (default 100%)
  - previously executed UPDATE was unable to place a new row version into this block
- rough estimate of row versions that can be purged: `pg_stat_all_tables . n_ dead_tup`

In-page clearing in tables

A server process executing SELECT and other commands can remove dead tuples (row versions that have passed the database visibility horizon , xmin horizon) by reorganizing the row versions within the block. This is called in-page cleanup.

HOT cleanup/pruning is fulfilled if one of the conditions is met:

the block is more than 90% full or FILLFACTOR (default 100%) :

```
minfree = Max( minfree , BLCKSZ / 10);
```

Previously executed UPDATE didn't find a place (that is, set a hint in the block title `PD_PAGE_FULL`).

Intra-page cleaning works within one table page, does not clean index pages (index pages have a similar algorithm), does not update the free space map and the visibility map.

, page pruning was designed specifically for cases where autovacuum wasn't running or couldn't keep up .

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, since they may be referenced from indexes, the server process cannot check this. Only vacuum can free the pointers (make unused pointers) so that the pointer can be used again. In the version data area, dead tuples is cleared and the remaining lines are shifted.

You should not update the same row several times in one transaction , since the space occupied by the generated versions cannot be cleared. This can happen when using AFTER.. FOR EACH ROW triggers. It is worth checking the application code for the presence of the FOR EACH ROW and DEFERRABLE INITIALLY DEFERRED phrases .

https://www.cybertec-postgresql.com/en/reasons-why-vacuum-wont-remove-dead-rows/

In-page cleaning in indexes (repeat)

- performed during index scan
- If a row in a table is deleted, the index entry for the row or version chain may be marked with the LP_DEAD flag.
- the mark can be set by the SELECT command
- no journal entry is created but the block gets dirty
- marked index entry ignored on master but not on replica
- The marked index entry will be cleared when executing commands that change data in the table.

```
create table t (id int primary key, c text) with ( autovacuum_enabled = off);
insert into t SELECT i , 'simple delete ' || i from generate_series (1, 1000000) as
i ;
delete from t where id between 100 and 900000;
analyze t;
explain (analyze, buffers, costs off) select * from t where id between 1 and 900000;
 Index Scan using t_pkey on t (actual time=0.010..218.477 rows=99 loops=1)
Buffers: shared hit=11489
Execution Time: 218.600 ms
```


In-page cleaning in indexes (repeat)

If, during an Index Scan , the server process detects that a row (or a chain of rows referenced by an index entry) has been deleted and has gone beyond the database horizon, then in the index entry of the leaf block (leaf page) in lp_flags the LP_DEAD hint bit is set (called known dead, killed tuple). The bit can be viewed in the `dead column` , returned by **the bt_page_items (' t_idx ', block) function** . When Bitmap Index Scan and Seq Scan are not set. A row marked with this flag will be deleted later when a command that modifies an index block is executed. Why isn't the index space freed immediately ? An index scan is performed by a SELECT , which sets shared locks on the object and pages. Hint bits in both index blocks (flags) and table blocks (infomask and infomask2) can change with such locks. For other changes in the block, an exclusive lock on the block and another lock on the object itself are needed. SELECT will not set them. Because of this, marking the record and freeing up space are separated in time.

Returning to the block and setting a flag in it adds overhead and increases the execution time of the command, but it is done once. However, subsequent commands will be able to ignore the index entry and will not access the table block.

No changes can be made to the block on replicas, and SELECT does not set hint bits on replicas. Moreover, on replicas, LP_DEAD (" ignore_killed_tuples ") set on the master is ignored. Changing the LP_DEAD bit is not logged , but the block is dirty and transmitted via full_page_writes . Because of this feature, **queries on the replica can be executed an order of magnitude slower than on the master . After autovacuum on the master has been processed and the journal entries generated** by autovacuum on the replica have been applied , there will be no difference in speed.

Example of SELECT with bits set on 899900 deleted rows in 7308 table blocks:

`Buffers: shared hit=11489` index and table blocks are read

`Execution Time: 218.600 ms`

The same SELECT again on blocks that haven't been cleared yet:

`Buffers: shared hit=2463` index blocks and several table blocks were read

`Execution Time: 8.607 ms`

After REINDEX or vacuuming the table (the result is approximately the same):

`Buffers: shared hit=6` several index and table blocks were read

`Execution Time: 0.373 ms`

8 -2

Data types

Data types smallest size: `boolean , "char", char, smallint`

- list of data types and their characteristics is in the `pg_type table`
- if the column will be used for searching, it is worth evaluating the efficiency of column indexing, composite indexes, efficiency scanning the index using available methods (Bitmap Index Scan, Index Scan, Index Only Scan)
- Data types that take up the least space:
  - `boolean` takes up 1 byte
  - `" char "` takes 1 byte, stores ASCII characters
  - `char` takes up 2 bytes,
    - stores characters in the database encoding
  - `smallint` , takes 2 bytes
    - stores integers from -32768 to 32767

Smallest data types: `boolean , " char ", char , smallint`

The list of data types and their characteristics can be found in the pg_type table :

```
select typname , typalign , typstorage , typcategory , typlen from pg_type where typtype ='b' and typcategory <>'A' order by typlen,typalign,typname ;
```

Type boolean occupies 1 byte . The " char " type also occupies 1 byte, but stores ASCII characters .

You can confuse "char" with char (synonymous with character(1) or char(1)) . char takes up 2 bytes instead of 1, but stores characters in the database encoding, i.e. more characters than in ASCII encoding :

```
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 | \x31
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 | 26 | 24 | \x0531
```

"char" takes 1 byte, and char 2 bytes. Why lp_off (beginning of line) is the same ? Because there is alignment of the whole line by 8 bytes and it is necessary to remember about it. "char" is intended for use in system catalog tables, but can be used in regular tables. It is necessary to consider how the column will be used. If for searching, then evaluate the efficiency of indexing columns, composite indexes, efficiency scanning the index using available methods (Bitmap Index Scan, Index Scan, Index Only Scan).

The third most compact type is int2 (synonym smallint) , the value of this type takes 2 bytes. It is worth using the name smallint , as it is defined in the SQL standard. The range is -32768 ..32767.

Variable Length Data Types

- for variable length strings it is worth using the text type
- the dimension for text is not specified
- Place occupied:
  - one byte if the field length is less than 127 bytes and the string is empty ''
  - if the encoding is UTF8 , then ASCII characters take 1 byte. Therefore, the value '1' will take 2 bytes: \ x 0531 . `The` value ' 11' will take 3 bytes: \x `073131` . A field consisting of the letter ' `э` ' will take 3 bytes : \x `07d18d`
  - if the field length is more than 126 characters, then the field header will be 4 bytes and the fields will be aligned to 4 bytes
- fields can be compressed and remain in the block
- fields can be TOASTed , leaving 18 bytes in the block (not aligned)
- Binary data should be stored in the bytea data type. This is a variable-length data type and its behavior is the same as the text type.

Variable Length Data Types

Next in compactness are variable-length data types.

For variable-length strings, you should use the text type. The type is not in the SQL standard, but most built-in string functions use text rather than var char. varchar described in the SQL standard . For varchar you can specify the varchar dimension (1..10485760). The dimension for text is not specified. The dimension works as a " domain " (restriction). Checking the restriction takes processor resources. Of course, if the restriction is important for the correct operation of the application (business rules), then you should not refuse them.

Place occupied:

1) First byte allows to distinguish what is stored in the field: a byte with a length (odd HEX values 03, 05, 07... fd , ff) and data up to 126 bytes ; 4 bytes with a length (the first byte is an even HEX value 0c, 10, 14, 18, 20...) ; the field is TOASTed (0x01); the presence of compression is determined by the field size value.

For example: if the field is empty (''), the first byte stores the value \x03. If the field stores one byte, then 0x05, if two bytes - 0x07.

2) if the encoding is UTF8 , then ASCII characters take 1 byte. Therefore, the value '1' will take 1 byte: 31 (in HEX form) . The value '11' will take 2 bytes: 3131. The Cyrillic character ' э ' will take 2 bytes : d18d .

3) Optional zeros. **Fields up to 127 bytes long are not aligned** . Fields over 127 bytes are aligned by pg_type.typalign (i = by 4 bytes) .

Example:

```
drop table if exists t5; create table t5( c1 text default '1' , c2 text default
' eh' , 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 ;
 lp_off | lp_len | t_hoff | t_data
--------+--------+--------+----------------
8144 | 30 | 24 | \x 05 31 07 d18d 03
```

Fields can be compressed and remain in the block. In example **05 07 03** - length of fields.

Fields can be TOASTed , leaving 18 bytes in the block (not aligned).

Binary data should be stored in the bytea data type. This is a variable-length data type and its behavior is the same as the text type. Binary data can be unloaded using the `COPY command` . with the `WITH BINARY option` , otherwise they are exported in text form by default.

Integer data types

- integers can be stored in types `int (integer, int4), bigint (int8), smallint (int2)`
- typically used for PRIMARY KEY columns
- `bigint` aligned to 8 bytes
- `int` for a primary or unique key will limit the number of rows in a table to 4 billion (2 ^ 32)
- to generate values for types `smallint , int` and `bigint` sequences are used and there are synonyms `smallserial (serial2), serial ( serial4 ), bigserial (serial8)`
- to store numbers, the variable length type `numeric` (synonym `decimal`) can be used , **overhead 4 bytes for storing the field length**

Integer data types

Integers can be stored in the int (integer) and bigint types (in addition to smallint). These names are defined in the SQL standard . They correspond to the names int2, int4, and int8. These types are typically used for PRIMARY KEY columns. bigint aligned to 8 bytes. Using int for a primary or unique key will limit the number of rows in the table to 4 billion (2 ^ 32) . The number of fields put into a TOAST table is also limited to 4 billion (2 ^ 32) , but this limit can be reached earlier .

To generate values for types smallint , int and bigint sequences are used and there are synonyms smallserial (serial2), serial (serial4), bigserial (serial8) These are auto-incrementing columns. The numeric types are signed, and if you use only positive numbers, serial uses the range 1 to 2 billion (2147483647), not 4 billion.

The numeric (synonym decimal) length variable type described in the SQL standard can be used to store numbers. **Overhead of 4 bytes to store the field length.**

The range for this type is significant: 131072 digits before the point and 16383 digits after the point. But if you specify numeric(precision, scale) when defining the type, then the maximum precision and scale values are 1000. numeric can be declared with a negative scale: values can be rounded to tens, hundreds, thousands. In addition to numbers and null , numeric supports the values Infinity , -Infinity , NaN .

The advantage of numeric is that columns typically store small numbers, and numeric fields use less space **than** fixed-length decimal types.

To handle decimal numbers , you can use numeric instead of float4(real) or float8(double precision).

Some recommendations for using data types:

https://wiki.postgresql.org/wiki/Don't_Do_This

Selecting data types for the primary key

- with fewer lines the size of the index on a uuid column is significantly larger than on a bigint column
- additional functions are installed by the extension:

```
create extension uuid- ossp ;
ERROR: syntax error at or near "-"
create extension "uuid- ossp ";
CREATE EXTENSION
```

- index size on primary key column and test example:

```
psql -c "create table tt1 (id bigint generated by default as identity (cache 6 0) primary key, data bigint
);"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
tps=2693
psql -c "select count(*), pg_indexes_size ('tt1'), pg_table_size ('tt1') from tt1;"
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
 80760 | 1884160 | 3612672
psql -c "create table tt1 (id uuid default gen_random_uuid () primary key, data bigint );"
pgbench -T 30 -c 4 -f txn.sql 2> /dev/null | grep tps
tps=2338
psql -c "select count(*), pg_indexes_size ('tt1'), pg_table_size ('tt1') from tt1;"
count | pg_indexes_size | pg_table_size
--------+-----------------+---------------
 70115 | 2777088 | 3760128
```


Selecting data types for primary key

uuid type as a unique key (except for those generated by `the uuidv7 () function of the pg_uuidv7` extension , which is available in the Tantor DBMS starting with version 1 6.6) , you need to remember that the field size for this type is 16 bytes and the optimization will not work: " The fastpath optimization avoids most of the work of searching the tree repeatedly when a single backend inserts successful new tuples on the rightmost leaf page of an index " (`nbtinsert.c`). Also, inserting random values, rather than increasing ones, into leaf blocks of an index leads to an increase in the volume of full page writes in the logs .

Example of `bigint` comparison and `uuid` :

```
psql -c "drop table if exists tt1;"
psql -c "create table tt1 (id bigint generated by default as identity (cache 6 0) 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 "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;"
tps=2693
count | pg_indexes_size | pg_table_size
 80760 | 1884160 | 3612672
tps=2338
count | pg_indexes_size | pg_table_size
 70115 | 2777088 | 3760128
```

With fewer lines the size of the index on a uuid column is **significantly** larger than on a bigint column . The second column is filled with a constant, not a function, so as not to measure the speed of the function.

Instead of `gen_random_uuid ()` you can use `uuid_generate_v4()` .

Installing an extension with additional features:

```
create extension uuid- ossp ;
ERROR: syntax error at or near "-"
create extension "uuid- ossp ";
CREATE EXTENSION
```

`uuidv7 ()` may appear in PostgreSQL https://commitfest.postgresql.org/47/4388/

Example of tests: https://ardentperf.com/2024/02/03/uuid-benchmark-war/

cache parameter for sequences

- sequence parameter and column identity
- by default `cache= 1`

```
create table tt1 (id bigint generated by default as identity ( cache 1 ) primary key)
```

- there is no need to increase the value
- sampling values from a sequence is not a bottleneck
- setting it to a higher value results in inefficient inserts into the index created on the column into which the generated values are inserted
- Increasing **`cache`** slows down index inserts

cache parameter for sequences

Parameter sequences and column identity . By default cache=1. Set for sequences and identity columns. Determines how many values the server process will cache in its memory for future inserts. Increasing the value above 1 worsens performance. The reason is that if several sessions cache ranges of values, there is a high probability that inserts will not go to the rightmost leaf block of the index. If inserts do not go to the rightmost block of the index, the index structure becomes inefficient, the index becomes larger than it could be. Selecting values from a sequence is not a bottleneck and is a fast operation. There is no **point in using `cache > 1` .**

In Part 8 of the practice for this chapter, you will measure the speed of insertions for different cache values.

Using the cache parameter would be justified if the database cluster served multiple instances on different nodes and access to the sequence needed to be synchronized between instances.

Storing dates, times, and their intervals

- to store dates, times, intervals the following types are used:
  - date (4 bytes , accurate to the day)
  - `timestamp` , `timestamptz` , `time` precision up to microsecond, size is the same 8 bytes, content is the same
  - `timetz` - length 12 bytes, `interval` - length 16 bytes
- data types `timestamp, timestamptz` do not store time zone
- `timestamptz` converts the stored time to the client's time zone
- `timestamptz` physically stores values in UTC

```
create table t(t TIMESTAMP, ttz TIMESTAMPTZ);
insert into t values (CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
set timezone = 'UTC';
select t, ttz from t;
 2024-11-25 23:19:47.833968 | 2024-11-25 20:19:47.833968+00
update t set ttz =t;
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
--------+--------+--------+--------------------------------------------------------
8096 | 40 | 24 | \x 70580939c1ca020070580939c1ca0200
 8136 | 40 | 24 | \x 7044c4bcc3ca0200 70580939c1ca0200
select t, ttz from t;
 2024-11-25 20:19:47.833968 | 2024-11-25 20:19:47.833968+00
```


Storing dates, times, and their intervals

When storing dates, times, intervals, it is worth considering the size that the values of the selected type will occupy in blocks, as well as whether there are functions, type casts, operators for the selected type.

The most compact type for storing dates is date . The date data type takes up only 4 bytes and stores data with an accuracy of up to a day. The date data type does not store time (hours, minutes). This is not a disadvantage, since you do not need to think about rounding up to a day when comparing dates.

Data types `timestamp, timestamptz` store time and date with microsecond accuracy and occupy 8 bytes. Both types do not store time zone, values are physically stored in the same form .

`timestamptz` Store data in UTC. `Timestamp` data type does not display time zone , does not use time zone, keeps value as is (without conversions) . `timestamptz` displays and performs calculations in the time zone specified by `the timezone` parameter :

```
show timezone ;
Europe/Moscow
create table t(t TIMESTAMP, ttz TIMESTAMPTZ);
insert into t values (CURRENT_TIMESTAMP, CURRENT_TIMESTAMP);
SELECT t, ttz FROM t;
 2024-11-25 23:19:47.833968 | 2024-11-25 23:19:47.833968+03
set timezone = 'UTC';
select t, ttz from t;
 2024-11-25 23:19:47.833968 | 2024-11-25 20:19:47.833968+00
update t set ttz =t;
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
--------+--------+--------+------------------------------
8096 | 40 | 24 | \x 70580939c1ca020070580939c1ca0200 -- current version of the line
8136 | 40 | 24 | \x 7044c4bcc3ca0200 70580939c1ca0200 -- old version of the string
select t, ttz from t;
 2024-11-25 20:19:47.833968 | 2024-11-25 20:19:47.833968+00
```

`Time` data type stores time with microsecond accuracy and also takes up 8 bytes, which is quite a lot.

`Timetz` data type occupies 12 bytes . Data type `interval` takes up the most space, being 16 bytes long . Because of their larger size, these two data types are not practical.

Functions for checking data type and field size

- to check the type of a value, the `pg_typeof ( value ) function is used`
- function to get the field size and the total row size `pg_column_size ( row or field )`
- when storing a string in a block, its size must be a multiple of 8 bytes, if it is less, then 1 to 7 bytes will be added
- for date-time types there is the date type (4 bytes , accurate to a day), the other types are `timestamp , timestamptz , time` have microsecond precision, 8 bytes in size. `Timetz types` And `intervals` are 12 and 16 bytes long and should not be used due to their length.

```
pg_typeof (now()) - > timestamp with time zone
pg_typeof (now()::date) -> date
pg_typeof ( current_date ) - > date
pg_typeof (1.1) -> numeric
pg_column_size (1.1) -> 10
pg_column_size (1.1::float) -> 8
interval type: pg_column_size (interval '1s') -> 16
the size of a string of three fields in a good order: pg_column_size (row(true:: boolean , 1::int4, 1::int8)) -> 40
unsuccessful column rearrangement: pg_column_size (row(true:: boolean , 1::int8, 1::int4)) -> 44
string size with one more order: select pg_column_size (row(1::int4, 1::int8, true:: boolean ))  -> 4 1 ;
```


Data type and size checking functions fields

To check the type of a value, use the `pg_typeof ( value ) function` . It is useful to see if there will be an implicit type cast.

To quickly get the size of a field or row, use the `pg_column_size ( row or field ) function` . When storing a row in a block, it can take up a larger size so that the space occupied is a multiple of 8 bytes. Returns the size in bytes.

If `pg_column_size` is applied to a stored table field and the field has been compressed, it will return the compressed size:

```
select pg_column_size ( relname ) from pg_class limit 1;
 pg_column_size
----------------
64
```

For date-time types there is the date type (4 bytes , accurate to a day), the other types are `timestamp , timestamptz , time` have microsecond precision, 8 bytes in size. `Timetz types` And `interval` have a length of 12 and 16 bytes and due to their length they should not be used:

```
select pg_column_size (interval '1year');
 pg_column_size
----------------
16
```

Function `now()` gives timestamptz transaction start date . To obtain the date (e.g. in DEFAULT or CHECK integrity constraints) of the transaction start date also you can use the `current_date function` , described in the SQL standard as being called without parentheses :

```
create table t (c date constraint c check(c<=now()), c1 date constraint c1
check( c1<= current_date ), c2 date constraint c2 check(c2<= current_timestamp
));
```

To get the command start time as `a timestamptz` you need to use the function `statement_timestamp () ;`

To get the current (at the time of the call) timestamptz , you need to use `the clock_timestamp ()` function ;

Knowing the point at which time is given is important when calculating performance indicators.

Data types for real numbers

- fixed width, floating point, rounded to 6 or 15 digits (significant numbers in decimal format):
  - `float4 , real , float(1..24)` - stores at least in 4 bytes 6 ranks
  - `float8 , float , double precision , float(25..53 )` stores at least 15 digits in 8 bytes

```
select 12345 6 78901234567890123456789.1234567890123456789::float4::numeric;
       1234570 00000000000000000000000
select 12345678901234 5 67890123456789.1234567890123456789::float 8 ::numeric;
       123456789012346 00000000000000
```

- variable width, **without loss of precision** in calculations:
  - `numeric , decimal`
  - precision can be set by parameters: `numeric( presision,scale )`

```
select 1234567890123456789.123456789::numeric + 0.000000000000000000 0123456789 ::numeric;
       1234567890123456789.123456789000000000 0123456789
```

- an example of exponential notation of identical numbers with different mantissa and order :

```
 select 12345.6 ::float4, ' 12.3456 e + 03 '::float4, ' 123.456 e + 02 '::float4, ' 1234.56 e +
01 '::float4;
 1.235e+04 | 1.235e+04 | 1.235e+04 | 1.235e+04
```


Data types for real numbers

Data types for working with real numbers:
1) `float4` synonym `real` synonym `float(1..24)`
2) `float8` synonym `float` synonym for `double precision` synonym `float(25..53)`
3) `numeric` synonym `decimal` .

`float4` provides 6 digits of precision (decimal notation), `float8` provides 15- digit accuracy . The last digit is rounded off:

```
select 12345 6 78901234567890123456789.1234567890123456789::float4::numeric;
       12345 7 00000000000000000000000
select 12345678901234 5 67890123456789.1234567890123456789::float 8 ::numeric;
       12345678901234 6 00000000000000
```

are highlighted in red , which have been rounded. You can also see that digits greater than the sixth and fifteenth digits have been replaced with zeros, meaning that precision is not preserved. The downside of these data types is that adding a small number to a large number is equivalent to adding a zero :

```
select (12345678901234567890123456789.1234567890123456789::float8 +
123456789::float8)::numeric;
12345678901234 6 00000000000000
```

Adding `123456789::float8` is equivalent to adding zero.

Using float can lead to poorly diagnosed errors. For example, a column stores the flight range of an airplane, when testing for short distances, the airplane lands with an accuracy of a millimeter, and when flying for long distances with an accuracy of a kilometer.

When rounding float8, the sixteenth digit is taken into account :

```
select 12345678901234 4 9 99::float8::numeric, 12345678901234 4 4 99::float8::numeric;
       12345678901234 5 000    |            12345678901234 4 000
select 0.12345678901234 4 9 99::float8::numeric, 0.12345678901234 4 4
99::float8::numeric;
       0.12345678901234 5      |              0.12345678901234 4
```

When rounding float4, the seventh digit is taken into account :

```
select 1234 49 9 ::float4::numeric, 12344 4 9 ::float4::numeric;
1234 5 00 | 12344 5 0
select 0.1234 49 9 ::float4::numeric, 0.12344 4 9 ::float4::numeric;
0.1234 5 | 0.12344 5
```

Configuration parameter `extra_float_digits`

- `extra_float_digits` parameter can be used to reduce the number of digits in the text representation of float8, float4 and geometric numbers.
- range of values from -15 to 3 inclusive
- values 1,2,3 are equivalent
- does not affect calculations and casts to the numeric type

```
show extra_float_digits ;
1
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890.1234567 | 1.1234568
set extra_float_digits = 0;
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890.12346 | 1.12346
set extra_float_digits = -5;
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890 | 1
reset extra_float_digits ;
select 234567890.199999989::float8::numeric, 1.19999999123::float4::numeric;
       234567890.2           |          1.2
```


Configuration parameter `extra_float_digits`

Affects text presentation `float8`, `float4` and geometric types. The default value is 1. Values `extra_float_digits` parameter 1,2,3 are equivalent to:

```
show extra_float_digits ;
1
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890.1234567 | 1.1234568
set extra_float_digits = 3 ;
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890.1234567 | 1.1234568
```

The value zero and negative values remove digits from the output. with rounding :

```
set extra_float_digits = 0;
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890.1234 6 | 1.1234 6
set extra_float_digits = -1;
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890.1235 | 1.1235
set extra_float_digits = -2;
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890.123 | 1.123
set extra_float_digits = -5;
select 1234567890.123456789::float8, 1.123456789::float4;
1234567890 | 1
```

The parameter affects only the presentation (display, output) . It does not affect calculations and casts to the numeric type :

```
select 1234567890.123456789::float8::numeric, 1.123456789::float4::numeric;
1234567890.12346 | 1.12346
```

Rounding can remove many digits:

```
reset extra_float_digits ;
select 234567890.199999989::float8::numeric, 1.19999999123::float4::numeric;
       234567890.2           |          1.2
```

Storing real numbers

- Usually, columns store numbers of small dimensions
- numeric type for small integers stores data more compactly than float8 (precision of 15 digits after the point)
- float8 and float4 types lose integer precision , numeric does not
- float4 precision may not be enough: 6 decimal places
- float8 precision 15 decimal places

```
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_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
--------+--------+--------+------------------------------------------------------------------
   7984 |     43 |     24 |\x 3d0ad7a3705c9140 85e38a44 0f008157044c04
   8032 |     41 |     24 |\x 00000000005c9140 00e08a44 0b00805704
   8080 |     49 |     24 |\x 555555555555d53f abaaaa3e 1b7f8a050d050d050d050d050d
   8136 |     36 |     24 |\x 000000000000f03f 0000803f 0b008001 00
```


Storing real numbers

When choosing a data type for storing real numbers, it is worth considering that the numeric type has a variable length and for small numbers stores data more compactly than float8 : precision 15 " decimal places " , digits in decimal form, that is, digits before and after the point in decimal form, that is, if there are not enough digits, decimal and then integer digits are removed and replaced with zeros.

float4 (real) precision may not be enough: 6 digits.

```
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_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
------+------+------+----------------------------------------------------------------------
   8136 |     36 |     24 |\x 000000000000f03f 0000803f 0b008001 00
   8080 |     49 |     24 |\x 555555555555d53f abaaaa3e 1b7f8a05 0d050d050d050d050d
   8032 |     41 |     24 |\x 00000000005c9140 00e08a44 0b008057 04
    7984 |     43 |     24 |\x 3d0ad7a3705c9140 85e38a44 0f008157 044c04
```

Fields of three types are highlighted in color.

The field size check function shows the space occupied by fields more clearly:

```
select pg_column_size (c1), pg_column_size (c2), pg_column_size (c3) from t5;
 pg_column_size | pg_column_size | pg_column_size
----------------+----------------+----------------
              8 |              4 |              5
              8 |              4 |             13
              8 |              4 |              5
              8 |              4 |              7
```

Digit width of division result numeric

- at least 16 significant digits, i.e. no worse than float8
- not less than the bit depth of any of the input parameters

```
 insert into t5 values (1,1, 1.0000000000000000000000000000000000001/3);
 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;
  lp | lp_off | lp_len | t_hoff | t_data
 ----+--------+--------+--------+-----------------------------------------------------------------------------------
 5 | 7912 | 57 | 24 | \x 000000000000f03f 0000803f 2b7f92050d050d050d050d050d050d050d060d
 (5 rows)
 select * from t5 order by ctid desc limit 1;
 c1 | c2 | c3
 ----+----+-----------------------------------------
 1 | 1 | 0.3333333333333333333333333333333333333334
```

- You can view the description of the operators using the psql command

```
 \ 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 | / | numeric | numeric | numeric | numeric_div | divide
 ...
```


Digit width of division result numeric

The digit capacity of the result of dividing two numbers of the numeric type:
1) not less than 16 significant digits, i.e. not worse than float8
2) not less than the bit depth of any of the input parameters .
to calculating square roots and other operations with loss of precision. There is no loss of precision for addition, subtraction, multiplication, and other operators.

Example:

```
 insert into t5 values (1,1, 1.0000000000000000000000000000000000001/3);
 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;
  lp | lp_off | lp_len | t_hoff | t_data
 ----+--------+--------+--------+----------------------------------
 5 | 7912 | 57 | 24 | \x 000000000000f03f 0000803f
2b7f92050d050d050d050d050d050d050d060d
 (5 rows)
 select * from t5 order by ctid desc limit 1;
 c1 | c2 | c3
 ----+----+-----------------------------------------
 1 | 1 | 0.3333333333333333333333333333333333333334
```

You can view the description of the operators using the psql command :

```
 \ doS + /
 List of operators
 Schema | Name | Left arg type | Right arg type | Result type | Function | Descri
 ------------+------+---------------+----------------+---------------+-------------+-------
  pg_catalog | / | bigint | bigint | bigint | int8div | divide
  pg_catalog | / | numeric | numeric | numeric | numeric_div | divide
 ...
```

Comment in `numeric.h file` PostgreSQL source code :

```
/*
* For inherently inexact calculations such as division and square root,
* we try to get at least this many significant digits; the idea is to
* deliver a result no worse than float8 would.
*/
#define NUMERIC_MIN_SIG_DIGITS    16
```

Practice

- Part 1. TOAST
- Part 2. TOAST Table Structure
- Part 3. UPDATE vs. INSERT Efficiency
- Part 4. HOT cleanup
- Part 5. Monitoring HOT cleanup
- Part 6. Small Data Types
- Part 7. Storage of variable-length data types
- Part 8. Data type for the primary key column
- Part 9. Data types for storing dates and times
- Part 10. Data types float and real

Practice

In practice, you will see cases of slowdowns in the commands when triggers are present and evaluate the effects of using different types of triggers. Replacing FOR EACH ROW with FOR EACH STATEMENT can increase productivity several times.

You will see an example where replacing application logic from UPDATE to INSERT and DELETE gives a hundredfold increase in performance.

You will see the performance degradation mechanism when increasing the number of cached sequence values (the CACHE parameter).

The practice is useful for understanding how PostgreSQL stores and works with data. It is useful for those who design storage structures (ER- modeling).

9-1

Architecture

Starting an instance, postgres process

- postgres process is starting
- parameter files are read and combined with parameters passed on the command line
- permissions on the PGDATA directory are checked
- `pg_control` control file is checked
- memory is allocated, shared libraries are loaded, event handlers are registered
- postmaster.pid file is created in PGDATA , the presence of which and the correctness of the process number are checked once per minute
- postgres process registers server sockets with the TCP_NODELAY and TCP_KEEPALIVE parameters , a UNIX socket file is created
- `pg_hba.conf` file is read
- startup process and background processes are launched

Launch instance, process postgres

The basic steps to launch an instance are:

1. The postgres process ("postmaster") is started. The parameters LC_MONETARY = "C", LC_NUMERIC="C", LC_TIME="C" are set, unserenv LC_ALL is removed , the variables LC_COLLATE , LC_CTYPE , LC_MESSAGES are set to the environment variables of the same name. LC_MESSAGES (messages in stdout , stderr or log) will be set later by the configuration parameter lc_messages
2. Configuration parameter files are read, the parameters are combined with command line parameters passed to postmaster and environment variables.
3. The rights to the PGDATA directory are checked, they should be 0700 or 0750
4. The presence of the pg_control file is checked , the current directory for the process is set to PGDATA, the postmastr.pid file is created in it , TLS is initialized , shared libraries are loaded using the shared_preload_libraries parameter, a handler is registered in case the process disappears for the correct termination of child processes, the memory manager is initialized (using configuration parameters), and a handler for closing network sockets is registered.

In postmaster.pid the first line stores the PID of the running postmaster . It is checked once per minute. If the file does not exist or the PID stored in it is not equal to the PID of the process , the postgres process will be stopped by the SIGQUIT signal.

5. sockets are registered at all addresses (configuration parameter listen_addresses). A UNIX socket file is created . The host of the first successfully opened socket (TCP or UNIX) is added to LockFile . TCP_NODELAY and TCP_KEEPALIVE parameters will be set for sockets
6. The pg_hba.conf file is read
7. The s startup process is launched and it is assigned the STARTUP_RUNNING status, and the postgres process assigns itself the PM_STARTUP status. The background processes are started.

Postmaster stores the PIDs of eight main background processes:

```
/* PIDs of special child processes; 0 when not running */
static pid_t StartupPID = 0, BgWriterPID = 0, CheckpointerPID = 0, WalWriterPID
= 0,WalReceiverPID = 0, AutoVacPID = 0, PgArchPID = 0, SysLoggerPID = 0;
```

All spawned processes, including server ones, are periodically checked for existence.

Startup process

- startup process determines the cluster state from the control file. Possible states are:
  - " `shut down` " after a correct (with checkpoint) stop of the instance
  - " `shut down in recovery` " the replica instance crashed while in recovery mode
  - " `in production` " instance was stopped without a checkpoint
  - " `shutting down` " instance crashed while shutting down
  - " `in crash recovery` " the master instance crashed during recovery
  - " `in archive recovery` " instance failed during managed recovery
  - " `uncognized status code` " status unknown
- startup process performs synchronization of PGDATA files and their restoration from WAL files

```
postgres@tantor :~$ pg_controldata | grep state
Database cluster state: shut down
```


Launching the startup process

Process startup performs cluster recovery. The actions are programmed in `the StartupXLOG () function` . If the process runs longer than specified in the `log_startup_progress_interval = 10s configuration parameter, then the` startup process will write information about what it is doing to the cluster log at the frequency specified by this parameter . The instance state can be viewed in the control file:

```
pg_controldata | grep state
Database cluster state: in production
```

1) The state of the instance, determined by the control file , is written to the log. List of possible states `by control` file (`pg_control`) and log entries :

a) `"shut down"` log entry: `database system was shut down at` after a correct (with checkpoint) shutdown of the instance

b) `"shut down in recovery" database system was shut down in recovery at` replica instance crashed while in recovery mode

c) `"in production" database system was interrupted; last known up at` instance was stopped in immediate mode or crashed (e.g. power was lost)

d) `"shutting down" database system shutdown was interrupted; last known up at` instance crashed during shutdown

e) `"in crash recovery" database system was interrupted while in recovery at time. HINT: This probably means that some data is corrupted and you will have to use the last backup for recovery.` the master instance crashed during recovery ("crash recovery")

f) `"in archive recovery" the database system was interrupted while in recovery at log` time . `HINT: If this has` occurred `more than once some data might be corrupted and you might need to choose an earlier` recovery target.

g) `"unrecognized status code" c control file contains invalid database cluster state` - status unknown

2) For all states except "a" and "b" from `pg_wal` uncreated ones are deleted WAL files to keep the directory clean. The failure could have occurred during the creation of the WAL file.

3) For all states except "a" and "b" `the SyncDataDirectory ()` function is launched , which flushes the Linux cache pages by files in PGDATA and tablespace directories

synchronization , parameter recovery_init_sync_method

- startup process performs PGDATA synchronization using the method specified in the `recovery_init_sync_method configuration parameter.`
- by default `recovery_init_sync_method=` **`fsync`** , which means the startup process will open and send fsync to ALL files in PGDATA :

```
LOG: database system was interrupted; last known up at 03:17:41 MSK
LOG: syncing data directory ( fsync ), elapsed time: 10 .00 s, current path: ./base/5/4066825
LOG: syncing data directory ( fsync ), elapsed time: 20 .00 s, current path: ./base/5/1903193
...
LOG: syncing data directory ( fsync ), elapsed time: 70 .00 s, current path: ./base/5/4081550
LOG: database system was not properly shut down; automatic recovery in progress
```

- `recovery_init_sync_method=` **`syncfs`** . With this value, sync is performed on all mounted file systems where PGDATA is located

PGDATA synchronization , parameter recovery_init_sync_method

For all states except "a" and "b" SyncDataDirectory () performs fsync across all PGDATA in the manner specified in the recovery_init_sync_method configuration parameter .

By default, `recovery_init_sync_method = fsync` , which means the startup process will open and send fsync for ALL files in PGDATA and files in directories pointed to by the PGDATA/pg_wal symbolic link (if any) and symbolic links in the PGDATA/ pg_tblspc directory (tablespace directories) . No synchronization will be performed on other symbolic links. This is done to ensure that all files in PGDATA are safely saved to disk before the WAL rollforward. There is no guarantee that PGDATA file blocks were physically written to disk as a result of an improper instance shutdown. fsync sent to data files at the end of a checkpoint.

For all states except "a" and "b" means that SyncDataDirectory () is also executed for restoration from backups created by the pg_basebackup utility . Synchronization time increases the downtime of DBMS user service during restoration.

If there are hundreds of thousands of files in the cluster, synchronization for each file will take a long time. The time does not depend on whether there is a page cache linux dirty pages or not, all files will open.

The default value is set based on the fact that there are few files in the cluster, and the file system where PGDATA is located , the operating system actively works with files of other applications, for example, the host is used as a file server. For industrial DBMS, this is not the case: other applications usually do not work on the host with the DBMS.

It is recommended to set the configuration parameter value recovery_init_sync_method = syncfs . This value performs sync on the mounted filesystems as a whole, where PGDATA , tablespaces, and the `PGDATA/pg_wal directory are located` . **This value will perform sync much faster than opening each file and fsyncing it.**

After synchronizing PGDATA, the startup process starts rolling back logs using the InitWalRecovery (..) function and other actions. The rollback procedure starts from the start of the checkpoint specified in the bacup_label file or control file may take time. The roll-forward is performed by one startup process sequentially, but it uses preliminary reading of log blocks (prefetching), so startup works quickly and there is no need to think that it needs to be " parallelized " .

Backup synchronization , `pg_basebackup --sync-method parameter`

- `pg_basebackup` utility has a parameter `-- sync-method` with two values fsync and syncfs
  - fsync default value
- Before version 17, you can use the `--no -sync` (`-N`) parameter and then the operating system command `sync - f` .
- when there are a large number of files in the cluster, and therefore in the created backups , use syncfs or sync allow to reduce the backup time with the same fault tolerance . Example:

```
rm - rf / var /lib/ postgresql /backup/1
time pg_basebackup -c fast -D $HOME/backup/1 -P
3783151/3783151 kB (100%), 1/1 tablespace
real 2m50 .010s
user 0m52.035s
sys 0m3.515s
rm - rf / var /lib/ postgresql /backup/1
time pg_basebackup -- sync-method= syncfs -c fast -D $HOME/backup/1 -P
real 0m40 .807s
user 0m36.967s
sys 0m2.612s
```


Backup synchronization , parameter `pg_basebackup --sync-method`

The number of files can be large not only due to the number of objects, but also the size of the objects. The maximum file size is 1GB.

In version 17 of the `pg_basebackup utility --sync-method parameter` with two fsync values appeared and syncfs . The default value is fsync .

After the backup is complete, the utility either performs synchronization at the file system level where the backup was performed, or fsync for all files that I put in the backup directory . When using fsync All files in all backup directories are opened in turn . Both methods guarantee that data is saved on the disk.

Before version 17, you can use the `--no-sync` (`-N`) parameter and then the operating system command **`sync - f`** to synchronize file systems.

With a large number of files in the cluster, and therefore in the created backups , use syncfs or sync allow you to reduce the backup time with the same fault tolerance.

Example:

```
rm - rf / var /lib/ postgresql /backup/1
time pg_basebackup -c fast -D $HOME/backup/1 -P
3783151/3783151 kB (100%), 1/1 tablespace
real 2m50.010s
user 0m52.035s
sys 0m3.515s
rm - rf / var /lib/ postgresql /backup/1
time pg_basebackup --no-sync -c fast -D $HOME/backup/1 -P
real 0m40.807s
user 0m36.967s
sys 0m2.612s
time sync
real 0m0.015s
user 0m0.001s
sys 0m0.000s
```

Parameter `restart_after_crash`

- default value `is on` and this increases the availability of the instance, since when disconnected the instance will simply be forcefully stopped
- Determines whether instance processes will be restarted if the server process fails:

```
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)
```

- When changing the value, it is enough to reread the configuration
- The parameter can be set to `off` if the instance is managed by cluster software. and it decides whether to launch the instance or not

Parameter `restart_after_crash`

When a server process crashes, the process behavior postgres is defined by 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 setting is enabled and this increases the availability of the instance, since disabling it will simply force the instance to stop.

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 " crash recovery " - starting the instance after a failure). If you disable the parameter, all instance processes will be forcibly stopped, including the postgres process. The state of the cluster after stopping will be the same as after a forced stop using the `pg_ctl utility stop - m immediate` :

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

This option is typically disabled when the instance is managed by cluster software for which auto-start of the instance is not desired.

Regardless of the parameter value, the crash procedure is executed when processes are started. recovery : synchronization and rollback of logs.

The server process may crash due to a SIGKILL signal being sent from OOM kill when there is insufficient memory.

Features of running an instance in a docker container

- Modifiable files, in particular PGDATA, must be located on volumes
- running in a container does not add high availability
- postgres process should **not** have PID=1
- when creating and running a container, you need to use `the docker run -d -- init parameter`

```
root@tantor :~# docker exec container_name p.s.
  PID USER TIME COMMAND
    1 postgres 0:38 postgres
31 postgres 0:09 postgres: logger
32 postgres 0:45 postgres: checkpointer
33 postgres 0:38 postgres: background writer
...
root@tantor:~# docker rm -f container_name
root@tantor :~# docker run --init -d -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 container_name
```


Features of running an instance in a docker container

number (PID) of the postmaster in the container must not be equal to one (1). The process with PID= 1 is the first user process that starts after the Linux kernel initialization . The process with 1 spawns (starts) all other processes. It is the parent of all other processes it spawns. All processes must have a parent process. Process 1 has a property : if the parent process of some process dies, the kernel automatically assigns the process 1 parent for orphaned process. Process 1 must adopt all orphans.

postgres process monitors the state of its child processes and receives an exit status when any child process terminates. The normal behavior of postmaster if a child process terminates with a status other than 0 (normal termination) is to restart the instance. In addition to breaking sessions, the instance will be unavailable while the wal log is being restored .

In a Docker container , process 1 is the process that is started to run the container. The postgres process **should not have PID= 1** :

```
root@tantor:~# docker exec -it container /usr/bin/ps -ef
PID USER TIME COMMAND
  1 postgres 0:38 postgres
```

To use tini To start an instance in a container, you need to use the `--init option` .

Modifiable files, in particular PGDATA , must be located on volumes , otherwise the data will be lost when the container is deleted . Example of creating and running a container:

```
sudo docker pull postgres
sudo docker run -d --init -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
```

Running an instance in a container does not add high availability.

Running an instance in a container provides greater performance than running it in a virtual machine.

https://www.cybertec-postgresql.com/en/docker-sudden-death-for-postgresql/

What happens when a server process starts

- for each session a server process is created
- gets the structure (part of memory) PGPROC from the free list and sets the fields to initial values
- lists of free PGPROCs for server processes are stored in the FreeProcs field PROC_HDR structures , all of which are in shared memory
- Three caches are initialized (allocated and filled) in the local memory of the server process:
  - Cache for fast access to tables (RelationCache)
  - System Catalog Table Cache (CatalogCache)
  - Command plan cache (PlanCache)
- memory is allocated for the " portals " manager TopPortalContext
- the client is authenticated
- shared libraries specified in the parameters are loaded `session_preload_libraries` and `local_preload_libraries`
- MessageContext memory is allocated for commands and the process is ready to receive commands

What happens when a server process starts

The main steps in starting a server process are:

1. When starting, the process gets a PGPROC structure (part of the memory) from the free list and sets the fields to the initial values . Free PGPROC lists for server processes are stored in the FreeProcs field. PROC_HDR structures . The structures are in shared memory. Write access to PROC_HDR uses calls to SpinLockAcquire , SpinLockRelease . Access to the fields of these structures (one PGPROC ~ 8 80 bytes, size multiple of 16 bytes) is frequent from each process. They are stored in 4Kb pages, references to which will occupy part of the records in the TLB of processors, and the fields that are frequently accessed will take up space in the caches of processors of various levels. Then each process changes the fields in its PGPROC structure to meaningful values . Then the process receives the process ordinal number (id) in the shmInvalBuffer array .

2. The process registers timeouts:

```
/* Identifiers for timeout reasons. Note that in case multiple timeouts trigger at the same time, they are serviced in the order of this enum . */
typedef enum TimeoutId
{ /* Predefined timeout reasons */
STARTUP_PACKET_TIMEOUT,
DEADLOCK_TIMEOUT,
LOCK_TIMEOUT,
STATEMENT_TIMEOUT,
STANDBY_DEADLOCK_TIMEOUT, STANDBY_TIMEOUT,
STANDBY_LOCK_TIMEOUT,
IDLE_IN_TRANSACTION_SESSION_TIMEOUT,
IDLE_SESSION_TIMEOUT,
CLIENT_CONNECTION_CHECK_TIMEOUT,
USER_TIMEOUT, /* First user-definable timeout reason */
MAX_TIMEOUTS = USER_TIMEOUT + 10 /* Maximum number of timeout reasons */
} TimeoutId ;
```

3. Three caches are initialized in the local memory of the server process:

Cache for fast access to tables (RelationCache)

System Catalog Table Cache (CatalogCache)

Command plan cache (PlanCache)

4. Memory is allocated for the " portals " manager TopPortalContext . Portal is an executable query that appears in the extended protocol at the binding stage, after parsing . Portals can be named (for example, the name of the cursor) or unnamed - SELECT.

5. The delay is performed by the `pre_auth_delay parameter` . The PgBackendStatus structure is initialized . Authentication is performed.

6. The values of the configuration parameters that are set at the connection stage are updated. A delay is performed according to the post_auth_delay parameter.

7 . The PgBackendStatus structure is updated .

8. The following parameters are sent to the client: server version, time zone, localization parameters, data type formats, a pair of process ordinal numbers (id) and a cancellation token , by which the client can cancel the execution of the request.

9. The server process loads the libraries specified in the parameters `session_preload_libraries are loaded` and `local_preload_libraries` . During the loading process, the compatibility of libraries with the PostgreSQL version is checked . If the library was loaded earlier (`shared_preload_libraries`) , the process simply receives a pointer to the loaded library.

10. Memory is allocated for processing messages from the client (MessageContext), r ow_description_context - description of the rows when responding to the client (column name, column data type (object ID) or type modifiers) , input_message - storage of the incoming request row.

11. The ReadyForQuery message is sent to the client . If the command is received from the client, the IDLE_IN_TRANSACTION_SESSION_TIMEOUT and IDLE_SESSION_TIMEOUT timers are reset .

The client can send messages:

```
 case 'Q': /* simple query */
 case 'P': /* parse */
 case 'B': /* bind */
 case 'E': /* execute */
 case 'F': /* fastpath function call start transaction */
 case 'C': /* close close portal or named query */
 case 'D': /* describe */
 case 'H': /* flush send buffer */
 case 'S': /* sync commit transaction */
 case 'X': /* 'X' means that the frontend is closing down the socket. */
 case EOF: /* EOF means unexpected loss of frontend connection. Either way,
perform normal shutdown. */
 case 'd': /* copy data ignore packets, can be sent with copy command */
 case 'c': /* copy done */
 case 'f': /* copy fail */
```

Examples of structures in shared memory of an instance:

```
 Proc Array , PROC, PROCLOCK, Lock Hashes, LOCK, Multi-XACT Buffers, Two-Phase
Structs , Subtrans Buffers, CLOG Buffers, XLOG Buffers, Shared Invalidation,
Lightweight Locks, Auto Vacuum, Btree Vacuum, Buffer Descriptors, Shared Buffers,
Background Writer Synchronized Scan, Semaphores, Statistics
```

Examples of structures in the local memory of the server process:

```
 Relation C ache, Catalog C ache, Plan C ache, work_mem , maintenans_work_mem ,
StringBuffer , temp_buffers
```

PGPROC is used not only by server processes, but also by autovacuum launcher and workers , background worker processes (bgworkers) , walsender and other auxiliary processes, processes servicing distributed transactions.

Shared memory of instance processes

- there may be more than 72 structures
- The sizes of most structures can be viewed in the `pg_shmem_allocations view`
- the size of some structures changes with configuration parameters
- the sizes of all hash structures are given incorrectly

```
 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> | | 4946048 | 4946048 |
  Archive Data | 147726208 | 8 | 128 | 128
 ...
 XLOG Recovery Ctl      | 4377728 | 104 | 128 | 128
                          | 148145024 | 2849920 | 2849920 |
 (60 rows)
 select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations ) as a where a.true <> a.allocated_size order by 1;
 LOCK hash | 142583808 | 2896      | 2944 |    695168
 PREDICATELOCK hash | 144423680 | 2896      | 2944 |    1260416
 ...
```


Shared memory of instance processes

LWLock (lightweight) locks are used to obtain both read and write access . Extensions may create their own structures. List of structures and their sizes:

```
 select * from (select *, lead(off) over(order by off)-off as true from
pg_shmem_allocations ) as a order by 1;
 name | off | size | allocated_size | true
----------+----------+----------+----------------+----------
 <anonymous> | | 4946048 | 4946048 |
  Archive Data | 147726208 | 8 | 128 | 128
 ...
 XLOG Recovery Ctl | 4377728 | 104 | 128 | 128
 | 148145024 | 2849920 | 2849920 |
 (60 rows)
```

A string with an empty (NULL) name reflects unused memory. A string with the name "<anonymous>" reflects the total size of the structures for which memory was allocated by the function call. ShmemAlloc ( ), not by the ShmemInitStruct (" name ..) and ShmemInitHash (" name ..) functions .

By names you can search in the source code for places where memory is allocated. For example:

```
procs = (PGPROC *) ShmemAlloc ( TotalProcs * sizeof (PGPROC));
```

Sizes of all The hash structures are not returned correctly , but their size can be estimated .

The view does not show structures that are allocated and freed " dynamically " - during the operation of the instance. Dynamic shared memory structures are used by worker processes . Workers are used, in particular, to execute SQL commands in parallel . By default, it is allocated by calling shm_open () .

Configuration parameter `min_dynamic_shared_memory` can allocate pages of type HugePages to be used by processes that allocate memory dynamically. If the memory specified by this parameter is not enough, it will be allocated in the usual way: in 4K pages , unless Transparent HugePages is used .

Only `shared_buffers` and `min_dynamic_shared_memory` can use HugePages .

System Catalog Table Cache

- allocated in the local memory of each process in the context of CacheMemoryContext
- When an object is created or deleted, the process that committed the transaction sends a message to the shared memory ring buffer shmInvalBuffer
- The buffer stores up to 4096 messages
- processes consume messages and update their local caches
- if the process misses messages, it will completely clear its local cache of system catalog tables (CatalogCache) and will fill it up again

```
select * from (select *, lead(off) over(order by off) - off as true_size
from pg_shmem_allocations ) as a where name=' shmInvalBuffer ' order by 1;
 name | off | size | allocated_size | true_size
----------------+-----------+-------+----------------+----------
  shmInvalBuffer | 146865024 | 68128 | 68224 | 68224
(1 row)
```


System Catalog Table Cache

CatalogCache allocated in the local memory of each process in the context of CacheMemoryContext . When accessing the system catalog tables, the process searches for data in this cache . If no data is found, then the rows of the system catalog tables are selected and cached. The index access method is used to access the system catalog tables. If no entry is found in the system catalog table, then the absence of an entry flag (negative entry) is cached . For example, a table is searched for , and there is no such table, a record is saved in the local cache of the process that there is no table with such a name. There are no limits on the size of CacheMemoryContext , It is not a circular buffer or a stack.

When committing a transaction creating, deleting, modifying an object, leading to changes in the system catalog tables, the process that performed the changes saves a message that the object has been modified in the ring buffer shmInvalBuffer in shared memory. The buffer can store up to 4096 messages (the constant MAXNUMMESSAGES). Buffer size:

```
 select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations ) as a where name=' shmInvalBuffer ' order by 1;
 name | off | size | allocated_size | true_size
----------------+-----------+-------+----------------+----------
  shmInvalBuffer | 146865024 | 68128 | 68224 | 68224
```

If a process has not consumed half of the messages, it is notified to consume the accumulated messages. This reduces the likelihood that the process will miss messages and will have to clear its local system directory cache. Shared memory stores information about which processes have consumed which messages . If a process, despite the notification, does not consume messages and the buffer is full, the process will have to completely clear its system directory cache.

To prevent the process system directory caches from being flushed too often, objects (including temporary tables) should not be created or deleted too often. Tables, including temporary ones, should not be created or deleted too often during a session.

Statistics of cache resets and number of messages is not built with standard PostgreSQL extensions .

Performance pg_stat_slru

- PGDATA contains subdirectories where cluster service data is stored
- to speed up read/write access to files in these directories, caches in the instance's shared memory are used
- statistics are used to set configuration parameters that determine the sizes of SLRU caches

```
 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 | 103 | 0
  multixact_member | 0 | 0 | 0 | 0 | 103 | 0
  multixact_offset | 0 | 3 | 2 | 0 | 103 | 0
 notify | 0 | 0 | 0 | 0 | 0 | 0
  serializable | 0 | 0 | 0 | 0 | 0 | 0
  subtransaction | 0 | 0 | 26 | 0 | 103 | 102
 transaction | 349634 | 4 | 87 | 0 | 103 | 0
 other | 0 | 0 | 0 | 0 | 0 | 0
 (8 rows)
```


View pg_stat_slru

IN PGDATA are subdirectories that store cluster service data. To speed up read/write access to files in these directories, caches are used in the instance's shared memory. Files are formatted in 8Kb blocks. Caches operate using a simple algorithm for displacing long-unused data (simple least-recently-used , SLRU). Cache usage statistics can be viewed in the view:

```
 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 | 103 | 0
  multixact_member | 0 | 0 | 0 | 0 | 103 | 0
  multixact_offset | 0 | 3 | 2 | 0 | 103 | 0
 notify | 0 | 0 | 0 | 0 | 0 | 0
  serializable | 0 | 0 | 0 | 0 | 0 | 0
  subtransaction | 0 | 0 | 26 | 0 | 103 | 102
 transaction | 349634 | 4 | 87 | 0 | 103 | 0
 other | 0 | 0 | 0 | 0 | 0 | 0
```

Reset statistics : `select pg_stat_reset_slru (null);`

You can pass the cache name as an argument, or NULL if you want to reset statistics for all caches .

In PostgreSQL starting with version 17 (in Tantor DBMS starting with version 15), cache sizes are configurable.

The statistics from the view can be used to set configuration parameters that specify the sizes of SLRU caches : `\ 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
```

https://docs.tantorlabs.ru/tdb/ru/16_4/se/monitoring-stats.html

Local process memory

- is accessible only to one process, so no locks are needed to access it
- most of the structures do not take up much memory and are only interesting for understanding the algorithms of the processes
- The parameters that most strongly influence the allocation of local process memory are:
  - `work_mem` - allocated for servicing nodes (steps) of the execution plan (if the steps can be executed simultaneously) , including each parallel process. Together with the `hash_mem_multiplier parameter` , it affects the memory allocated by each server and parallel process.
  - `maintenance_work_mem` default value is 64MB. Specifies the amount of memory allocated by each process (server, parallel) participating in the execution of `the VACUUM , ANALYZE, CREATE INDEX , ALTER TABLE ADD FOREIGN KEY commands`

Local process memory

Local memory is accessible only to one process, so locks for accessing it are not needed. Memory is allocated for various structures (" contexts "). A universal set of functions is used to allocate and account for the allocated memory, rather than situational calls to the operating system. Most structures do not take up much memory and are interesting only for understanding the algorithms of the processes. Of interest are those structures that are large in size or whose size can be influenced, for example, by configuration parameters.

The parameters that most strongly influence the allocation of local process memory are:

work_mem - allocated for servicing nodes (steps) of the execution plan (if the steps can be executed simultaneously) , including each parallel process . Together with the hash_mem_multiplier parameter , it affects the memory allocated by each server and parallel process. For example, when joining tables using hashing (Hash Join) , the amount of memory allocated for servicing the JOIN will be work_mem * hash_mem_multiplier * (Workers + 1) .

maintenance_work_mem default value is 64MB. Specifies the amount of memory allocated by each process (server, parallel) participating in the execution of the VACUUM , ANALYZE, CREATE INDEX , ALTER TABLE ADD FOREIGN KEY commands. The number of parallel processes is limited by the max_parallel_maintenance_workers parameter . The creation of indexes and normal (without FULL) vacuum . When vacuuming only the index vacuuming phase (other phases are not parallelized), one index can be processed by one (rather than several) parallel processes. Whether parallel processes are used depends on the size of the indexes.

enable_large_allocations by default false. The amount of memory that a process gradually allocates for processing one line The sample size is limited to 1GB. A process can allocate up to 2GB of memory to process a string, in addition to all other memory it uses. This parameter allows you to increase the size of the string buffer (StringBuffer) to 2GB.

Performance pg_backend_memory_contexts

- shows the memory allocated by the server process serving **the current** session
- memory contexts - a set of memory parts (chunks) , which are allocated by a process to perform some task
- a child context can be allocated to execute a subtask
- Contexts form a tree (hierarchy)
- In the hierarchy view, the columns display: name (name of the memory context) , parent (name of the parent memory context , level
- in the ident column contains details of what is stored in the context

```
 select sum( total_bytes ), sum( used_bytes ), sum( free_bytes ) from
pg_backend_memory_contexts ;
 sum | sum | sum
---------+---------+--------
2114816 | 1380760 | 734056
```


Performance pg_backend_memory_contexts

The view shows the memory allocated by the server process serving **the current** session. Memory contexts - a set of memory parts (chunks) , which are allocated by the process to perform some task. If there is not enough memory, it is allocated additionally. A child context can be allocated to perform a subtask. Contexts form a tree (hierarchy). The root of the tree is TopMemoryContext . The purpose of such an organization of memory allocation and accounting is to not forget to free some part when freeing memory, otherwise a memory " leak " will occur . When a memory context is freed, all child memory contexts are freed.

In the `pg_backend_memory_contexts view` , the hierarchy is displayed by the columns: name (name of the memory context) , parent (name of the parent memory context) , level. In the ident column contains details of what is stored in the context. Example of a hierarchical query:

```
 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 * from dep limit 3;
 name | total|ident | parent |level| path
 --------------------+-----+-----+-----------------+-----+-------------------------------
 TopMemoryContext |97664| | | 1 | TopMemoryContext
 TopTransactionContext | 8192| | TopMemoryContext | 2 | TopMemoryContext -> TopTransactionContext
 PLpgSQL cast cache | 8192| | TopMemoryContext | 2 | TopMemoryContext -> PLpgSQL cast cache
 (3 rows)
```

Memory allocated to the current server process:

```
 select sum( total_bytes ), sum( used_bytes ), sum( free_bytes ) from
pg_backend_memory_contexts ;
 sum | sum | sum
---------+---------+--------
2114816 | 1380760 | 734056
```

In future versions of the view, the columns id , parent_id , path will be added .

Function `pg_log_backend_memory_contexts (PID)`

- Since version 17, the EXPLAIN command has a memory option (disabled by default), which displays how much memory the scheduler used and the total memory of the server process
- The memory of other sessions can be output to the cluster diagnostic log using the function:

```
postgres=# SELECT pg_log_backend_memory_contexts (11 1 );
 pg_log_backend_memory_contexts
--------------------------------
t
(1 row)
LOG: statement: SELECT pg_log_backend_memory_contexts (111);
...
LOG: logging memory contexts of PID 111
LOG: level: 0; TopMemoryContext : 60528 total in 5 blocks; 16224 free (6 chunks); 44304 used
LOG: level: 1; TopTransactionContext : 8192 total in 1 blocks; 6728 free (0 chunks); 1464
used
...
LOG: level: 2; AV dblist : 8192 total in 1 blocks; 7840 free (0 chunks); 352 used
LOG: Grand total: 658848 bytes in 38 blocks; 270616 free (32 chunks); 388232 used
```


Function `pg_log_backend_memory_contexts (PID)`

The memory of other sessions can be output to the cluster diagnostic log using the function:

`select pg_log_backend_memory_contexts ( PID ) ;`

to the log:

```
LOG: statement: SELECT pg_log_backend_memory_contexts (111);
...
LOG: logging memory contexts of PID 111
LOG: level: 0; TopMemoryContext : 60528 total in 5 blocks; 16224 free (6 chunks); 44304 used
LOG: level: 1; TopTransactionContext : 8192 total in 1 blocks; 6728 free (0 chunks); 1464 used
...
LOG: level: 2; AV dblist : 8192 total in 1 blocks; 7840 free (0 chunks); 352 used
LOG: Grand total: 658848 bytes in 38 blocks; 270616 free (32 chunks); 388232 used
```

The function appeared in version 14 of PostgreSQL and was available only to users with the SUPERUSER attribute. In version 15, it became possible to grant the privilege to execute the function to an unprivileged user:

```
postgres=# create role alice ;
postgres=# grant execute on function pg_log_backend_memory_contexts to alice ;
postgres=# drop role alice ;
ERROR: role " alice " cannot be dropped because some objects depend on it
DETAIL: privileges for function pg_log_backend_memory_contexts (integer)
postgres=# revoke all on function pg_log_backend_memory_contexts from alice ;
postgres=# drop role alice ;
```

Starting with version 17, the EXPLAIN command has a memory option (disabled by default), which displays how much memory the scheduler used and the total memory of the server process as a line at the end of the plan:

```
Memory: used=N bytes, allocated=N bytes
```

During the planning phase, using a large (thousands) number of partitions on a partitioned table can consume a lot of memory.

The memory for TID store during vacuuming is taken into account in the lines:

```
level: 1; TopTransaction Context : 33570864 total in 3 blocks; 11056 free (405 chunks); 33559808 used
level: 2; _ bt_pagedel : 8192 total in 1 blocks; 7928 free (0 chunks); 264 used
Grand total: 35510408 bytes in 234 blocks; 736144 free (626 chunks); 34774264 used
```

Memory of size `maintenance_work_mem` is allocated in the context of (memory for) the transaction. After When a transaction is executed during the vacuum process , the transaction context memory is freed.

9-2

Blockages

Types of locks

- spinlock (cyclic check)
  - Used for very short-term actions - no longer than a few dozen processor instructions
  - There are no monitoring tools
- lightweight (LWLocks)
  - Used to access structures in shared memory.
  - They have exclusive (read and modify) and shared (read) modes.
  - no more than 200 at a time
- regular (heavy)
  - Automatically released upon completion of the transaction
  - There are several levels of blocking
  - Serves **12 types of locks** , including advisory locks.
- predicate locks (SIReadLock) are used by transactions with the SERIALIZABLE isolation level

Types of locks

The instance uses locks for interprocess communication:

1) spinlock (cyclic check) . Used for very short-term actions - no longer than a few dozen processor instructions. Not used if an input-output operation is being performed, since the duration of such an operation is unpredictable. It is a variable in memory that is accessed by atomic processor instructions. A process wishing to obtain a spinlock checks the status of the variable until it is free. If the lock cannot be obtained within a minute, an error is generated. There are no monitoring tools.

2) Lightweight (LWLocks). Used to access structures in shared memory. Have exclusive (read and change) and shared (read) modes. There is no deadlock detection, they are automatically released in case of failure. The overhead of obtaining and releasing a lock is small - several dozen processor instructions, if there is no conflict for the lock. Waiting for the lock does not load the processor. Processes obtain the lock in the order of the queue. There are no timeouts for obtaining lightweight locks. When accessing LWLock structures, spinlocks are used . The number of LWLocks is limited by a constant: `MAX_SIMUL_LWLOCKS = 200` . There are more than 7 3 named LWLocks , tranches of which protect access to structures in shared memory. Their names appear in wait events. Examples of names: XactBuffer , CommitTsBuffer , SubtransBuffer , WALInsert , BufferContent , XidGenLock , OidGenLock .

3) Regular (heavyweight). Automatically released at the end of the transaction. There is a procedure for detecting and resolving deadlocks. There are several levels of locks. Serves locks at the level of 12 object types (LockTagTypeNames) .

4) Predicate locks (SIReadLock) - used by transactions with the SERIALIZABLE isolation level .

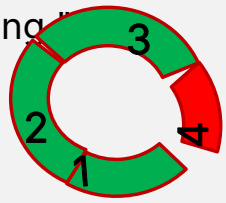
Parallel processes are combined into a group with their server process (group leader). In version 16, processes in a group do not conflict, which is implemented by the algorithm of their work. Parallelism develops and the logic of blocking can develop.

One of the lock types (`pg_locks.locktype`): advisory locks (application-level locks, user-level locks), can be obtained at the session and transaction level, managed by the application code.

While waiting to acquire a lock, the process does not perform useful work, so the shorter the time it waits to acquire locks, the better.

Parameters `deadlock_timeout` And `log_lock_waits`

- the amount of time to wait for a lock to be acquired before the waiting process begins checking for deadlocks
- one check in one waiting cycle
- default is 1 second , the value should be increased at least up to the duration of a typical application transaction
- It is recommended to set `log_lock_waits = true` to get messages in the diagnostic log when any process waits longer than `deadlock_timeout`
- the presence of deadlock is error in application architecture
- two or more processes can become mutually blocked (a " deadlock ring "



Parameters `deadlock_timeout` And `log_lock_waits`

If the process cannot obtain lock, it goes to sleep and sets a timer to wake up after the time specified by the `deadlock_timeout configuration parameter` (default 1 second). It will wake up before the timeout expires if the lock is granted to it. If the timeout expires, the deadlock detection routine will be called. If there is no deadlock, the process will go to sleep again and will no longer check for a deadlock in this wait loop.

The process checking for deadlock exclusively locks (with lightweight locks) access to all partitions of the lock table until the check is complete, so that all processes in the instance that want to acquire even a shared lock (for example, to perform a SELECT) will wait until the check is completed :

```
CheckDeadLock (void)
{
 int i ;
/* Acquire exclusive lock on the entire shared lock data structures. Must grab LWLocks in partition-number order to avoid LWLock deadlock.
 Note that the deadlock check interrupt had better not be enabled anywhere that this process itself holds lock partition locks, else this will wait forever. Also note that LWLockAcquire creates a critical section, so that this routine cannot be interrupted by cancel/die interrupts . */
for ( i = 0; i < NUM_LOCK_PARTITIONS; i ++)
 LWLockAcquire ( LockHashPartitionLockByIndex ( i ), LW_EXCLUSIVE );
```

Deadlock is always a bug in application architecture. Most applications are designed correctly and do not allow deadlocks. What value should I set the parameter to ?

It is worth setting `log_lock_waits = true` and adjusting the `deadlock_timeout value` so that messages about waiting for a lock to be acquired are infrequent. Ideally, the value should be greater than the typical transaction time to increase the chances that the lock will be released before the waiting transaction decides to run a deadlock check. It is important to remember that the wait will be long if some process requests an AccessExclusive lock on an object and waits to receive it.
https://www.postgresql.org/message-id/flat/4530a101c4d17174582b07875ead600d%40oss.nttdata.com

Parameter `lock_timeout`

- by default it is not valid (value is zero)
- should not be installed at cluster level
- affects both explicit and implicit attempts to obtain a lock

```
set lock_timeout = '1s';
ALTER TABLE test ADD COLUMN ts timestamp DEFAULT clock_timestamp ();
ERROR: canceling statement due to lock timeout
```

- `granted= f` means that the lock was not obtained (ShareLock another process prevents it from being received):

```
 select pid , relation:: regclass ::text, mode, granted, fastpath from
pg_locks where locktype ='relation' and database = (select oid from
pg_database where datname = current_database ()) order by pid , fastpath
desc ;
   pid | relation | mode | granted | fastpath
-------+----------+---------------------+---------+----------
 33210 | test | AccessExclusive Lock | f | f
 43344 | test | ShareLock | t | f
```


Parameter `lock_timeout`

If you are running a command that might encounter a long wait to acquire a lock, you can use the NOWAIT or SKIP LOCKED option of the command if it has one. The disadvantage of NOWAIT is that even if there is a fraction of a second to wait, the command will return an error.

In addition to these options, you can set the `lock_timeout parameter` . The advantage is that it affects any commands and what can be set to a reasonable time for a specific moment to wait for a lock. Setting it at the cluster level is not recommended, since it affects all commands, which may be undesirable.

It is recommended to set this at session level before executing commands that require an exclusive lock, since while such a command is waiting for the lock to be acquired, no other process (even one executing a SELECT) will be able to acquire the lock and will wait until the command acquires the lock, executes, and releases the exclusive lock. Often the duration of the wait for an exclusive lock to be acquired is longer than the duration of the command execution.

When executing commands that modify objects, it is worth familiarizing yourself with what type of lock on the object the command uses ; whether locks are set on other objects ; whether data blocks are changed and whether an exclusive lock is required to define a buffer (lightweight lock BufMappingLock) for each data block.

For example, adding a column to a table with a default value returned by a function rather than a constant not only sets an AccessExclusive lock on the table, but also changes all the blocks in the table. Removing a column sets the same lock, but does not change the blocks, and is executed immediately after acquiring the lock.

Creating indexes with the CONCURRENTLY option sets a lock on the table, which allows commands that change rows in the table to work. Creating an index without the CONCURRENTLY option does not allow this. The disadvantage of this option is that the index may be created with an error - the command is not rolled back. In this case, the index must be deleted with the REINDEX INDEX CONCURRENTLY command. You can also delete indexes with a lower lock level with the DROP INDEX CONCURRENTLY command .

Subtransactions

- subtransactions are save points
  - are used to roll back rather than to put the transaction into a failed state
- are being created
  - SAVEPOINT team
  - EXCEPTION section in pl/ pgsql block
- The PGPROC structure stores up to 64 subtransaction numbers
- subtransactions that only read data are assigned a virtual number
- if a data modification command is encountered, then subtransactions up to the main transaction are assigned real numbers
- recommendation: do not write frequently executed code that generates more than 64 subtransactions , its execution will become a bottleneck (TPS drop by ~25%)
- Tantor DBMS has a parameter `subtransaction_buffers`

Subtransactions

The PGPROC structure stores **up to 64** (`PGPROC_MAX_CACHED_SUBXIDS`) subtransactions . Subtransactions are savepoints that can be rolled back to rather than causing the transaction to fail.

Subtransactions are created :

1) SAVEPOINT command ;

2) the EXCEPTION section in a block in the pl / pgsql language (the savepoint is implicitly set at the beginning of the block with the EXCEPTION section) .

Subtransactions can be created in other subtransactions and a tree of subtransactions is formed . Subtransactions that only read data are assigned a virtual number. If a data modification command is encountered, then subtransactions up to the main transaction are assigned real numbers. The xid of a child subtransaction is always lower than that of the parent.

Each server process's PGPROC structure caches up to 64 subtransaction numbers . If the number of subtransactions is greater, the overhead of supporting work with subtransactions increases significantly.

Recommendation: do not write frequently executed code that generates more than 64 subtransactions , its execution will become a bottleneck (**TPS drop ~ 25%**) .

`PGPROC_MAX_CACHED_SUBXIDS` is a constant (macro) because compilation with constants produces more efficient and compact code than with variable parameters.

Tantor DBMS has a configuration parameter `subtransaction_buffers` which you can configure the size of the buffer that stores subtransaction data . The default value is 256Kb. The buffer size can be viewed:

```
SELECT name, allocated_size , pg_size_pretty ( allocated_size ) FROM
pg_shmem_allocations where name like '% btrans %';
 name | allocated_size | pg_size_pretty
------------------+----------------+----------------
  subtransaction | 267520 | 261 kB
```

Increasing the buffer solves problem #4 described in:

https://postgres.ai/blog/20210831-postgresql-subtransactions-considered-harmful

https://gitlab.com/postgres-ai/postgresql-consulting/tests-and-benchmarks/-/issues/20

https://www.cybertec-postgresql.com/en/subtransactions-and-performance-in-postgresql/

Multitransactions

- A FOR NO KEY UPDATE lock is set by an UPDATE command that makes no changes to the key columns.
- FOR KEY SHARE lock is set by DELETE and UPDATE commands that update the values of key columns
- also shared locks are set by the commands SELECT .. FOR SHARE, FOR NO KEY UPDATE, FOR KEY SHARE
- Shared locks allow multiple transactions to work on a row at the same time
- Simultaneous work is implemented by " multi-transactions " , which have their own `xid` counter , their own files and caches
- the correspondence between `xid` transactions and multitransactions is stored in the directory `PGDATA/ pg_multixact`
- advisory locks are not a replacement for row locks, as their number is limited

Multitransactions

The SELECT .. FOR SHARE, FOR NO KEY UPDATE, FOR KEY SHARE commands allow several transactions to work with a row simultaneously. The FOR NO KEY UPDATE lock is set by the UPDATE command , which does not make changes to the key columns. The FOR KEY SHARE lock is set by the DELETE and UPDATE commands , which update the values of the key columns. More detailed wording is in the documentation. The important thing is that regular DELETE and UPDATE commands can set shared locks on rows. When a second transaction appears while the first one is running, the second server process will create a multitransaction . Most applications that mainly create rows do not experience problems, since the inserted row is not visible to other sessions and they cannot lock it. A conflict can occur when inserting a record into a unique index, and then the second transaction will wait (there will be no multitransactions). And this is unlikely, since properly designed applications use auto-incrementing columns. Updating rows is a labor-intensive operation in all relational DBMSs, and especially in PostgreSQL because PostgreSQL stores old versions of rows in data blocks. If the architect (designer) of the application actively uses UPDATE , then in addition to reducing the share of HOT cleanup , it is possible that some transactions will " collide " on some rows and the second server process will create a multitransaction . Subsequent transactions can join to a multitransaction , that is, there can be two or more transactions. Moreover, a new multitransaction is created , where the previous transactions are included. This is not optimal , but the probability that not two, but three or more transactions will want to change the row is usually not high.

If deadlocks occur in an application, this directly indicates errors in the application architecture. If, instead of changing the logic of working with data, shared locks are used, then deadlocks may stop, but performance will not improve.

In `pg_stat_activity wait_event IN(' MultiXactMemberControlLock ', ' MultiXactOffsetControlLock ', ' multixact_member ', ' multixact_offset ')` , they **are of type `wait_event_type = LWLock` .**

LWLock (lightweight locks) is a type of lock (algorithm) that is used along with spinlock to arbitrate access to shared memory structures . **Their total number in an instance is limited by a constant: `MAX_SIMUL_LWLOCKS = 200` .**

https://gitlab.com/postgres-ai/postgresql-consulting/tests-and-benchmarks/-/issues/24

An auxiliary function for outputting information bits in a convenient form:

```
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 ;
select * from heap_page ('t',0);
```

A multitransaction can also be generated by the following commands:

```
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 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));
 lp  | lp_off | lp_len | t_ctid | t_xmin | t_xmax | t_ctid | t_infomask | m
----+--------+--------+--------+--------+--------+--------+------------+---
1 | 8144 | 28 | (0,2) | 46891 | 11 | (0,2) | 4416 | t
2 | 8112 | 28 | (0,2) | 46893 | 46892 | (0,2) | 8336 | f
select * from heap_page ('t',0);
 ctid | state | xmin | xmax | hhu | hot | t_ctid | multi
------+--------+--------+-------+-----+-----+--------+-------
(0,1) | normal | 46891 c | 11 | t | | (0,2) | t
(0,2) | normal | 46893 | 46892 | | t | (0,2) | f
```

This is one of the reasons to avoid using savepoints and analyze subroutine blocks in pl/ pgsql with EXCEPTION sections .

HOT chain was created because there are no indexes on the table.

hhu - a 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 .

When accessing a line:

```
select * from t;
```

the server process will access the multitransaction status and update the bits in the mask to indicate that the transaction has been committed:

```
select * from heap_page ('t',0);
 ctid | state  | xmin    | xmax  | hhu | hot | t_ctid | multi
------+--------+---------+-------+-----+-----+--------+-------
(0,1) | normal | 46891 c |   11  |   t |     | (0,2)  | t
(0,2) | normal | 46893 c | 46892 |     |   t | (0,2)  | f
(2 rows)
```

After vacuum version of the string and traces of multitransaction will be cleared :

```
vacuum t;
 select * from heap_page ('t',0);
 ctid | state         | xmin   | xmax   | hhu | hot | t_ctid | multi
------+---------------+--------+--------+-----+-----+--------+-------
(0,1) | redirect to 2|        |        |     |     |        | f
(0,2) | normal        |46,893 c|46 892 a|     |   t |  (0,2) | f
```

If the horizon were held it would be like this:

```
 ctid | state  |   xmin |  xmax | hhu | hot | t_ctid | multi
------+--------+--------+-------+-----+-----+--------+-------
(0,1) | normal |     2c | 46893 |   t |     | (0,2)  | f
(0,2) | normal |46893 c |   0 a |     |   t | (0,2)  | f
```

Hint bits `c a` are a sign of line version freezing.

There are parameters:

```
\ dconfig multixact *
List of configuration parameters
Parameter                | Value
-------------------------+-------
 multixact_member_buffers| 256kB
 multixact_offset_buffers| 128kB
```

which solve a particular problem: when multitransactions appear , processes must check the status of transactions and for this purpose access multixact structures . The effect of changing the size is ~ 4% .

The number of object locks and advisory locks is limited on an instance by the product of `max_locks_per_transaction * ( max_connections + max_prepared_transactions )` . Increasing the values of these parameters results in an increase in the shared memory structures, while the number of partitions (on which **the LWlock is set) is not changed by these parameters** . Extensive use of advisory locks as a replacement for row locks can lead to exhaustion of the total number of locks on an instance.

https://www.postgresql.org/message-id/flat/2BEC2B3F-9B61-4C1D-9FB5-5FAB0F05EF86@yandex-team.ru

Fast way to block

- designed to reduce the overhead of acquiring and releasing locks on relations (table sections, regular tables, indexes) that are frequently accessed but rarely conflict
- used for " weak " locks: AccessShare , RowSharek , RowExclusive which set the commands SELECT, INSERT, UPDATE, DELETE, MERGE
- the presence of a " strong " (Share, ShareRowExclusive , Exclusive, AccessExclusive) locks on a relation that fall in the 1/1024 portion of the lock table prevent the use of the fast path, so:
  - If you place a strong lock on a table, commands that operate on it will not be able to use the fast path.
  - Avoid executing large numbers of commands that set heavy locks during heavy instance load.

Fast way to block

is designed to reduce the overhead of acquiring and releasing locks that are frequently requested but rarely conflict with. These are locks on related objects of the relations type . The SELECT, INSERT, UPDATE, DELETE, MERGE commands must lock all relations (tables, indexes, table sections) during the plan generation and leave locks on those relations that are used in the execution plan. Many of these commands can be executed simultaneously on one table and they usually do not conflict.

A shared lock table, even if partitioned, would be a bottleneck. Even on two processor cores, processes collide when trying to set a lock on a table partition. As the number of cores, or worse, processors, increases, waits would become a bottleneck. To eliminate the bottleneck, a PROC structure is used. It allocates space for each process to store locks for a maximum of 16 relations .

How can I make sure there are no conflicting locks on a relation without accessing the lock table (otherwise I'll have to get a shared LWLock) to the lock table section , which is not very fast) and looking at only the PGPROC structures? Locks on the same relation always end up in the same section. **The hash for relations is calculated based on dboid only and reloid** . A shared memory array of 1024 parts (sections) with integers (FAST_PATH_STRONG_LOCK_HASH_PARTITIONS) is used . Each number reflects the number of " strong " locks (`Share`, `ShareRowExclusive` , `Exclusive`, `AccessExclusive`) , which can conflict with those that can be installed via the fast path in 1/1024 of the lock table. Calculating which object is in which part is done quickly - by hash . If the number of strong locks on a part is zero, then the fast path is used.

The lock table is divided into `NUM_LOCK_PARTITIONS = 16` partitions .

If the number of strong locks on a partition is greater than zero, then the fast path is not used. **If you set a strong lock on a table, then commands working with it will not be able to use the fast path.** The same thing happens if you set a lock on an object whose hash falls into the same 1 /1024 part of the lock table. Rough estimate of the probability: if the number of strong locks on objects is 10, the fast path will not be able to be used in ~ 1%. For processes that **cannot** use fastpath Only 16 tranches are available for the total lock table of the entire cluster . That is, a maximum of **16 processes can simultaneously create or remove locks on tables and indexes via the normal path** .

Strong and weak table locks

- weak locks can be obtained via the fast path
- strong table locks prevent weak locks from being set on the fast path
- autovacuum and autoanalysis do not interfere with the use of the fast path
- autovacuum and autoanalysis do not block commands, in case of a lock conflict, the autovacuum workflow interrupts table processing

	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCL .				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Strong and weak table locks

Weak locks can be obtained via the fast path:

AccessShare - sets SELECT , COPY TO , ALTER TABLE ADD FOREIGN KEY (PARENT) and any query that reads the table. Conflicts only with AccessExclusive .

RowShare - sets SELECT FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE, FOR KEY SHARE. Conflicts with Exclusive and AccessExclusive .

RowExclusive - sets INSERT, UPDATE, DELETE, MERGE, COPY FROM . Conflicts with Share, ShareRowExclusive , Exclusive, AccessExclusive .

Not weak and not strong blocking:

ShareUpdateExclusive - **sets autovacuum, autoanalysis** and commands VACUUM (without FULL), ANALYZE , CREATE INDEX CONCURRENTLY , DROP INDEX CONCURRENTLY, CREATE STATISTICS, COMMENT ON, REINDEX CONCURRENTLY , ALTER INDEX (RENAME) , 11 types of ALTER TABLE

Autovacuum and autoanalysis do not interfere with the use of the fast route.

Strong lock tables, if present, do not allow weak locks to be set on the fast path. Their list:

Share - CREATE INDEX (without CONCURRENTLY)

ShareRowExclusive - sets CREATE TRIGGER and some types of ALTER TABLE

Exclusive - installs REFRESH MATERIALIZED VIEW CONCURRENTLY

AccessExclusive - sets DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL and REFRESH MATERIALIZED VIEW (without CONCURRENTLY) , ALTER INDEX, 21 types of ALTER TABLE .

Autovacuum does not interfere with the execution of commands by server processes . If autovacuum or autoanalysis processes a table and the server process requests a lock that is incompatible with the lock that autovacuum has set (ShareUpdateExclusive), the autovacuum worker process is terminated by the server process via `deadlock_timeout` and the following message is written to the diagnostic log:

```
ERROR:  canceling autovacuum task
DETAILS: automatic vacuum of table ' name '
```

Autovacuum will try to process the table and its indexes again in the next cycle. Such errors should not be constant for the same table. If Autovacuum cannot process the table for a long time, this will lead to the bloating of its data files.

https://github.com/postgres/postgres/blob/master/src/backend/storage/lmgr/README

Directory command- set locks

ShareUpdateExclusive install commands:

```
VACUUM
REINDEX CONCURRENTLY
DROP INDEX CONCURRENTLY
CREATE STATISTICS
CREATE INDEX CONCURRENTLY
COMMENT ON
ANALYZE
ALTER TABLE VALIDATE CONSTRAINT
ALTER TABLE SET WITHOUT CLUSTER
ALTER TABLE SET TOAST
ALTER TABLE SET STATISTICS
ALTER TABLE SET N_DISTINCT
ALTER TABLE SET FILLFACTOR
ALTER TABLE SET AUTOVACUUM
ALTER TABLE DETACH PARTITION CONCURRENTLY (PARENT)
ALTER TABLE CLUSTER ON
ALTER TABLE ATTACH PARTITION (PARENT)
ALTER INDEX (RENAME)
```

AccessExclusive is installed by the following commands:

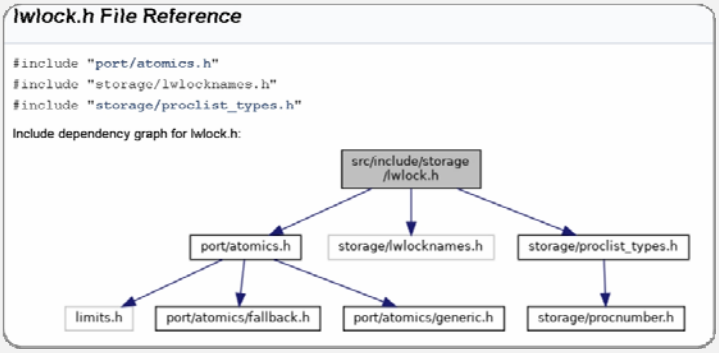
```
VACUUM FULL
TRUNCATE
REINDEX
REFRESH MATERIALIZED VIEW
DROP TABLE
DROP INDEX
CLUSTER
ALTER TABLE SET/DROP DEFAULT
ALTER TABLE SET TABLESPACE
ALTER TABLE SET STORAGE
ALTER TABLE SET SEQUENCE
ALTER TABLE SET DATA TYPE
ALTER TABLE SET COMPRESSION
ALTER TABLE RESET STORAGE
ALTER TABLE RENAME
ALTER TABLE INHERIT PARENT
ALTER TABLE ENABLE/DISABLE RULE
ALTER TABLE ENABLE/DISABLE ROW LEVEL SECURITY
ALTER TABLE DROP EXPRESSION
ALTER TABLE DROP CONSTRAINT
ALTER TABLE DROP COLUMN
ALTER TABLE DETACH PARTITION (PARENT)
ALTER TABLE DETACH PARTITION (TARGET/DEFAULT)
ALTER TABLE DETACH PARTITION CONCURRENTLY (TARGET/DEFAULT)
ALTER TABLE ATTACH PARTITION (TARGET/DEFAULT)
ALTER TABLE ALTER CONSTRAINT
ALTER TABLE ADD COLUMN
ALTER TABLE ADD CONSTRAINT
ALTER TABLE SET (LOGGED | UNLOGGED)
ALTER INDEX SET TABLESPACE
ALTER INDEX SET FILLFACTOR
ALTER INDEX ATTACH PARTITION
```

https://pglocks.org/

`The ALTER TABLE SET (LOGGED | UNLOGGED)` command completely rebuilds the table and indexes, taking exclusive locks on them and on dependent objects (sequences).

Sections of the lock table

- The lock table is divided into 16 sections (partitions)
- 's PGPROC maintains a list of parts it locks.
- The number of lightweight locks, which are used to protect memory structures such as the lock partition , is limited to 200 (`MAX_SIMUL_LWLOCKS`)



Sections of the lock table

The lock table is divided into 16 parts (2 ^ 4). The power of two is set by the macro :

```
/* Number of partitions the shared lock tables are divided into */
#define LOG2_NUM_LOCK_PARTITIONS 4
#define NUM_LOCK_PARTITIONS (1 << LOG2_NUM_LOCK_PARTITIONS)
```

's PGPROC maintains a list of the parts it locks:

```
/* All PROCLOCK objects for locks held or awaited by this backend are linked
into one of these lists, according to the partition number of their lock. */
dlist_head myProcLocks [ NUM_LOCK_PARTITIONS ] ;
```

In PostgreSQL and Tantor DBMS `LOG2_NUM_LOCK_PARTITIONS is` a constant . **In forks with the `log2_num_lock_partitions configuration parameter` , increasing its value increases the PGPROC structure .**

The total number of lightweight locks that are used to protect memory structures such as the lock partition is limited by the macro `MAX_SIMUL_LWLOCKS=200` .

Access to PGPROC structures is very frequent , as is PROC_HDR , BufferDesc . If " hot " structures or their parts no longer fit into caches processors, the performance will start to decrease. Because of this, in forks where **`log2_num_lock_partitions is used`** forced to change the logic of LWLocks making them heavier (`the lwlock_shared_limit` parameter) .

Compiling with constants produces more efficient and compact code than with mutable parameters like **`log2_num_lock_partitions`** : "As for turning the parameter into a GUC, that has a cost too. Either direct - a compiler can do far more optimizations with compile-time constants than with values that may change during execution, for example. Or indirect - if we can't give users any guidance how/when to tune the GUC, it can easily lead to misconfiguration (I can't even count how many times I had to deal with systems where the values were "tuned" following the logic that more is always better)." (Tomas Vondra , EnterpriseDB)

https://www.postgresql.org/message-id/flat/779f2bd6-00f3-4aac-a792-b81f47e41abd%40enterprisedb.com

Tranches of blockings

- number of lightweight locks 200 is the general limit, the number of locks for each LWLock subtype (of which there are more than 73) has a smaller limitation and is called a tranche
- to work with the heavy-weight lock table, 16 tranches can be used simultaneously (NUM_LOCK_PARTITIONS = 16)
  - in the normal way only 16 processes can be in the process of acquiring a lock on an object at a time
- to work with the buffer descriptor of 128 tranches (`NUM_BUFFER_PARTITIONS =128`)
- lightweight lock names - `wait_event column value` for `wait_event_type = ' LWLock '` `pg_stat_activity` views

Tranches of blockings

Maximum number of LWLocks of all types is limited to 200. At the same time, for each subtype of LWLocks There is a smaller limitation that limits the number of subtype locks.

Tranches are allocated by the LWLockInitialize (..) function. Three subtypes of LWLocks that most often become bottlenecks are: BufferMapping , LockManager , ProcArray .

For BufMappingLock 128 tranches, that is, 128 tranches are used for locks on access to buffer cache buffer descriptors :

```
 /* Initialize buffer mapping LWLocks in main array */
lock = MainLWLockArray + BUFFER_MAPPING _LWLOCK_OFFSET;
for (id = 0; id < NUM_BUFFER_PARTITIONS; id++, lock++)
  LWLockInitialize (&lock->lock, LW TRANCHE _BUFFER_MAPPING);
```

For the lock table, 16 tranches are initialized:

```
 /* Initialize lmgrs ' LWLocks in main array */
lock = MainLWLockArray + LOCK_MANAGER_LWLOCK_OFFSET ;
for (id = 0; id < NUM_LOCK_PARTITIONS; id++, lock++)
  LWLockInitialize (&lock->lock, LW TRANCHE _LOCK_MANAGER);
```

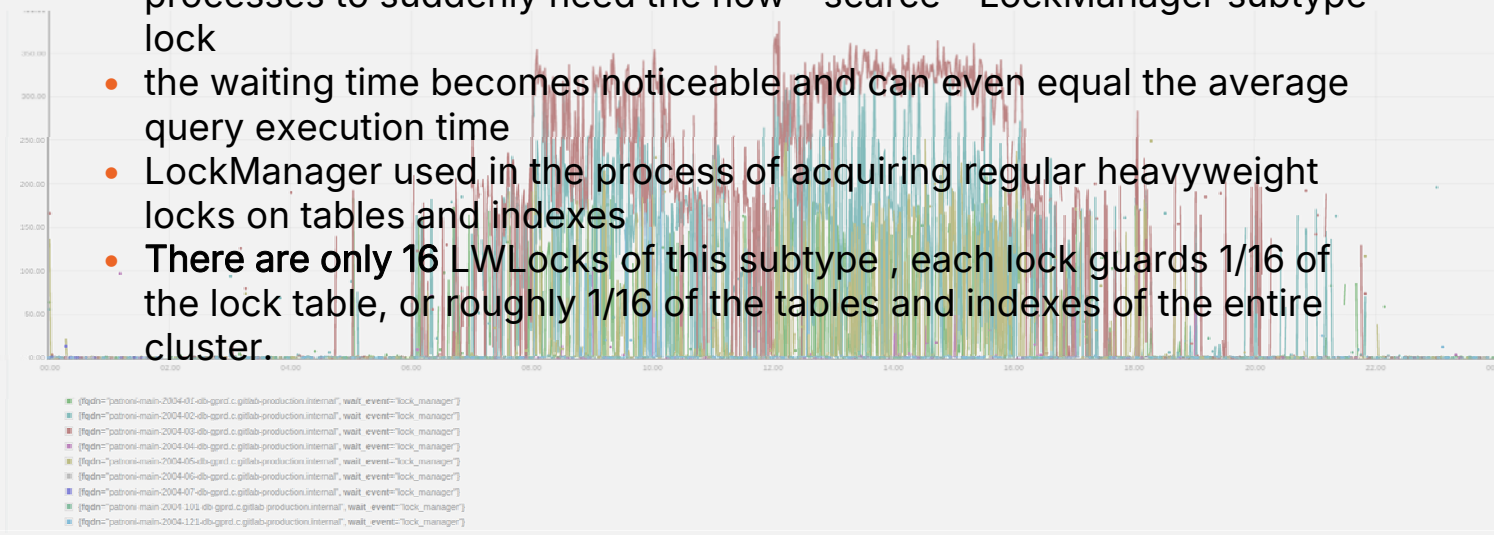
This means that in the normal way only 16 processes can be in the process of acquiring a lock on an object at the same time. From the point of view of monitoring, i.e. finding out what subtype of lightweight lock is expected by processes, and therefore was the bottleneck, you can use the `pg_stat_activity view` . The names of lightweight locks are the value of the wait_event column for wait_event_type = ' LWLock ' of the pg_stat_activity table :

https://docs.tantorlabs.ru/tdb/ru/16_4/se/monitoring-stats.html#WAIT-EVENT-LWLOCK-TABLE

The names of lightweight locks that use tranches are in the BuiltinTrancheNames [] and IndividualLWLockNames [] arrays of lwlocknames.c Extensions can create and register tranches for their locks.

Lightweight locks

- LWLocks microseconds must be maintained
- a delay of hundreds or thousands of microseconds is enough for other processes to suddenly need the now " scarce " LockManager subtype lock
- the waiting time becomes noticeable and can even equal the average query execution time
- LockManager used in the process of acquiring regular heavyweight locks on tables and indexes
- **There are only 16** LWLocks of this subtype , each lock guards 1/16 of the lock table, or roughly 1/16 of the tables and indexes of the entire cluster.



Lightweight locks

Usually, LWLocks are removed quickly - **within microseconds** , but at times of high load they can last a long time and processes waiting to receive a lightweight lock on 1 /16 of the lock table can wait relatively long, the problem snowballs. Tranche (subtype) LockManager used in the process of acquiring regular heavyweight locks on tables and indexes. For example, just to create an execution plan. Each query must acquire these locks. **There are only 16** LWLocks of this subtype , each lock guards 1/16 of the lock table, or roughly 1/16 of the tables and indexes of the entire cluster.

While a server process holds a LockManager lock , if it slows down for any reason (such as being preempted by the operating system scheduler), it can quickly escalate to blocking other processes. Application server connection pools will become overwhelmed by the increasing duration of requests generated by application code. Delays in acquiring LWLocks increase with non-uniform access to shared memory structures: if multiple processors are used.

If the LockManager holder lags for a few microseconds, that's fine. But a lag of hundreds or thousands of microseconds is enough to cause other processes to suddenly need the now " scarce " LockManager subtype lock . Since this subtype is used in exclusive mode (the record in the lock structure is changed, not read), all processes wait until the current lock holder releases it. This results in a " bottleneck " . And the waiting time becomes noticeable quite quickly and can even equal the average query execution time.

Gitlab , which serves a large number of requests (lock_manager) encountered this problem former name LockManager):

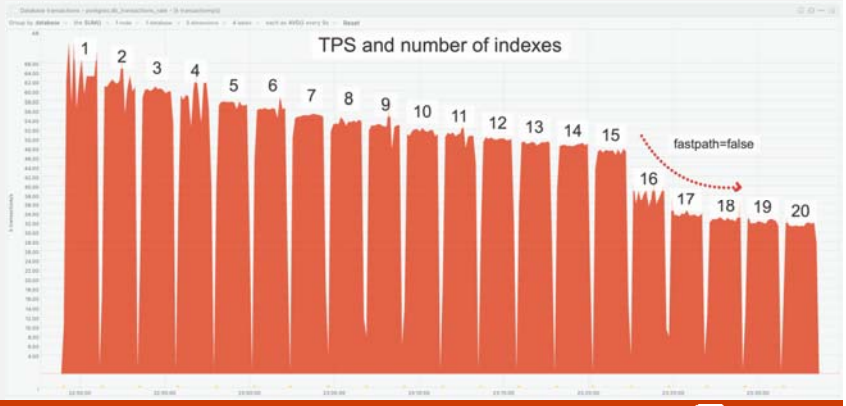
https://gitlab.com/gitlab-com/gl-infra/scalability/-/issues/2301

Contention most often occurs for LockManager , BufferMapping , ProcArrayLock locks . Bottlenecks are gradually being eliminated in new versions of PostgreSQL . The probability of bottlenecks occurring occurs when memory structures and processor cores are increased more than usual. In such cases, the bottleneck can be eliminated by: limiting instance processes to one processor ; disabling Hyper Threading (the number of " cores " is reduced by 2 times) ; reducing shared_buffers from hundreds of gigabytes to tens gigabyte ; max_connections from thousands to hundreds . Example of problems on older versions PostgreSQL:

https://pgconf.ru/media//2020/02/06/Korotkov_pgconfru_2020_Bottlenecks_2.pdf

Fast Path Blocking and 16 locks

- the first 16 locks of the type AccessShareLock , RowShareLock , RowExclusiveLock the process gets the fast path
- for the rest a slower method is used
- with a large number of active processes and a large number of `fastpath=false` locks `wait_event='LockManager ' ` in `pg_stat_activity` go to the top and become a bottleneck



Fast Path Blocking and 16 Blocks

Macro in proc.h FP_LOCK_SLOTS_PER_BACKEND sets that **16 locks of type AccessShare , RowShare , RowExclusive will be obtained via fast path** , which reduces contention for access to memory structures and reduces wait event (pg_stat_activity) of type LockManager . Only for 16 locks of each process on objects of type relation in column `pg_locks.fastpath= true` . For others `fastpath= false` .

When fastpath=false , the lock manager uses a slower method to acquire locks. When there are a large number of active processes and a large number of fastpath=false locks, `wait_event='LockManager ' ` in `pg_stat_activity` come to the top and become a bottleneck. Moreover, blockages are taken both at the plan creation stage and at the execution stage, including for SELECT , which means that the problem will also manifest itself on replicas. Performance degradation manifests itself under high load.

This problem was discovered when replicas were consolidated: the number of replicas was reduced and queries were moved to one. The problem can occur with a large number of sections with queries that do not allow excluding sections from scanning (partition pruning) or indices.

Increasing FP_LOCK_SLOTS_PER_BACKEND from 16 to 64 will increase the size of the PGPROC structure in shared memory. PGPROC stores the process state. It occupies 880 bytes, which is equal to 14 cache lines (data blocks) , adding 48 xid will increase it by 192 bytes (3 cache lines) . " Cache lines " are 64-byte blocks that transfer data between the processor cache and memory, containing a copy of the data from main memory.

In version 16, the way to obtain a lock in the usual way was optimized. Every time a SELECT is executed or a transaction is started (getting a snapshot) , a call to PGPROC is made to compile a list of active transactions. A call is also made when checking for deadlocks. The number of **PGPROC structures is specified by the max_connections parameter .** Increasing either reduces productivity .

The performance impact begins to be felt when there are a large number of active requests and more than one processor.

https://ardentperf.com/2024/03/03/postgres-indexes-partitioning-and-lwlocklockmanager-scalability/

https://gitlab.com/postgres-ai/postgresql-consulting/tests-and-benchmarks/-/issues/41

Let's use standard pgbench tables. Let's create a table with 8 sections and execute a query on it that has no predicates that allow excluding sections from scanning: the query does not contain a column that is part of the partition key, the bid column is used . Example:

```
pgbench -i -s 1 --partitions=8
```

first session:

```
postgres=# begin transaction;
postgres=*# explain (analyze, costs false) select abalance from pgbench_accounts where bid=1 limit 1;
 QUERY PLAN
 ------------------------------------------------------------
 Limit (actual time=0.030..0.054 rows=1 loops=1)
 -> Append (actual time=0.022..0.034 rows=1 loops=1)
 -> Seq Scan on pgbench_accounts_1 (actual time=0.013..0.017 rows=1 loops=1)
Filter: (bid = 1)
 -> Seq Scan on pgbench_accounts_2 (never executed)
 -> Seq Scan on pgbench_accounts_3 (never executed)
 -> Seq Scan on pgbench_accounts_4 (never executed)
 -> Seq Scan on pgbench_accounts_5 (never executed)
 -> Seq Scan on pgbench_accounts_6 (never executed)
 -> Seq Scan on pgbench_accounts_7 (never executed)
 -> Seq Scan on pgbench_accounts_8 (never executed)
 Planning Time: 0.170 ms
 Execution Time: 0.095 ms
```

second session:

```
postgres=# select pid , relation:: regclass ::text, mode, granted, fastpath from pg_locks where locktype ='relation' and database = (select oid from pg_database where datname = current_database ()) order by pid , fastpath desc ;
   pid | relation | mode | granted | fastpath
 -------+----------------------+-----------------+---------+----------
 93229 | pg_locks | AccessShareLock | t | t
 94341 | pgbench_accounts_7_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts_7 | AccessShareLock | t | t
 94341 | pgbench_accounts_6_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts_6 | AccessShareLock | t | t
 94341 | pgbench_accounts_5_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts_5 | AccessShareLock | t | t
 94341 | pgbench_accounts_4_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts_4 | AccessShareLock | t | t
 94341 | pgbench_accounts_3_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts_3 | AccessShareLock | t | t
 94341 | pgbench_accounts_2_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts_2 | AccessShareLock | t | t
 94341 | pgbench_accounts_1_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts_1 | AccessShareLock | t | t
 94341 | pgbench_accounts_pkey | AccessShareLock | t | t
 94341 | pgbench_accounts | AccessShareLock | t | t
 94341 | pgbench_accounts_8 | AccessShareLock | t | f
 94341 | pgbench_accounts_8_pkey | AccessShareLock | t | f
 (19 rows)
```

7 partitions and indexes are locked using fastpath (fastpath = t).

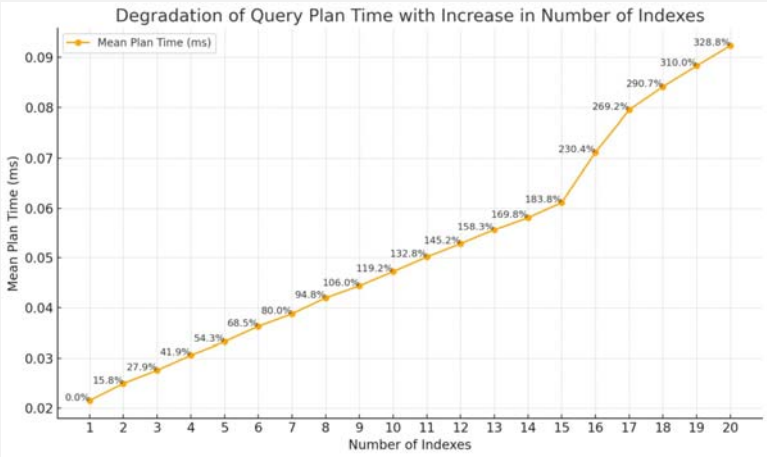
The eighth section and the index on it are locked in the usual way (fastpath = f)

Indexes, Joins, Sections, and the Fast Path

- AccessShareLock type locks are set on all tables, indexes, and table sections that are considered by the planner
- after creating a blocking plan, those relations that **are not in the plan** are removed

Recommendations:

- use prepared statements
- use fewer indexes
- use fewer connections
- use fewer sections



Indexes, Joins, Sections, and the Fast Path

AccessShareLock type locks are set at the planning stage - on all tables, indexes, table sections considered by the planner. After the plan is created, locks are removed from those relations that **are not in the plan** . The first 15 locks are obtained quickly and do not depend on the load on the instance, and subsequent locks are obtained longer, which increases the planning time. The slowdown depends on the level of competition between processes for LockManager resources (the number of sessions requesting locks) .

The increase in planning time can be noticeable for queries where the planning time is longer than the execution time, and the query is executed frequently. Unprepared queries are not cached and are re-planned each time.

Recommendations for the planning phase:

1. Using prepared statements (in JDBC Prepared Statements, in libpq advanced mode) allows you to use generic plan and eliminate the contention problem, but only at the planning phase .

2 . Use as few indexes as possible in tables that are used in frequently executed queries. If there are more than 15 indexes on a table (when executing a query, fast_path =false) it is worth changing the code so that this query is not executed too often (for example, more than 100 times per second). Perhaps the result can be cached .

Recommendations for the execution phase:

1. Use as **few indexes as possible** on tables that are used in frequently executed queries. An additional index may prevent the use of HOT, which will lead to performance degradation .

2. Use **fewer** table joins . Each join this is a lock on the joined table and its index (if used in the join)

3. When using partitioned tables, you need to understand that a large number of sections does not increase productivity in itself, but simplifies administration and is justified only for large volumes of data. For example, you do not need to make sections daily if you can get by with weekly sections (7 times smaller).

It is worth paying attention to whether sections are effectively excluded from scanning in execution plans (partition pruning).

4. On highly loaded systems, you may want to consider increasing the number of replicas or clusters with subscriptions and redistributing requests to additional instances.

`join_collapse_limit` parameter

- when connecting more than 8 tables , the efficiency decreases
- `from_collapse_limit` (default 8) sets the maximum number of " elements " in the FROM list **up to** which the planner will combine subqueries with the outer query
- `join_collapse_limit` (default is equal to `from_collapse_limi t`) specifies the maximum number of " elements " with the JOIN clause , **up to** which the planner will completely iterate over the sequence of joins
- `geqo_threshold` (default 12) the minimum number of " elements " at which the genetic algorithm will be used to plan the query

```
drop table if exists t cascade;
create table t (id numeric, c text);
insert into t values (1, 'a');
create or replace view t1 as select * from t;
select * from t1 as ta join (select * from t) as tb on (ta.id=tb.id) join
generate_series (1, 3) as tc (id) on (tb.id=tc.id);
```


Parameter `join_collapse_limit`

The first 1 6 locks are obtained quickly and do not depend on the load on the instance and that each connection This is a lock on the table being joined and its index (if used in the join). Locks on 8 joined tables (with 8 indexes) will be taken via the fast path . It is worth designing storage schemes so that the number of joined tables is, if possible, no more than 8 . If an object is already locked, it is not locked again until the end of the transaction or the lock is removed. Therefore, the Tantor SE1C `enable_self_join_removal parameter` does not affect locking.

Parameter `join_collapse_limit` (default is `from_collapse_limi t`) sets how many " elements " (relations, subqueries, table functions) **connected by the JOIN phrase** can be in the query so that the planner completely iterates over the sequence of connections (a pair of " elements " is always connected). With small values, the planning time is reduced, **with an increase greater than 6, the planning time increases sharply** . If the number of " elements " is greater than `join_collapse_limit` , then the relations are divided into groups with the number of " elements " not greater than `join_collapse_limit` and the join order is searched within the groups, and then the sets of rows that are returned from the groups are joined. Elements joined by FULL OUTER JOIN are not included in groups and are considered as one " element " , since it does not participate in the enumeration (rearrange tables cannot be JOIN'd into other tables).

If the value is 1, the relations will be joined in the order in which they are specified in the JOIN and the planner will be able to choose only the " element " by which the hash table will be built when hashing join . The order of " elements " , specified in the query will determine the actual join order. It is possible set the parameter value to 1 and list the " elements " in the desired order.

`from_collapse_limit` (default 8) specifies the maximum number of " elements " listed in the FROM list **up to** which the planner will combine nested queries with the main query. **It is recommended that the value be less than `geqo_threshold` .**

`geqo_threshold` (default 12) the minimum number of " elements " at which the genetic algorithm will be used to plan the query. " `x FULL OUTER JOIN y` " counts as one " element " .

If the application code connects a large number of " elements " (1C), you can set the values of both parameters `* _collapse_limit` at 20 and above .

Performance pg_locks

- returns all heavyweight locks on the instance (across all databases)
- The number of heavyweight locks per instance is limited by `max_locks_per_transaction * ( max_connections + max_prepared_transactions )`
- `max_prepared_transactions` does not apply to prepared statements but to distributed transactions, the default value is zero.
- `max_locks_per_transaction` default 64
- `max_connections` default 100
- in the `locktype column pg_locks` views one of 12 types of locks is specified: **`relation`** , `extend, frozenid , page, tuple , transactionid , virtualxid , spectoken , object, userlock ,` **`advisory`** , `applytransaction`

Performance pg_locks

The size of the memory structure that stores heavyweight locks is determined by the product: `max_locks_per_transaction * ( max_connections + max_prepared_transactions )` . `max_prepared_transactions` parameter does not apply to prepared statements, but to distributed transactions, and the default value is zero.

The value of `max_locks_per_transaction` default 64.

The value of `max_connections` default 100.

The number of locks that a single transaction can use is not limited by the `max_locks_per_transaction parameter` .

The values of all three parameters are stored in the pg_control file in addition to the configuration parameter files .

Since the memory for the structure is allocated when the instance is started, changing the parameters requires restarting the instance. Changes to their values are transmitted via WAL , on physical replicas the values should be no less (better equal) than on the master.

The size of the lock structure is taken into account in the line :

```
SELECT name, allocated_size , pg_size_pretty ( allocated_size ) FROM
pg_shmem_allocations where name=' <anonymous> ';
 name | allocated_size | pg_size_pretty
------------+----------------+----------------
  <anonymous> | 5003392 | 4886 kB
(1 row)
```

In this example, the lock structure takes up ~ 1MB out of 4.8MB.

In the locktype column one of **12 lock types is specified: relation, extend, frozenid , page, tuple , transactionid , virtualxid , spectoken , object, userlock , advisory, applytransaction** .

be joined with pg_stat_activity to get data about the session holding or waiting for a lock :

```
select * from pg_locks pl left join pg_stat_activity psa ON pl.pid = psa.pid;
```

When getting the object name from pg_class It is important to remember that pg_class stores data only for local database objects :

```
select relation:: regclass ::text from pg_locks where locktype ='relation' and
database = (select oid from pg_database where datname = current_database ());
```

Parameter `track_commit_timestamp`

- the default value is false and should not be enabled, since the instance does not use this data
- in the `PGDATA/ pg_commit_ts directory` the transaction commit timestamp will be preserved
- transaction commit time data can be used by the pglogical extension for conflict resolution procedures
- timestamps can be obtained using the functions:
  - `pg_last_committed_xact ()`
  - `pg_xact_commit_timestamp ()`
  - `pg_xact_commit_timestamp_origin ()`
- the parameter value is saved in the control file
- changing the value requires restarting the instance

Parameter `track_commit_timestamp`

The default value is false. If `track_commit_timestamp = true` , then **when a transaction is committed,** in addition to saving the transaction status in the `PGDATA/ pg_xact directory` , in the `PGDATA/ pg_commit_ts directory` the transaction commit timestamp will be saved . **This does not speed up the instance and the saved information is not used. instance** . Reason for the functionality to appear: commit time data is used by the `pglogical extension` for conflict resolution procedures. If this extension or similar ones are not used, then setting `track_commit_timestamp = true` is not worth it.

Timestamps can be obtained using the functions:

```
pg_last_committed_xact ()
pg_xact_commit_timestamp ()
pg_xact_commit_timestamp_origin ()
```

For example:

```
select pg_xact_commit_timestamp ( xmin ), xmin , * from t;
select * from pg_last_committed_xact ();
```

Also the `pg_waldump utility` will show the timestamp of the transaction commit. Example:

```
rmgr : Transaction len ( rec /tot): 34/ 34, tx : 1 08, lsn : 0/0 2 1588, prev
0/0 2 1550, desc : COMMIT 202 5 - 01 - 01 1 1 : 11 : 00 . 000011 With ET
```

The parameter value is saved in the control file (`pg_control`):

```
pg_controldata | grep timestamp
track_commit_timestamp setting: off
```

Transaction commits are also slowed down by the use of cursors. `WITH HOLD` .

Practice

- Part 1. Launching an instance in a docker container
- Part 2. Shared Instance Memory
- Part 3. Local memory of the server process
- Part 4. Logging process memory to the diagnostic log
- Part 5. Deadlocks during testing
- Part 6. Multitransactions
- Part 7. Test example

Practice

You will create a container with PostgreSQL version 17 and see how the instance crashes when executing simple commands.

You will see the distribution of local and shared memory, how memory is allocated and deallocated.

10

Buffer cache

Memory structures that support the buffer cache

- **Buffer Blocks** - the buffer cache itself
  - memory is allocated according to the number of buffers * 8192 bytes plus 4096 bytes
  - `shared_buffers` configuration parameter specifies the number of buffers
- **Buffer Descriptors** - buffer descriptors (headers)
  - memory is allocated according to the number of buffers * descriptor size (64 bytes)
- Each descriptor stores:
  - block address on disk in the form of a block label (BufferTag)
    - block address contains: TBS, DB, file, fork , block number from the beginning of the first file
  - buffer address as the buffer's ordinal number in the buffer cache
- the descriptor is associated 1:1 (one to one) with the buffer
- data in 20 bytes that BufferTag occupies enough (no need to contact anywhere) to read a block from disk

Memory structures that support the buffer cache

Access to cluster data is via the buffer cache. To tune performance, it is worth getting to know its operating model in general. This can be useful for guessing where and in what cases bottlenecks may occur. Cases: unusual or extreme use of database functionality. For example: frequent creation and deletion of tables, warming up the cache.

The following are the names of the structures in the shared memory of the instance related to the buffer cache and the formulas for calculating their size in bytes. The names are given as in the `pg_shmem_allocations view` . The names of the types and macros are given to make it convenient to search for text in the PostgreSQL source code if you want to study the algorithms in detail.

Buffer Blocks - the buffer cache itself. The size of each buffer is equal to the block size. The exact size of the allocated memory: `NBuffers * ( BLKSZ=8196 ) + ( PG_IO_ALIGN_SIZE=4096 )` . `NBuffers` - the number of shared buffers is specified by the `shared_buffers configuration parameter` (default 16384, maximum 1073741823=30 bits).

Buffer Descriptors - buffer descriptors (headers). The descriptor structure is called `BufferDesc` . It is located in a separate part of memory, one descriptor for each buffer. Size: `NBuffers * ( BufferDescPadded = 64)` - descriptors are aligned by cache line , which modern processors have usually 64 bytes . These 64 bytes contain:

1) the `BufferTag structure` , which specifies the direct (self-sufficient , that is, storing everything to find the file and the block in it) address of the block on disk:

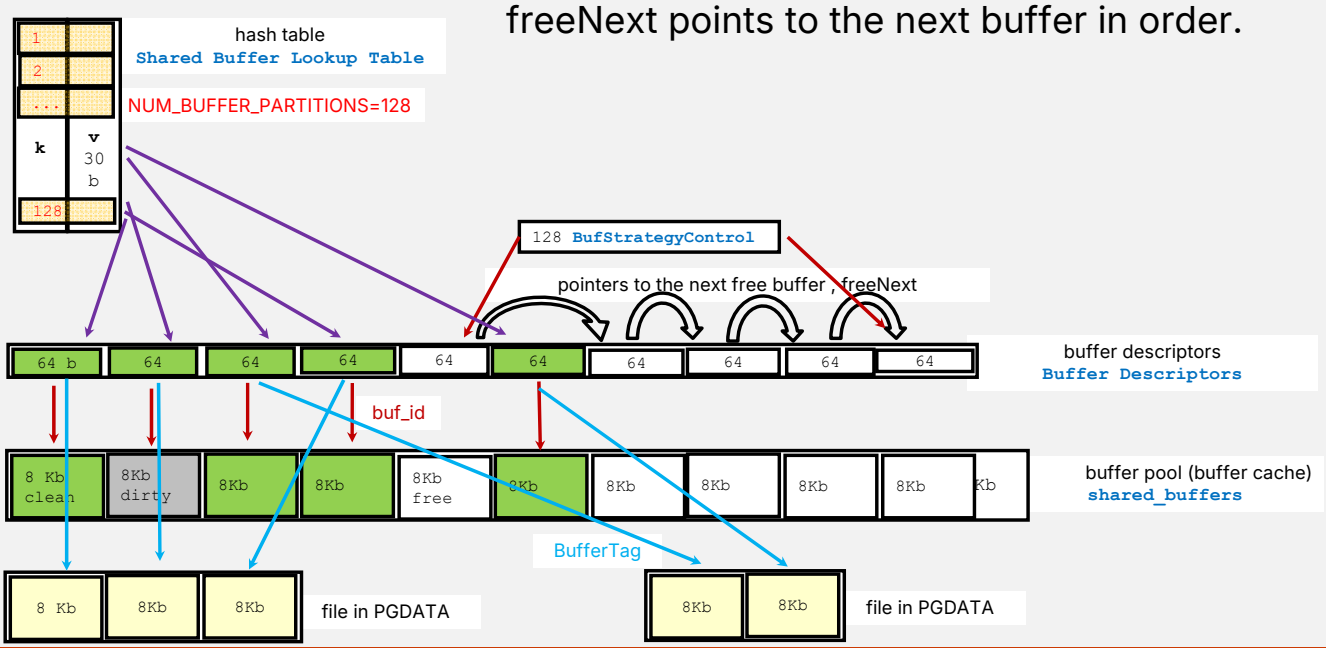
```
typedef struct buftag
{
  Oid   spcOid ;  oid tablespace ( symlink name in PGDATA/ pg_tblspc )
  Oid   dbOid ;  oid databases (subdirectory)
  RelFileNumber relNumber ;  file name, is a number
  ForkNumber   forkNum ;  fork number ( enum into 5 possible values: -1 invalid, 0 main, 1 fsm , 2 vm , 3 init)
  BlockNumber blockNum ;  block number relative to 0 block 0 of file, size 4 bytes, maximum is set by macro MaxBlockNumber
} BufferTag ;
```

`BufferTag` Size 17 bytes. Size with alignment 20 bytes.

2) `int buf_id` - the sequence number of the buffer in the buffer cache starting from zero.

Memory structures that support the buffer cache

- after instance startup, while all buffers are free freeNext points to the next buffer in order.



Memory structures serving the buffer cache (continued)

3) 32 bits, which contain : 18 bits `refcount` , 4 bits `usage count` (from 0 to BM_MAX_USAGE_COUNT =5 , 6 gradations in total) , 10 bits flags, which reflect:

1 - BM_LOCKED there is a lock on the buffer header

2 - BM_DIRTY **filthy**

3 - BM_VALID block is not damaged

4 - BM_TAG_VALID block exists in file on disk

5 - BM_IO_IN_PROGRESS buffer is in the process of filling an image from disk or writing to disk

6 - BM_IO_ERROR The previous I/O operation failed.

7 - BM_JUST_DIRTIED became dirty while writing to disk

8 - BM_PIN_COUNT_WAITER **waits for other processes to unpin to lock the buffer for modification**

9 - BM_CHECKPOINT_NEEDED **marked by the checkpoint process for writing to disk**

10 - BM_PERMANENT refers to the journaled object.

Some of these flags are used by bgwriter and checkpointer to track whether the block has changed during its writing to disk, since shared locks are set during the writing process (input/output operation) . This speeds up the DBMS.

4) `int wait_backend_pgprocno` - the identifier of the process that is waiting for other processes to remove the buffer pins (waiting for pincount 1)

If a process wants to work with a block, it looks for it in the buffer cache . If it finds it, it pins it. Multiple processes can pin a buffer. If the process does not need the buffer, the process unpins it.

Pinning prevents a block in the buffer from being replaced by another block.

A process that wants to clear space in a block from rows that have crossed the database horizon must wait until no other process is interested in the block in the buffer. except for himself, that is, the unit in pincount was set by him himself.

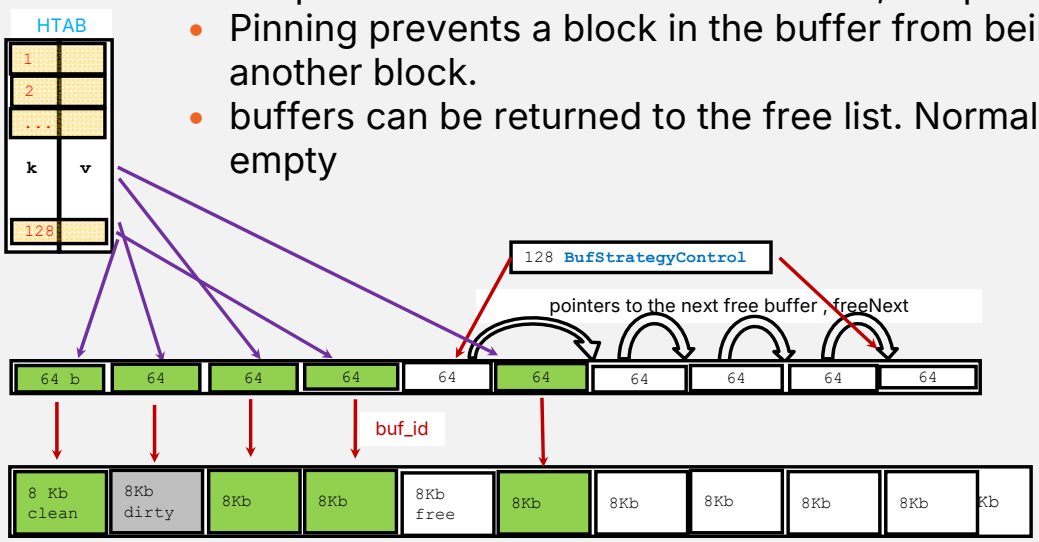
5) `int freeNext` - reference to the number of the next free block . **After the instance is started, while all buffers are free, it points to the next buffer in order** . The logic of a linked list is used ("Linked List") .

6) `LWLock content_lock` - lightweight lock on buffer contents

Lightweight buffer content_locks are set by processes for a short time. Two types: Exclusive and Shared. Exclusive prevents other processes from setting the lock , Shared can be set by several processes.

Search for a free buffer

- If a process wants to work with a block, it looks for it in the buffer cache . If it finds it, it pins it. Multiple processes can pin a buffer. If the process does not need the buffer, the process removes the pin.
- Pinning prevents a block in the buffer from being replaced by another block.
- buffers can be returned to the free list. Normally the free list is empty



Search for a free buffer

Buffer Strategy Status size: `BufferStrategyControl = 128` . Stores data for searching for free blocks:

`slock_t buffer_strategy_lock ;` Spinlock - to access this structure

`pg_atomic_uint32 nextVictimBuffer ;` counter for searching for free buffers, pointer to next free buffer is obtained by modulo division by NBuffers

`int firstFreeBuffer ;` the first unused (after instance restart) buffer, after all buffers are used, will have the value -1.

`int lastFreeBuffer ;`

`uint32 completePasses ;` used for statistics

`pg_atomic_uint32 numBufferAllocs ;` used for statistics

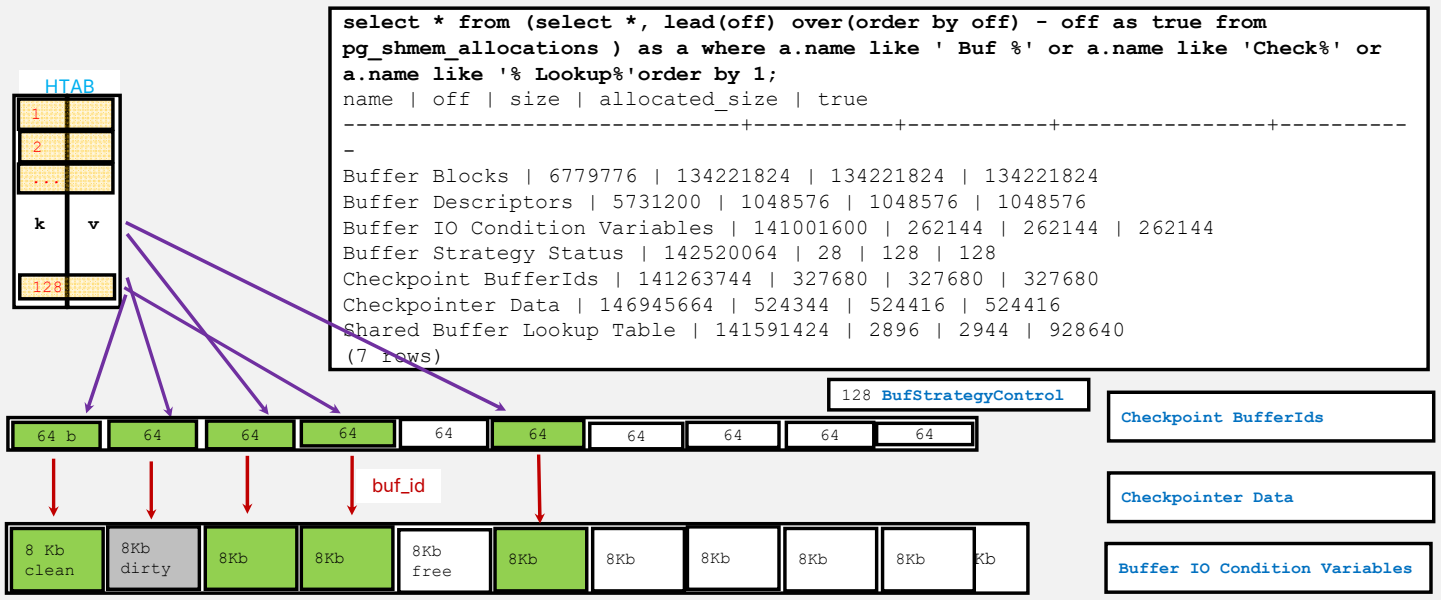
`int bgwprocno ;` bgwriter process number for notification

The word " strategy " is used in the sense of " method " from the phrase "buffer cache replacement strategy" .

Buffers can be returned to the free list using **the StrategyFreeBuffer (** BufferDesc) function. and the chain of free buffers can be supplemented and even re-created after `firstFreeBuffer` has become equal to `-1` . This happens after an object is deleted, when the buffers containing its blocks are invalidated function InvalidateBuffer (BufferDesc), which is called from the DropRelationsAllBuffers (..) function or when deleting an individual object file with the DropRelationBuffers (..) function or truncating a file (including vacuuming) with the RelationTruncate () function calling smgrtruncate (..) or when deleting a database with the DropDatabaseBuffers (dbid) function. When extending object files with the ExtendBufferedRelShared (...) function, when the extensions occurred simultaneously and another process has already loaded a block into the buffer while the first process was clearing the buffer. The cleared buffer will be returned by the first process to the free list. Or a process intends to use the buffer with the BufferAlloc (...) function to load a block, but while the process was clearing the buffer, another process has already loaded a block into another buffer. In this case, the first process will return the cleared block to the free list. Why does this happen ? Because access to structures is built on lightweight locks that are not optimally held for long and set in bulk. The process breaks a large task into subtasks and sets lightweight locks in the subtasks only where they are needed. The process releases the lock, performs other subtasks, returns to the first one and gets the lock again, and the object has changed in the meantime.

Dirty Buffer Eviction Algorithm

- the checkpointer process sorts dirty blocks separately for each file before sending them for writing



Dirty Buffer Eviction Algorithm

Checkpoint BufferIds size: `NBuffers *( CkptSortItem =20)` 320Kb if the buffer pool is 128Mb. Memory allocated with a reserve for sorting dirty buffers that will be written to disk by checkpoint. **The checkpointer process sorts dirty blocks separately for each file before sending them for writing** . If the file is on the HDD , this reduces the movement of the HDD heads .

The command to write to disk is sent via `checkpoint_flush_after` blocks (from 0 to 2MB, default 256KB) , `bgwriter_flush_after` = 512KB, `wal_writer_flush_after` =1MB

Checkpointer Data size: `CheckpointerShmemSize () = 524416`

Buffer IO Condition Variables NBuffers * (ConditionVariableMinimallyPadded = 16) flags (variables) that put the process to sleep and wake it up if the flag changes. Implement the wait method.

Shared Buffer Lookup Table , formerly known as `Buffer Mapping Table` . The third most important and second largest structure. Unlike other structures, it is a hash table. Not the most optimal (no range search), but acceptable.

Dimensions of nine structures, including `Shared Buffer Lookup (Mapping)Table` in `pg_shmem_allocations` are issued incorrectly. The sizes of structures can be estimated by the column off (offset) from the neighboring structure :

```
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;
 name | off | size | allocated_size | true
 ----------------------------+---------+------+----------------+---------
 LOCK hash | 142635392 | 4944 | 4992 | 7210752
 PREDICATELOCK hash | 159427456 | 4944 | 4992 | 5940096
 PREDICATELOCKTARGET hash | 157270016 | 2896 | 2944 | 2157440
  PredXactList | 165367552 | 88 | 128 | 890624
 Proc Header | 167902336 | 112 | 128 | 475136
 PROCLOCK hash | 149846144 | 9040 | 9088 | 7419648
  RWConflictPool | 166496256 | 24 | 128 | 1272192
 SERIALIZABLEXID hash | 166258176 | 2896 | 2944 | 238080
  Shared Buffer Lookup Table | 141706624 | 2896 | 2944 | 928640
 (9 rows)
```

Size 928640 bytes for a 128MB buffer cache.

Buffer Replacement Strategies

- number of records in BufTable this is the sum of NBuffers and NUM_BUFFER_PARTITIONS
- buffer cache replacement strategy :
  - BULKREAD. For sequential reading of table blocks (Seq Scan) whose size **is not less than 1/4 of the buffer cache** , a set of buffers in the buffer cache with a size of 256 KB is used.
  - 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` . Default is 256K.
  - BULKWRITE. Used by COPY and CREATE TABLE AS SELECT commands. Ring size is 16MB.
- in a buffer cache a block can only be in one buffer
- If a buffer becomes dirty, it is excluded from the buffer ring.

Buffer Replacement Strategies

Number of records in BufTable this is the sum of NBuffers and NUM_BUFFER_PARTITIONS due to the specifics of initialization of block descriptors. For efficiency, the BufferAlloc () function, when loading a block into a buffer that occupied another block, first inserts into BufTable record with a reference to the new block descriptor and only then releases the record in BufTable with a reference to the descriptor of the old block. To avoid a situation where in BufTable there is no space to insert a reference to a new block descriptor (which is difficult to process) additional space is immediately added to the table. Since parallel work with the contents of BufTable limited by the number of partitions , then the space for spare records is allocated in the number of partitions , the number of which is set by the NUM_BUFFER_PARTITIONS macro and is equal to 128.

Methods (**Buffer Access** Strategy **Type**) for replacing blocks in the buffer ring:

1) **BAS** _BULKREAD. For sequential reading of table blocks (Seq Scan), a set of buffers in the buffer cache of 256 KB is used. The size is chosen so that these buffers fit into the second-level cache (L2). processor cores. The ring should not be too small to accommodate all the pinned buffer process. Also, in case other processes want to scan the same data, the size should provide " gap " so that the processes synchronize and simultaneously commit, scan, and uncommit the same blocks. This method can also be used by commands that pollute buffers. Also, other processes can pollute buffers while they are in the reader's buffer ring, since **a block can only be in one buffer** . **If a buffer becomes dirty, it is excluded from the buffer ring.**

being scanned **must be larger than a quarter of the buffer cache:**

```
scan-> rs_strategy = GetAccessStrategy (BAS_BULKREAD);
```

The method is used when creating a new database using the WAL_LOG method to read the pg_class table source database. **For TOAST tables buffer rings are not used, since access to TOAST is always via the TOAST index.**

2) BAS_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, 256Kb.

3) BAS_BULKWRITE. Used by COPY, CREATE TABLE AS commands. SELECT. Ring size is 16MB. When copying (RelationCopyStorageUsingBuffer (...)) a table, **two rings are used** : one for reading the source table and one for filling the target table.

Finding a block in the buffer cache

Instance process:

- BufferTag structure in its local memory
- computes a 4-byte hash from BufferTag
- determines the partition number in **the Shared Buffer Lookup Table** hash table based on the hash value
- requests a lightweight (LWLock) lock of type BufMappingLock partitions of the table where the hash value was found
- in the hash table finds the block sequence number in the buffer cache or -1 if the block is not in the cache

Finding a block in the buffer cache

The process needs to operate on a block. Process:
1) creates an instance of the BufferTag structure in its local memory .
2) calculates the hash using the `uint32` function `newHash = BufTableHashCode ( BufferTag )`
3) determines the partition number based on the hash value `newPartitionLock = BufMappingPartitionLock ( newHash )`
4) requests a lightweight (`LWLock`) lock of type `BufMappingLock` partitions of the hash table into which the hash was placed : `LWLockAcquire ( newPartitionLock , LW_SHARED);`
3) calls the `BufTableLookup function ( BufferTag , uint32 hashcode )` which returns the sequence number of the block in the buffer cache of type int or -1 if the block is not in the cache .
Record size (hash bucket slot) in **Shared Buffer Lookup Table** 8 bytes, it consists of a hash (type uint32 , unsigned integer of size 4 bytes) and the sequence number of the buffer (its header) of type int .
The number of blocks may be greater than the number of buffers, and the hash from different blocks may coincide. In this case, records with the same key value are inserted into the table, but with references to different buffers (cache chains) .
The table is divided into NUM_BUFFER_PARTITIONS = 128 parts. One process can acquire locks on several parts, even on all parts. The lock is not held for long: the buffer header (`Buffer Descriptors`) is read by the buffer number in the record without locks, the refcount (aka ref_count , 18 bits) and usage_count are increased by an atomic operation (`pg_atomic_read_u32(& buf ->state)`) . (4 bits) which are stored with flags (10 bits) in 4 bytes . LWLock:BufMappingLock is immediately released. and only then LWLock:content_lock is set in the buffer header which provides access to the buffer and the rest of the header contents.
Would it be worth increasing NUM_BUFFER_PARTITIONS? The value does not depend on the size of the buffer cache, it depends on min(CPU cores , active processes) , but more on the duration of holding LWLock . If NUM_BUFFER_PARTITIONS were increased to 1024, then MAX_SIMUL_LWLOCKS = 200 would have to be increased, it should be no less. A simple increase does not give an effect and in some forks PostgreSQL changes the algorithm for lightweight locks by introducing a queue parameter `lwlock_shared_limit` configurations . Complex algorithms are not always justified due to overhead costs . For example, in linux on nvme optimal planner "none" . https://www.postgresql.org/message-id/8c4a5f06-7476-4646-bb8a-6581a26b0650%40enterprisedb.com

Pinning the buffer (pin) and locking content_lock

- pin used to ensure that a block in the buffer is not replaced by another
- to read or change the contents of a block in a buffer, a lightweight content_lock lock is required , a reference to which is stored in the `Buffer Descriptors block descriptor`
- the lock must be held for a short time, unlike the pin
- to remove the space occupied by the string (HOT cleanup or vacuum) after pin and Exclusive the process waits until other processes do not have pin (i.e. pincount = 1)
- to add a new line to a block or change xmin , xmax existing rows the process must acquire the content_lock lock type Exclusive
- if a process has a pin and Shared content_lock , it can change some bits in t_infomask , in particular the commit / rollback status

Pinning the buffer (pin) and locking content_lock

Pinning can be held for a long time and is used to ensure that a block in the buffer is not replaced by another. To read or change the contents of a block in the buffer, a lightweight content_lock is needed , a reference to which is stored in the block descriptor `Buffer Descriptors` . The size of each block descriptor is 64 bytes (aligned). This lock must be held for a short time, unlike pin.

1. To access the lines and their headers in the block, the following are set: pin and content_lock (Exclusive or Shared depending on the intentions of the process) .

2. After finding the necessary content_lock lines can be removed, but the pin will not be removed and in this mode the process will be able to read the lines of the block that it saw while the process had content_lock .

3. To add a new line to a block or change xmin , xmax existing rows process must obtain content_lock type Exclusive. With Exclusive , no one can have Shared content_lock and accordingly see new lines that are in the process of changing. Old lines can continue to be read, since they are not changing anyway: they cannot be cleared and frozen due to the retention of the event horizon.

4. If the process has pin and Shared content_lock , it can change some bits in t_infomask , in particular the commit / rollback status. These bits can even be lost, in which case the process will simply recheck the transaction status. Change bits and xmin which are related to freezing, however, cannot be done, for this you need Exclusive content_lock and such changes are logged . And what about checksums ? If any bits change, the checksum will be different, but the checksum changes before the block is written to disk.

5. To remove the space occupied by the string (HOT cleanup or vacuum) after pin and Exclusive, the process waits until other processes do not have pin (i.e. pincount = 1) . After reaching pincount = 1 (and getting Exclusive if it was removed), you can free up space. It is interesting that other processes can increase pincount (pin a block, showing the intention to work with its contents, because they cannot load a block into another buffer) , since due to Exclusive they will not be able to set Shared, which is needed to look into the block.

If pincount >1 , then (auto)vacuum writes itself into the block descriptor field waiting for pincount 1 , removes Exclusive and waits. HOT cleanup does not wait. There can only be one waiting, but this is normal, since the table can only be cleaned by one vacuum process.

Freeing buffers when deleting files

- When deleting a database, a full scan of all buffer descriptors is performed
- A full descriptor scan is performed if the size of the relation being deleted is greater than 1/32 of the buffer pool
- in other cases, the search for descriptors is done through a hash table
- when increasing the buffer cache from 1GB to 16GB, the time to search for buffers when deleting a table increases by an order of magnitude
- search is performed when files are deleted as a result of DROP , TRUNCATE and vacuum (if file truncation is not disabled)

```
pgbench --file=CreateAndDrop.sql -j 1 -c 1 -T 10
TPS for HP 128 MB 4 17
TPS for HP 1 GB 37 5
TPS for HP 4GB 227
TPS for HP 8GB 127
TPS for HP 16 GB 55
TPS for HP 18 GB 33
```

CreateAdndDrop.sql:

```
create table x(id int );
insert into x values (1);
drop table x;
```


Freeing buffers when deleting files

When a database is dropped, a full scan of all buffer descriptors (BufferDesc) is performed to find buffers that belong to database files . If the header indicates that the buffer does not belong to a database, it is skipped. If it does belong, a SpinLock is placed on the buffer descriptor , the descriptor is released, and the SpinLock is being removed.

A full scan is also performed if the size of the relation being deleted is greater than 1/32 of the buffer pool :

```
#define BUF_DROP_FULL_SCAN_THRESHOLD (uint64) ( NBuffers / 32)
```

In other cases (deletion, truncation of files) the search for buffers is performed by range and using a hash table, which is **also not fast** . Files can be deleted and truncated by vacuum, DROP, TRUNCATE command over permanent objects . Temporary objects do not store blocks in the buffer cache .

When the buffer pool size is large, the duration of these operations can be significant.

Speed of creating and deleting a small table using commands:

```
begin transaction;
create table x(id int );
insert into x values (1);
drop table x;
commit;
pgbench --file=CreateAndDrop.sql -j 1 -c 1 -T 10
```

TPS for `shared_pool` without Huge Pages (HP) size `128MB - 433`

```
1GB 367
4GB 22 0
8GB 123
16GB 43
18GB 32
```

The time to search for buffer descriptors in the hash table when deleting a small table increases by 10 times when the buffer pool increases from 1 GB to 16 GB . Using Huge Pages speed does not change significantly, since the buffer pool is not scanned.

Optimized file extension

- to extend the file size, two functions `mdzeroextend ( SMgrRelation reln , ForkNumber forknum , blocknumber blocknum , int nblocks , bool skipFsync )` and `mdextend (..)`
- `mdzeroextend (..)` function appeared in version 16 and can expand a file with several blocks (8Kb) at once in one call to the operating system
- when expanding files for more than 8 blocks (64Kb) the `posix_fallocate () system call is used` . In this case:
  - the command to fill blocks with zeros is not transmitted to the disk, but it will be considered that the blocks are filled with zeros
  - for added blocks, space for pages (4Kb) is not allocated in the page cache linux
  - file systems may not allocate real disk space (delayed allocation) and mark in the metadata that the block ranges contain zero values (sparse file)

Optimized file extension

To extend the file size, use the `mdzeroextend ( SMgrRelation reln , ForkNumber forknum , blocknumber blocknum , int nblocks , bool skipFsync )` or `mdextend ( SMgrRelation reln , ForkNumber forknum , blocknumber blocknum , const void *buffer, bool skipFsync ) .mdextend (..)` expands a file by one block (8Kb) . The `mdzeroextend (..) function` appeared in version 16 and can expand a file by several blocks (8Kb) at once in one call to the operating system. When expanding files for more than 8 blocks (64Kb) the posix_fallocate () system call is used . In this case:

1) the command to fill blocks with zeros is not transmitted via the I/O bus to the disk , but the file system and storage system (if implemented in the storage system) will consider that the blocks are filled with zeros

2) no space is allocated for pages (4Kb) in the page cache for added blocks linux

3) file systems may not allocate real space on the disk (delayed allocation) and mark in the metadata that block ranges contain zero values (sparse file). When writing to such blocks, the file system will allocate space. In file systems that are not recommended for use and which are usually implemented with errors (xfs) a false error may be issued : `could not extend file " ... " with FileFallocate ()` due to metadata corruption on such file systems.

For reference: if the number of blocks is less than 8, then the call chain FileZero (..) -> pg_pwrite_zeros (..) -> pg_pwritev_with_retry (..) is used ; for writing to WAL , pg_pwrite_zeros (..) is used if the configuration parameter wal_init_zero = true is set , if not set, then pg_write (..) is used.

Implementation of the record in version 16:

https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=4d330a61bb1969df31f2cebfe1ba9d1d004346d8

File resizing and buffer cache

- if there is no space in the file, then relations files are expanded in blocks from 1 to 64
- autovacuum and vacuum can truncate the last fork file at the last stage of table processing main or even delete the fork files if there are no lines in at least **a thousand blocks** from the end of the fork
  - If an exclusive lock on the table is not obtained within **5 seconds** , then truncation is not performed.
  - If an exclusive lock is obtained, **the hash table** is searched for buffers with blocks that will be truncated.
  - If not found, the buffer descriptors are scanned completely.

File resizing and buffer cache

If there is no space in the file, then relations files are expanded in blocks from 1 to `MAX_BUFFERS_TO_EXTEND_BY= 64` blocks. For commands that process a large number of lines, the BulkInsertStateData flag is used. bistate . If a file is expanded by a set of blocks, it will most likely continue to expand. Therefore, subsequent extensions will be performed by sets of blocks of the same size. This improves performance by preventing the mdzeroextend () function call from switching between different system calls: posix_fallocate () and write (), which would reduce performance. The limit for switching from single-block to multiple-block expansion is if the command requires at least 8 blocks (64K).

Autovacuum and vacuum can truncate the last fork file at the last stage of table processing main or even delete the fork files if there are no lines in at least a thousand blocks from the end of the fork . The exact formula is:

```
min( ( REL_TRUNCATE_MINIMUM= 1000) or ( relsize / 16 ))
```

Truncation is performed by the lazy_truncate_heap (..) function , which tries to get an exclusive lock on the table, but if it cannot get the lock within VACUUM_TRUNCATE_LOCK_TIMEOUT =5 seconds, then truncation is not performed. If it can get the lock, then it executes RelationTruncate (..), which calls smgrtruncate (..), which calls DropRelationBuffers (..), which must free the buffers that store the blocks being truncated. If at least one buffer is not freed, then bgwriter or checkpointer will crash and the instance will restart. The function first tries to find buffers through the hash table. If it does not find at least one buffer, it scans the buffer descriptors completely. You can disable this phase of the autovacuum operation with parameters at the table level and even for the TOAST table separately . Examples of commands:

```
CREATE TEMP TABLE reloptions_test ( i INT NOT NULL, j text)
WITH ( vacuum_truncate =false , toast.vacuum_truncate =false );
SELECT reloptions FROM pg_class WHERE oid = ' reloptions_test ':: regclass ;
                reloptions
-------------------------------------------
{ vacuum_truncate = false,autovacuum_enabled =false}
ALTER TABLE reloptions_test RESET ( vacuum_truncate );
VACUUM ( TRUNCATE FALSE , FULL TRUE ) reloptions_test ;
```

a wait event VacuumTruncate for monitoring .

Prefetching blocks

- pre-reading data blocks into linux page cache
- are implemented by calling the operating system `posix_fadvise ( VfdCache [file]. fd , offset, amount, POSIX_FADV_WILLNEED)`
- to monitor whether there are any delays associated with this call, there is a wait event DataFilePrefetch
- Bitmap Heap Scan index access method uses prefetch in the table block scanning phase
- vacuum and analysis also uses prefetch . The limit on the number of buffers is set by the parameter `maintenance_io_concurrency`
- in version 17 the parameter `io_combine_limit appeared`
  - Default 128Kb
  - The range of values is from 8 KB to 256 KB
  - Sets how many blocks can be combined into one call to linux

Prefetching blocks

Prefetch - preliminary reading of data blocks into the Linux page cache . For this, a system call with the `POSIX_FADV_WILLNEED parameter is used` , which tells the operating system read a block (size `BLCKSZ=8 Kb`). The call is called " asynchronous " because it returns the result without delay, before the blocks are actually read from the file.

For preliminary reading the function PrefetchBuffer (Relation reln , ForkNumber) is used forkNumber , BlockNumber blockNum) , which calls PrefetchLocalBuffer (..) for temporary objects or PrefetchSharedBuffer (..) for objects using the buffer cache. The functions eventually call linux by calling `posix_fadvise ( VfdCache [file]. fd , offset, amount, POSIX_FADV_WILLNEED)` .

To monitor whether there are any delays associated with this call, there is a **DataFilePrefetch wait event** .

Bitmap Heap Scan index access method uses prefetch at the table block scanning phase.

Vacuum and analysis also uses prefetch . The limit on the number of buffers is set by the parameter `maintenance_io_concurrency` .

`pg_prewarm` extension can also use prefetch .

Version 17 introduces the `io_combine_limit parameter,` 128K by default . The range of values is from 8K to 256K. Sets how many blocks can be combined into one call to the operating system to reduce the number of system calls. There is no combining before version 17.

How much data can be read-ahead ? read disk controller from specific block device:

```
postgres@tantor:~$ sudo blockdev --getra /dev/sda
256
```

Measured in 512 bytes, default 256 * 512 bytes = 128Kb

For a set of disks in RAID , you can set larger values:

```
postgres@tantor:~$ blockdev -- setra 4096 /dev/ sda
```

Performance pg_stat_recovery_prefetch

- pre-reading is used by the startup process
- recovery_prefetch parameter
  - set by default to try
  - startup process when starting an instance and on replicas
- wal_decode_buffer_size parameter
  - sets the size of the log records that the startup process reads ahead of time to determine which blocks to preload into the buffer cache
  - default value 512Kb

```
select * from pg_stat_recovery_prefetch \ gx
stats_reset | 2044-01-01 11:11:11.000000+03
prefetch | 2242 prefetched number blocks
hit | 52138 were not loaded, were already in the buffer cache
skip_init | 139 were not loaded because they were initialized with zeros
skip_new | 6 did not load because the blocks did not exist
skip_fpw | 1868 were not loaded because the WAL contained a full block image
skip_rep | 52272 prefetched command has already been sent
wal_distance | 0 number of WAL blocks read in advance for prefetch
block_distance | 0       number of blocks in prefetch process
io_depth | 0       number of outstanding prefetch calls
```


Performance pg_stat_recovery_prefetch

Prefetching is used by the startup process. Recovery is performed by a single process, and prefetching speeds up its operation. When a physical replica is running, the walreceiver process saves WAL blocks and they will be in the page cache if there is no delay in rolling out changes due to the max_standby_streaming_delay (and long query on the replica) and recovery_min_apply_delay parameters .

recovery_prefetch parameter is set to try by default . There is no need to change the value. It is used by the startup process . This process runs after the instance is started and continuously on replicas. The process reads WAL files. The log records contain references to blocks that are changed by log records and block images. The startup process calls the PrefetchBuffer (..) function on these blocks, which sets a recommendation to the operating system to read the blocks into the page cache.

wal_decode_buffer_size parameter sets the size of the log records that the startup process reads in advance to determine which blocks to prefetch into the buffer cache if recovery_prefetch not disabled. If the value is specified without units, it is assumed to be specified in bytes. The default value is 512K.

The view displays prefetch statistics. If all values are zero, there was no recovery when the instance was started (the instance was stopped gracefully).

The wal_distance , block_distance and io_depth columns contain current values, while the other columns contain statistics accumulated since the instance was started, which can be reset using the function:

```
 select pg_stat_reset_shared (' recovery_prefetch ');
select * from pg_stat_recovery_prefetch \ gx
-[ RECORD 1 ]--+------------------------------
stats_reset | 2044-01-01 11:11:11.000000+03
prefetch | 2242  prefetched number blocks
hit | 52138 were  not loaded, were already in the buffer cache
skip_init | 139   did not load because they were initialized with zeros
skip_new | 6      did not load because the blocks did not exist
skip_fpw | 1868  were not loaded because the WAL contained a complete image of the blocks
skip_rep | 52272  pre-read command has already been sent
wal_distance | 0      number of WAL blocks read in advance for prefetch
block_distance | 0       number of blocks in prefetch process
io_depth | 0      number of outstanding prefetch calls
```

Extension pg_prewarm

- standard extension
- set by adding the library to the parameter:
  `alter system set shared_preload_libraries =' pg_prewarm ';`
- saves the address of each block to a text file in the root of the PGDATA directory , which will be loaded into the buffer cache after the instance is restarted

```
postgres=# select autoprewarm_dump_now ();
 autoprewarm_dump_now
----------------------
                  264
(1 row)
postgres@tantor       :~$       head       $PGDATA/
autoprewarm.blocks
<< 264 >>
0,1664,1262,0,0
0,1664,6100,0,0
5,1663,1259,0,0
...
```

```
postgres=# \ dconfig * prewarm *
List of configuration parameters
Parameter | Value
----------------------------------+------
-
 pg_prewarm.autoprewarm | on
 pg_prewarm.autoprewarm_interval | 5min
(2 rows)
postgres=# select pg_prewarm (' pg_class
');
 pg_prewarm
------------
19
```


Extension pg_prewarm

Standard extension implementing periodic (or when stopping an instance) saving of addresses of blocks located in the buffer cache to a text file and loading of these blocks into the cache after restarting the instance . The file name is `autoprewarm.blocks` and does not change. The file is created in the `PGDATA directory` . The first line of the file is the number of blocks, data about which is saved in the file.

Each line contains 5 fields with the address of one block: `db, tbs , file, fork, block` .

The extension is installed by adding the library to the parameter:

`alter system set shared_preload_libraries =' pg_prewarm ';`

The functions (if needed) are installed by the **`create extension pg_prewarm command ;`**

After the instance is restarted, the background process `autoprewarm leader is started` , which While the instance is running or when the instance is stopped, it saves data to a file in the background . 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. The extension has two parameters:

```
\ dconfig * prewarm *
 pg_prewarm.autoprewarm | on
 pg_prewarm.autoprewarm_interval | 5min
```

Parameter `pg_prewarm.autoprewarm` can be disabled starting the `leader process` . Changing this parameter requires restarting the instance. You cannot start or stop a background process while the instance is running , `the autoprewarm_start_worker ()` function useless. The frequency of saving data 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.

`autoprewarm_dump_now ()` function immediately updates or creates the `autoprewarm.blocks file` . This can be useful if `the leader 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` .

You can load blocks of one relation into the buffer cache (`mode=buffer`) or only to page cache by synchronous call (`mode=read`) or `posix_fadvise (.., POSIX_FADV_WILLNEDED) )` (`mode= prefetch`) a function that has 5 parameters, 4 of which have default values. When calling the function, it is enough to specify the object name: **`select pg_prewarm (' pg_class ', ' prefetch ');`** The function will return the number of blocks loaded.

bgwriter background writing process

- The bgwriter process writes back dirty buffers and marks them as clean
- bgwriter reduces the likelihood that processes will encounter dirty blocks when searching for a candidate buffer (victim) to be replaced by another block
- " free block search " is usually a candidate buffer for evicting from the block buffer, since all buffers are usually occupied and the free block list is empty
- when a dirty block is evicted from the buffer, there is no access to the I/O bus, this is a " writeback ": copying from memory (buffer) to memory (linux page cache)

```
select name, setting, context, max_val , min_val from pg_settings where name ~ ' bgwr ';
name | setting | context | max_val | min_val
------------------------+---------+---------+-----------+---------
 bgwriter_delay | 200 | sighup | 10000 | 10
 bgwriter_flush_after | 64 | sighup | 256 | 0
 bgwriter_lru_maxpages | 100 | sighup | 1073741823 | 0
 bgwriter_lru_multiplier | 2 | sighup | 10 | 0
```


bgwriter background writing process

Dirty buffers can be written to disk (" cleaned " by clearing the BM_DIRTY flag) by processes working with the buffer cache , including checkpointer, bgwriter , server processes, and autovacuum worker processes. The `bgwriter process` writes dirty buffers and marks them as clean. `bgwriter` reduces the likelihood that server processes will encounter dirty blocks when searching for a candidate buffer (victim) on eviction to replace with another block . When evicting a dirty block from the buffer , there is no access to the I/O bus, it is a copy from memory (buffer) to memory (linux page cache). Latencies are not as critical as it may seem. The processes `bgwriter , walwriter , bgworker` the names sound similar, but these are different processes. The work of `the bgwriter` process configured by parameters:

```
select name, setting, context, max_val , min_val from pg_settings where name ~ ' bgwr ';
name | setting | context | max_val | min_val
------------------------+---------+---------+-----------+---------
 bgwriter_delay | 200 | sighup | 10000 | 10
 bgwriter_flush_after | 64 | sighup | 256 | 0
 bgwriter_lru_maxpages | 100 | sighup | 1073741823 | 0
 bgwriter_lru_multiplier | 2 | sighup | 10 | 0
```

`bgwriter_delay` on how many milliseconds does `bgwriter` sleep between iterations . `bgwriter_flush_after` – the number of blocks after sending for writing which the 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. server processes (" recent_alloc ") in previous cycles. The average value is multiplied by `bgwriter_lru_multiplier` and specifies how many buffers need to be flushed 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` - the maximum number of blocks that are written in one iteration, with a value of zero `bgwriter` stops working . Based on this, it makes sense to set `bgwriter_lru_maxpages` to the maximum value.

What if the server processes did not use new buffers in previous iterations ? To avoid a " slow start " , the iteration will scan at least:

`NBuffers /120000* bgwriter_delay+reusable_buffers_est` blocks.

For a buffer cache size of 128 MB and 200 milliseconds of latency, this would be **27** + `reusable_buffers_est` blocks.

Algorithm for clearing the buffer cache by the bgwriter process

- the buffer is not included in the free buffer list, the dirty buffer becomes clean
- only dirty, uncommitted blocks with `usage_count = 0` are written to disk
- bgwriter does not change **`usage_count`**
- when checksum calculation is enabled, the block is copied from the buffer cache to the local memory of the bgwriter process and the checksum is calculated in the local memory and stored in the block header

Algorithm for clearing the buffer cache by the bgwriter process

The block is written by the SyncOneBuffer (..) function. First, a spinlock is taken on the block descriptor and the BM_LOCKED bit is set . The values are checked: **refcount =0 (the block is not needed by processes) , usage_count = 0 (falls into the gradation of long unused) , bit BM_DIRTY = 1 (dirty) , BM_VALID = 1** and if the values are not as given, then the spin lock is removed and the block is not flushed to disk. Otherwise, the buffer is pinned, a lightweight shared lock is taken, the function of transferring the buffer to the Linux page cache is called , the lock and pinning are removed.

During the process of flushing the buffer, other processes may have time to lock and pin the buffer, change the hint bits that are allowed to be changed with a Shared lock and pin .

The LSN is read from the block in the buffer and the function is executed XLogFlush (XLogRecPtr record) , flushing the WAL buffer contents up to this LSN. This ensures the Write Ahead logic - the log with changes to the block must be written before the block itself .

If checksumming is enabled, the buffer contents are copied to the local memory of the bgwriter process using `the memcpy ()` system call . The checksum is calculated on the local copy and this 8K copy is passed to the kernel code. linux , which places a block in the form of two 4Kb pages in the linux page cache .

Why is it copied to local memory ? Because other processes in the block can change the hint bits (infomask) while bgwriter calculates the checksum and the checksum will be incorrect even if one bit changes. Therefore, to calculate the checksum, the block is copied to local memory. Some performance reduction when checksum calculation is enabled is associated with copying from memory to memory, and not with the load on the processor's computing power.

A set of flags (BM_JUST_DIRTIED , BM_IO_IN_PROGRESS , BM_CHECKPOINT_NEEDED , BM_IO_ERROR) are checked, which are used to track changes in the block during writing to disk. If the flags show that " everything is clean " (other processes did not change the contents of the block), then the BM_DIRTY flag is cleared in the buffer descriptor and the buffer becomes " clean " and the block descriptor lock is released.

Since long unused ones are pushed out (usage_count = 0 and refcount =0) of the buffer, then the probability that the block will be needed by another process is small ; that there will be waits for locks ; that a write to WAL will be required. The XLogFlush (XLogRecPtr record) function first checks that the LSN is less than the one already written to WAL.

Buffer is not included in the free list, buffer becomes clean.

Performance pg_stat_bgwriter

- displays statistics on the performance of bgwriter across the entire instance
- reset statistics: `pg_stat_reset_shared (' bgwriter ');`
- total number of buffers written to disk: `buffers_backend + buffers_clean + buffers_checkpoint`
- buffers_backend - how many buffers **were cleared** (sent for writing) by server processes
- buffers_clean - how many buffers cleared bgwriter
- **buffers_backend *100/ buffers_alloc** in what percentage of cases server processes collided with a dirty buffer , should be less than 1%

```
select * from pg_stat_bgwriter \ gx
-[ RECORD 1 ]---------+------------
checkpoints_timed | 1567
checkpoints_req | 97
checkpoint_write_time | 1463587
checkpoint_sync_time | 59994
buffers_checkpoint | 86752
buffers_clean | 4234
maxwritten_clean | 25
buffers_backend | 1570184
buffers_backend_fsync | 0
buffers_alloc | 450550
stats_reset | 202 5 -0 1 - 01 09:13:
```


Performance pg_stat_bgwriter

Displays statistics about the performance of bgwriter for the entire instance. The view contains one row. The column `maxwritten_clean` gives the number of times bgwriter paused because it reached `bgwriter_lru_maxpages` or the calculated value of the number of blocks (taking into account `bgwriter_lru_multiplier`), if `bgwriter_lru_maxpages` already set to maximum value.

reset (synonyms: reset) data in a view using the function `pg_stat_reset_shared (' bgwriter '); The reset` time is specified in the `stats_reset column . Calling the pg_stat_reset ()` function without parameters does not reset statistics in this view or in the `pg_stat_io view .`

Total number of buffers written to disk: `buffers_backend + buffers_clean + buffers_checkpoint .`

`buffers_clean` - how many buffers **were cleared** (sent for writing) by bgwriter .

`buffers_backend` - how many buffers **were cleared** (sent for writing) by server processes and autovacuum. In the process of freeing buffers to use them for new blocks, server processes encountered dirty buffers and freed them by sending them for writing, which is undesirable, since it increases the execution time of commands. If `buffers_backend *2> buffers_clean` or `buffers_backend *100%/ ( buffers_backend + buffers_clean + buffers_checkpoint )>~50%` , then work bgwriter needs to be made more aggressive.

`buffers_alloc` - how many blocks were loaded ("allocated") from disk to buffer cache server processes . When processing data , server processes check the hash table to see if the required blocks are in the buffer cache . If the blocks are not in the buffer cache , they allocate . buffers and load blocks into them. What does " allocated " mean? If there are free buffers, they are used, and if not (and there usually aren't any, since they can only appear as a result of DROP, TRUNCATE), then they try to free the buffers using the clock sweep algorithm . The value itself doesn't say anything. The buffer cache may be too small relative to the typical volume of blocks that the processes have worked with since the statistics were reset. The instance may have just started and the server processes were filling the buffer cache.

buffers_backend *100/ buffers_alloc - in what percentage of cases did server processes encounter the need to flush a dirty buffer to disk. This value should be less than 1%.

For monitoring, queries that output percentages are usually used:

```
 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 ;
ckpt by time | ckpt by size | checkpointer | backend | backend_fsync | bgwriter |    speed
-------------+--------------+--------------+---------+---------------+----------+-------------
100.0% |    0.0% |  100.0% |    0.0% |    0.0% |    0.0% | 32 bytes/s
(1 row)


 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 '
 JOIN pg_settings delay ON delay.name = ' bgwriter_delay '
 JOIN pg_settings block ON block.name = ' block_size '
 ) np \ gx
```

https://dataegret.com/2017/03/deep-dive-into-postgres-stats-pg_stat_bgwriter-reports/

Extension pg_buffercache

- standard extension
- creates a representation pg_buffercache , which is a wrapper for the function pg_buffercache_pages () and two functions without parameters pg_buffercache_summary () and pg_buffercache_usage_counts ()
- bgwriter does not reduce usage_count , it is reduced by server processes
- distribution of buffers into six " baskets " usage_count should not be tilted to one side or the other

```
select * from pg_buffercache_usage_counts
();
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
0 | 16000 | 0 | 0
1 | 17 | 0 | 0
2 | 23 | 1 | 0
3 | 43 | 0 | 0
4 | 24 | 2 | 0
5 | 277 | 1 | 0
(6 rows)
```


Extension pg_buffercache

Standard extension. Creates a view. pg_buffercache , which is a wrapper for the pg_buffercache_pages () function and two functions without parameters pg_buffercache_summary () and pg_buffercache_usage_counts () . These two functions do not place locks on the buffer cache memory structures, unlike the first function, and are fast.

To configure the bgwriter process to work Of particular interest are:

select * from pg_buffercache_usage_counts ();

bgwriter does not decrease usage_count , it is decreased by server processes. **In order for bgwriter cleared the buffer, it is necessary that usage_count = 0 and pinned=0** . If there are no such buffers, then bgwriter it would be pointless to spin cycles, loading the processor core and not freeing buffers. If the amount of dirty is large or server processes often encounter dirty buffers - for example, buffers_backend *100/ buffers_alloc >1% , then in this case it is necessary for checkpointer to work more actively: it is worth increasing the frequency of checkpoints. The checkpointer process flushes **all** dirty buffers to disk, to usage_count does not pay attention and does not change it .

Version 17 introduces a function that removes one block from the pg_buffercache_evict **cache** (buffer) and some columns from pg_stat_bgwriter moved to the new view pg_stat_bgwriter , part (for server processes) moved to pg_stat_io : **\d pg_stat_bgwriter**

```
View " pg_catalog.pg_stat_bgwriter "
Column | Type |
-----------------+--------------------------+
 buffers_clean | bigint |
 maxwritten_clean | bigint |
 buffers_alloc | bigint |
 stats_reset | timestamp with time zone |
postgres =# \d pg_stat_checkpointer
View " pg_catalog.pg_stat_checkpointer "
Column | Type |
----------------+--------------------------+
 num_timed | bigint |
 num_requested | bigint |
 write_time | double precision |
 sync_time | double precision |
 buffers_written | bigint |
 stats_reset | timestamp with time zone |
https://www.postgresql.org/message-id/flat/20230210004604.mcszbscsqs3bc5nx%40awork3.anarazel.de
```

Setting the buffer cache size

- the size is set by the `shared_buffers parameter`
- when changing the value you need to restart the instance
- you can't clear the cache, only by restarting the instance
- linux page cache can be cleaned
- distribution of buffers across six `usage_count values` should not have a clear bias in one direction or the other, otherwise the displacement algorithm is ineffective
- for evaluation, you can use the `usagecount_avg column` , the value of which should be approximately in the middle of the interval from 0 to 6 (2.5):

```
select * from pg_buffercache_summary ();
buffers_used|buffers_unused|buffers_dirty|buffers_pinned|usagecount_avg
------------+--------------+------------+--------------+--------------
384 | 16000 | 0 | 0 | 4.37760416666
```


Setting the buffer cache size

The size is set by the `shared_buffers parameter` . If the value changes, you must restart the instance.

`usage_count` columns and `buffers` help determine whether the buffer cache size is optimal for current activity and settings instance. If there is a bias in the buffers column : towards `u sage_count =5` this means that the buffer cache size is insufficient. If most of the buffers in the row with `usage_count = 1` the buffer cache size can be reduced. If 0 then either the instance has just rebooted and most of the buffers are free, or the buffer cache size can be reduced.

Distribution of buffers into six " baskets " `usage_count` should not have a clear bias in one direction or another. If there is a bias, the displacement algorithm loses its effectiveness. For evaluation, you can use the `usagecount_avg column` , the value of which is should be approximately in the middle of the range from 0 to 6:

```
select * from pg_buffercache_summary ();
buffers_used|buffers_unused|buffers_dirty|buffers_pinned| usagecount_avg
------------+--------------+------------+--------------+--------------
384 | 16000 | 0 | 0 | 4.3 776041666 6
```

It is worth looking at the distribution itself - the " figure " of the two-dimensional graph `of usage_count` horizontally, buffers vertically, returned by the table function **`pg_buffercache_usage_counts ()`** .

A special case of buffer cache configuration when it is larger than the cluster size. In this case, there is no displacement of blocks from the cache and this is the most optimal, since server processes do not need to load blocks and look for free blocks.

Example output of objects occupying the largest part of the cache:

```
create or replace view buffercache as
select relname , buffers, relpages pages, (buffers*8)/1024 as size_mb ,
usage, 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(*) 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 20;
```

```
             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 | 14006 | 3 | 1 | 0.11 | 100.14
 pgbench_accounts | 1698 | 1695 | 13 | 8490 | 0 | 0 | 0.00 | 100.18
 pgbench_history | 664 | 637 | 5 | 3320 | 0 | 0 | 0.00 | 104.24
 pgbench_accounts_pkey | 276 | 276 | 2 | 1378 | 0 | 0 | 0.00 | 100.00
 pg_class_oid_index | 64 | 1879 | 0 | 210 | 1 | 0 | 1.56 | 3.41
 pg_class_relname_nsp_index | 63 | 4237 | 0 | 123 | 1 | 0 | 1.59 | 1.49
pg_toast_2618 | 46 | 84 | 0 | 106 | 0 | 0 | 0.00 | 54.76
 pg_attribute | 42 | 4421 | 0 | 165 | 4 | 0 | 9.52 | 0.95
 pg_proc | 35 | 103 | 0 | 89 | 0 | 0 | 0.00 | 33.98
 pg_statistic | 29 | 182 | 0 | 71 | 0 | 0 | 0.00 | 15.93
 pg_attribute_relid_attnum_index | 16 | 11663 | 0 | 78 | 1 | 0 | 6.25 | 0.14
 pg_operator | 14 | 14 | 0 | 66 | 0 | 0 | 0.00 | 100.00
 pg_proc_proname_args_nsp_index | 12 | 34 | 0 | 30 | 0 | 0 | 0.00 | 35.29
 pg_depend_reference_index | 12 | 6991 | 0 | 56 | 3 | 0 | 25.00 | 0.17
 pg_type_typname_nsp_index | 12 | 505 | 0 | 51 | 2 | 0 | 16.67 | 2.38
 pg_proc_oid_index | 11 | 12 | 0 | 40 | 0 | 0 | 0.00 | 91.67
 pg_type | 10 | 487 | 0 | 33 | 1 | 0 | 10.00 | 2.05
 pg_depend_depender_index | 10 | 9076 | 0 | 46 | 2 | 0 | 20.00 | 0.11
(20 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` zero .
`buffers > pages` since fsm layer blocks are cached and vm , as well as in cases of irrelevant statistics .

Parameter `synchronize_seqscans`

- when reading the table larger than 1/4 buffer cache using Seq Scan table method
  - a buffer ring is used
  - the new process " synchronizes " with the process that is already scanning the table blocks to read blocks from the ring together
  - blocks in the buffer cache are read by running processes at approximately the same time, rather than being loaded into the buffer cache multiple times
- the order of rows returned by queries that lack an ORDER BY clause can be arbitrary
- enabled by default:

```
show synchronize_seqscans ;
 synchronize_seqscans
----------------------
on
```


Parameter `synchronize_seqscans`

The parameter is enabled by default . When reading the table larger than 1/4 of the buffer cache using the Seq Scan table method , in addition to using a buffer ring, the new process " synchronizes " with the process that is already scanning the table blocks.

The reading process is considered the leader , and the processes that started reading the table later are followers . Followers start reading the table from the last position that the leader read, and not from the first block. It turns out that the blocks in the buffer cache are read by the running processes at approximately the same time, and are not loaded into the buffer cache several times. When the leader reaches the end of the table, one of the processes that has not completed the selection becomes the leader. The new leader starts reading the table from the beginning and reads the blocks that it has not read. The order of rows returned by queries that do not have an ORDER BY clause can be arbitrary.

Side effect: the leader can finish reading the table at an arbitrary block, the address of that block is saved (the last position updated by the leader) and other processes will start reading blocks of that table from that position.

Practice

- Part 1. Extending pg_buffercache
- Part 2. Buffer rings
- pg_prewarm extension
- The bgwriter background writing process

Practice

In practice, you will see the use of extensions for diagnostics and optimization of filling the buffer cache.

11

Checkpoint

Checkpoint

- executes the checkpointer process:
  - periodically after `checkpoint_timeout`
  - by log growth `max_wal_size`
  - at the end of the instance stop or start procedure
  - promoting the replica
  - commands checkpoint , create database
  - when booking

```
postgres=# checkpoint;
CHECKPOINT
LOG: checkpoint starting: immediate force wait
LOG: checkpoint complete: wrote 5 buffers (0.0%);
```


Checkpoint

Performed by the checkpointer process . Checkpoints are performed : periodically, at the end of the instance stop and start procedure, replica promotion, backup, command checkpoint , creating a database . On the replica, checkpoints are not initiated, but restart points are executed . In case of an instance crash and subsequent restart, the checkpoint algorithm must ensure that the log data starting from the LSN of the beginning of a successfully completed checkpoint , i.e. written to pg_control (at the last phase of execution), will be sufficient to restore the cluster . Checkpoints allow you to avoid storing WAL segments that are not needed for recovery.

Properties of checkpoints that are reflected in the cluster log:

IS_SHUTDOWN (shutdown) stopping an instance in fast or smart mode

END_OF_RECOVERY (end-of-recovery) is called by the startup process at the end of recovery.

IMMEDIATE (immediate) complete an already started (if any) checkpoint at maximum speed, ignoring checkpoint_completion_target and immediately execute the checkpoint also at maximum speed

FORCE (force) even if there was no entry in WAL . Executed by checkpoint command , replica promotion pg_promote () , instance stop

WAIT (wait) return control only after the checkpoint is completed

CAUSE_XLOG (wal) by parameter max_wal_size when switching WAL segment

CAUSE_TIME (time) by time specified by checkpoint_timeout parameter

FLUSH_ALL (flush-all) saves blocks of unlogged objects, set when creating a database using the FILE_COPY method

Properties can be combined with each other. For example, the checkpoint command sets the properties immediate force wait .

The properties are described in the RequestCheckpoint function (int flags) checkpointer.c file

Steps to perform a checkpoint

- if the instance stops, in `pg_control` file the status of the start of the cancellation of the instance is recorded
- the LSN of the next log record is calculated . This will be the LSN of the start of the checkpoint
- waiting for processes to remove the DELAY_CHKPT_START flag
- slru buffers and other shared memory structures are flushed to disk
- checkpointer runs through all buffer descriptors in a loop and sets the BM_CHECKPOINT_NEEDED flag for dirty blocks, saves the block address (5 numbers) in the Checkpoint BufferIds memory structure for subsequent sorting
- after setting the flag checkpointer does not lock the buffer and the block in the buffer can be flushed to disk and replaced with another

Steps to perform a checkpoint

When a checkpoint is executed, the following actions are performed.

If the instance stops, then in the `pg_control file` the status of the instance's start of blanking is recorded. The LSN of the next log entry is calculated. This will be the LSN of the start of the checkpoint, but checkpointer does not create a separate log entry about the start.

Other processes may set the DELAY_CHKPT_START flag . The list is collected virtual transaction identifiers whose processes set the flag. If the list is not empty, checkpointer waits for flags to be removed in a loop, sleeping for 10 milliseconds between flag removal checks. Other processes may set flags, but they do not matter, since they are set after the previously calculated LSN. The flag is set for a short time: when the process performs a logically related action non-atomically : creating different journal entries. For example, updating the transaction status in slru and creates a commit log entry.

Then checkpointer starts resetting with the `CheckPointGuts function ( .. )` to disk slru buffers and other shared memory structures in the files they cache and / or in the WAL and synchronization is performed on these files (fsync) . These log records must be related to the checkpoint and come after the LSN of its beginning.

The algorithm for performing actions related to writing dirty blocks of the buffer cache is described in the `BufferSync ( int flags ) function` :

Checkpoints of the type IS_SHUTDOWN, END_OF_RECOVERY, FLUSH_ALL write all dirty buffers, including those related to non-logged objects. The checkpointer process runs through all buffer descriptors in a loop, gets a SpinLock on one block simultaneously. Then it checks that the block is dirty and sets the BM_CHECKPOINT_NEEDED flag for dirty blocks , saves the block address in the shared memory structure `Checkpoint BufferIds` . Then it removes the SpinLock . The block address is the traditional 5 numbers of the `BufferTag structure` .

If some process clears the buffer, then this flag will be removed by the clearing process - it doesn't matter which process writes the block, the main thing is that all dirty buffers that were dirty at the start of the checkpoint are written to disk. Now checkpointer has a list of blocks that it will write to disk.

Steps to perform a checkpoint

- The stored block identifiers are sorted in the order: tblspc , relation, fork, block
- blocks are sent one by one to the linux page cache
- If checkpoint_flush_after is not zero, then synchronization is performed on already sorted ranges of blocks for each file.
- WAL stores a snapshot with a list of active transactions
- a checkpoint end log record is generated containing the LSN of the log record that was generated at the start of the checkpoint
- `pg_control` file stores the LSN of the generated log record
- WAL segments that should not be retained are removed

Checkpoint Execution Steps (continued)

Next, checkpointer sorts the block identifiers standard quick sort algorithm. The comparison is performed by the ckpt_buforder_comparator (..) function in the order: tblspc , relation, fork, block . The first one is tblspc and this is essential. Sorting, in particular, is needed to avoid such a situation where blocks are sent to tablespaces in order, loading one tablespace at a time . It is assumed that tablespaces are separately mounted file systems on different devices.

The number of blocks for each tablespace is calculated, and the size of the block set (slice) is determined so that the write to all tablespaces ends up approximately the same.

checkpointer sends one block at a time from its list using the SyncOneBuffer () function with periodic delays (according to the `checkpoint_completion_target configuration parameter` and the calculated write speed) to linux page cache .

If checkpoint_flush_after is not zero, then synchronization is performed. by already sorted ranges of blocks for each file. Combining sorted ranges of blocks (if any) for each file, checkpointer sends system calls to linux to write ranges of blocks to the linux page cache , which were previously "sent to disk" by processes.

For checkpoints (except for the one performed on instance shutdown), a snapshot with a list of active transactions is stored in the WAL using the LogStandbySnapshot () function . This can be useful for replicas when restoring from archived logs.

A log record is generated containing the LSN of the log record that was generated at the start of the checkpoint. The generated log record c is sent to WAL by the fdatasync system call (or another method) . The LSN of the generated checkpoint end record is stored in pg_control . The checkpoint is complete .

Next, checkpointer checks whether replication slots need to be invalidated because the slot has not been used for a long time . WAL segments that should not be retained are removed. To restore the instance, segments are needed starting with the segment containing the log record with the LSN of the beginning of the checkpoint. New ones are allocated or old WAL segments are cleared and renamed according to configuration parameters.

Checkpointer Process Configuration Parameters

- `log_checkpoints` by default `on` starting from version 15
- decrease the value of `checkpoint_completion_target` not recommended
- `checkpoint_timeout` - the main parameter that affects performance. The optimal value is about 20 minutes
- `max_wall_size` the second one that affects performance when setting a checkpoint

```
select name, setting, unit, min_val , max_val , context from pg_settings where
name like '%checkpoint%' or name = ' max_wal_size ';
name | setting | unit | min_val | max_val | context
-----------------------------+---------+------+---------+------------+---------
 checkpoint_completion_target | 0.9 | | 0 | 1 | sighup
 checkpoint_flush_after | 32 | 8kB | 0 | 256 | sighup
 checkpoint_timeout | 300 | s | 30 | 86400 | sighup
 checkpoint_warning | 30 | s | 0 | 2147483647 | sighup
 log_checkpoints | on | | | | sighup
 max_wal_size | 1024 | MB | 2 | 2147483647 | sighup
(6 rows)
```


Checkpointer Process Configuration Parameters

Parameters: **\ dconfig *checkpoint***

```
Parameter | Value
---------------------+-------
 checkpoint_completion_target | 0.9
 checkpoint_flush_after | 256kB
 checkpoint_timeout | 5min
 checkpoint_warning | 30s
 log_checkpoints | on
 max_wall_size | 1GB
```

`log_checkpoints` became enabled by default starting with version 15. When a checkpoint is executed or on replicas of the restart point, an entry with the details of the checkpoint execution is output to the cluster log.

If the time interval between checkpoints is less than specified in `checkpoint_warning` , the following message is displayed in the cluster log:

```
LOG: checkpoints are occurring too frequently ( 25 seconds apart)
HINT: Consider increasing the configuration parameter " max_wal_size ".
```

If the messages are rare, it means that the database was running commands that generated more than `max_wal_size` log entries in `checkpoint_warning` seconds. If such messages are constantly coming, then `max_wal_size should be increased` .

`checkpoint_flush_after` parameter range from 0 (call flush is disabled) to 2MB. Limits the amount of dirty blocks in the page cache linux and reduces the likelihood slowdowns when executing fsync at the end of a checkpoint or when committing transactions.

Decrease `the checkpoint_completion_target value` not recommended. When increasing `checkpoint_timeout` , it is worth increasing it proportionally so that the idle time remains 30 seconds. For example, for 20 minutes, set 0.97.

`checkpoint_timeout` - the main parameter that configures the checkpoint. The optimal value of about 20 minutes is determined empirically by DBMS manufacturers - this is the time after which changes stop being made to a new data block due to the block being filled completely and the activity of correcting the inserted row subsiding. Reflects human behavior: after 20 minutes, almost everyone gets tired of correcting data in an application or withdrawing money from an ATM.

Statistics for setting checkpointer parameters

- server processes when searching for a buffer to replace a block with their own do not skip dirty buffers, they are guided only by pin and usagecount
- In addition to checkpointer and server processes, bgwriter performs the flushing of dirty blocks
- bgwriter resets buffers with usagecount = 0 and **only for unpinned buffers**
- after setup `checkpoint_timeout` you can configure bgwriter : check how many buffers it cleared `pg_stat_bgwriter . buffers_clean` and whether it could clear :

```
select dirty from pg_buffercache_usage_counts () where pinned=0 and usage_count
=0;
dirty
-------
     0
(1 row)
```


Statistics for setting checkpointer parameters

Using a value of 20 minutes is a good approximation, but how can we estimate the value more accurately ? A value that is too large can increase the cluster startup time if an invalid shutdown occurs. This may not be important if the instance fails over to a replica. `PGDATA/pg_wal` directory size may play a role, but there may be a lot of space.

The checkpoint resets dirty blocks, which speeds up the execution of commands by server processes. If the checkpoint interval is greatly increased, the percentage of dirty blocks in the buffer cache may increase, and server processes will most likely encounter dirty buffers more often when looking for space for blocks. When searching for buffers to replace blocks, server processes do not skip dirty buffers; they are guided only by pin and usagecount . A special case: the entire cluster fits into the buffer cache. In this case, server processes will not look for space in the buffer cache . But usually the cluster size is larger than the buffer cache. In addition to checkpointer and server processes, bgwriter flushes dirty blocks . If bgwriter flushed the buffer, then this buffer is not flushed by checkpoint, since both processes flush only dirty buffers. However, bgwriter flushes buffers with usagecount =0 , and usagecount is decreased by one (`-= BUF_USAGECOUNT_ONE;`) only by processes, who are looking for free blocks (server, autovacuum) by calling the `StrategyGetBuffer ( ..) function` and **only for unattached buffers** . That is, bgwriter not sufficient.

Because `full_page_writes =on` (it is not worth disabling) , changing a block after a checkpoint causes a one-time saving of the block image, with the exception of unused space in the block (pd_upper minus pd_lower). The block is not saved immediately after the change, but during the process of bypassing the buffers by the checkpointer process . It can be assumed that more frequent checkpoints will more often write entire blocks. However, if the block has finished getting dirty, including by autovacuum (before autovacuum , a SELECT is possible, which will set the infomask bits about committing the transaction), which will clear the block of old row versions, then the block is unlikely to change and will require writing. You can estimate how long the database horizon is held plus the duration of the autovacuum cycle and focus on this value to set `checkpoint_timeout` .

After setup `checkpoint_timeout` you can configure bgwriter : check how many buffers it cleared `pg_stat_bgwriter . buffers_clean` and whether it could clear `select dirty from pg_buffercache_usage_counts () where pinned=0 and usage_count =0;`

checkpointer parameters

- `select pg_sleep (:t3) ;` keeps the database horizon, which is closer to the real load
- " triangle " in dirty :

```
select * from pg_buffercache_usage_counts
();
 usage_count | buffers | dirty | pinned
-------------+---------+-------+--------
0 | 6957 | 0 | 0
1 | 88 | 13 | 0
2 | 48 | 23 | 0
3 | 62 | 24 | 0
4 | 81 | 61 | 0
5 | 9148 | 7895 | 0
```

```
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
select pg_size_pretty ((:'t2':: pg_lsn - :'t1':: pg_lsn )/
:'t3' );
```

```
#!/bin/bash
for (( i = 30 0 ;i<= 72 00
;i+= 30 0 ))
do
export t3="$ i "
psql -f ckpt.sql
done
60 494kB
120 360kB
180 254kB
240 199kB
300 165kB
360 143kB
420 128kB
540 133kB
600 81kB
660 98kB
780 85kB
1200 60kB
1500 53kB
1680 56kB
1800 51kB
2400 45kB
```


checkpointer parameters

Batch file to measure the amount of WAL written when the `checkpoint_timeout` parameter changes :

```
#!/bin/bash
for (( i = 60 ; i <= 2400 ; i += 60 ))
do
export t3="$ i "
psql -f ckpt.sql
done
```

Script named **`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
select pg_sleep (:t3);
select pg_current_wal_lsn () AS t2\ gset
\o
select pg_size_pretty ((:'t2':: pg_lsn - :'t1':: pg_lsn )/:'t3');
```

if there is no load on the cluster, you can create a test load by running load generation in a separate window :

```
pgbench -T 6 0000 -P 10
pgbench (16.2)
starting vacuum...end.
progress: 10.0 s, 529.6 tps, lat 1.879 ms stddev 1.984, 0 failed
...
```

WAL size will depend on the TPS and the size of full blocks written to WAL after the checkpoint. If the size remains the same when changing `checkpoint_timeout` , it means that the same block will be written once after the changes are completed . Measurements should be performed after setting up autovacuum. Should I measure the size of writes to data files ? in PGDATA? Not necessarily, since the amount written to WAL correlates with the amount written to data files.

`select pg_sleep (:t3);` keeps the database horizon, which is closer to the real load. In the pgbench test , by default, transactions are short, HOT works cleanup.

checkpointer parameters

- `\! sleep $t3` does not hold horizon, TPS increases from ~44 to ~ 6 2 0

```
progress: 600.0 s, 618.6 tps, lat 1.609 ms stddev 1.036
progress: 900.0 s, 629.0 tps, lat 1.582 ms stddev 0.914
progress: 1800.0 s, 627.7 tps, lat 1.585 ms stddev 0.858
progress: 3300.0 s, 615.7 tps, lat 1.616 ms stddev 0.912
progress: 10500.0 s, 618.4 tps, lat 1.609 ms stddev 0.882
progress: 12900.0 s, 618.5 tps, lat 1.609 ms stddev 0.842
```

```
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
select pg_size_pretty ((:'t2':: pg_lsn - :'t1':: pg_lsn
)/:'t3');
```

```
#!/bin/bash
for (( i = 30 ; i <= 6 00 ;
i += 30 ))
do
export t3="$ i "
psql -f ckpt.sql
done
30 689 kB
60 567kB
90 452kB
120 393kB
150 368kB
180 340kB
210 332kB
240 408kB
270 332kB
300 310kB
33 0 307 kB
360 300 kB
390 300 kB
420 295 kB
450 292kB
480 291kB
```


checkpointer parameters (continued)

If you use `\! sleep $t3` , then TPS 40 increases to 600 and the checkpoint frequency can be any. If you do not use pgbench or create a real test for it, then you should use `\! sleep $t3`

The values of kilobytes per second written to WAL depend on the volume of modified data (TPS), and the values are not comparable for different loads.

Why is the default value for `checkpoint_timeout =300` (5 minutes) ? This value Suitable for the load generated by the default pgbench test.

The default test ("TPC-B") consists of three UPDATE s , one INSERT , and one SELECT in a single transaction. The pgbench documentation says: " It is very easy to use pgbench to produce completely meaningless numbers " .

Practice

- Part 1. Setting the frequency of checkpoints
- Part 2. Delay in instance startup. `recovery_init_sync_method` parameter
- Part 3. Checkpoint duration
- Part 4. Duration of the final checkpoint when stopping the instance
- Part 5. Duration of checkpoint after instance crash
- Part 6. Checkpoint on request

Practice

You will consider an example of how, with 30 thousand files in a cluster, file synchronization takes more than a minute.

You will learn

read checkpoint messages in the cluster diagnostic log ;

Consider how different types of checkpoints are performed.

12

12

Autovacuum

Vacuuming algorithm

- each table is vacuumed in a separate transaction
- a snapshot is held until the end of the transaction
- If a lock cannot be set on a table, the table is skipped.
- statistics are not collected for TOAST tables
- indexes of a single table can be cleaned by parallel worker processes, they can use HugePages for TID list
- temporary objects are cleared sequentially
- Five phases of vacuuming each table:
  - `SCAN_HEAP, VACUUM_INDEX, VACUUM_HEAP, INDEX_CLEANUP, VACUUM TRUNCATE`

Vacuuming algorithm

, a list of tables to be cleaned is built. This preparatory phase of the autovacuum cycle is called `initializing` .

Before **each** table is processed (including TOAST) , a transaction is opened and a snapshot is created. This allows locks to be released more quickly and the database horizon to be advanced.

ShareUpdateExclusive lock is acquired on the table . If the lock cannot be acquired, the transaction is completed and the existing locks are released. If the lock is acquired, it is propagated to the session in which the vacuum is being performed (by calling LockRelationIdForSession (..)), so that the TOAST table is vacuumed in a separate transaction and does not have to wait for the lock to be acquired.

TOAST tables will not be analyzed, since TOAST rows are always accessed via the TOAST index and TOAST statistics are useless. The session switches to work under the table owner, so that functions in the index are executed under the table owner, since configuration parameters can be set on the owner role and affect the result of the functions.

Index cleaning **can use parallel processes** , but one index is vacuumed only **by one** worker process. A table (table sections) is always scanned by one process, in which vacuum is performed . If parallel index cleaning is planned (for each table separately), then shared memory (dynamic shared memory) is allocated to store row identifiers (TID , each 6 bytes) that are already marked in blocks with the LP_DEAD hint bit (this is done by HOT) and will be marked by vacuum in the first phase of its execution . If not planned, then local memory of the process that vacuums the table. Using shared memory does not affect performance, since locks are not needed: only the main vacuuming process writes to it. Shared memory **can use HugePages** .

Temporary objects are not processed by autovacuum , but can be processed by the server process using the ANALYZE and VACUUM commands. Temporary tables (and indexes on them) are cleared sequentially and cannot be cleared in parallel , since only the server process has access to temporary tables.

There are five phases of vacuuming each table, mwiew , toast and indexes on them:
SCAN_HEAP , VACUUM_INDEX , VACUUM_HEAP , INDEX_CLEANUP , VACUUM TRUNCATE .
In addition to these, there is a preparatory initialization phase and a final phase.

First phase of vacuuming

- memory in the size of `autovacuum_work_mem is allocated for accumulation of TID of dead lines (LP_DEAD)`
- in version 17, if there is no index on a table, the space in the blocks is cleared in one pass, not in two passes, each block is changed once, not twice, and fewer log records are generated
- in the first phase, blocks are read, LP_DEAD bits are set in them and the TID of all lines with this flag are saved in memory
- up to version 17, the memory for TID is limited to 1GB, from version 17, the entire `autovacuum_work_mem can be used` and the memory is used in ~ 20 times less
- if the blocks with LP_DEAD are less than 2% of all the blocks in the table and the memory used for TID is less than 32MB, the vacuum finishes processing the table and the indexes are not vacuumed

First phase of vacuuming

First phase: memory equal to `autovacuum_work_mem` (if it is set to -1, then `maintenance_work_mem`) is allocated for accumulation of dead line TIDs (LP_DEAD) .

Table blocks that may contain dead rows are scanned (the blocks have changed since the previous vacuum, checked against the visibility map), unless the VACUUM (DISABLE_PAGE_SKIPPING) option is set . Autovacuum does not have such a parameter. Rows that have gone beyond the database horizon are marked as dead and their TIDs (along with those already marked in the block earlier as HOT) are saved in memory.

If there is not enough memory, the first phase is suspended, there is a transition to the second phase, then a return to the first phase, and in the first phase, blocks continue to be scanned further.

In this case, the second phase will be performed completely - scan all indexes again. Therefore, **it is worth setting the memory volume for vacuum or adjusting the vacuum frequency so that the vacuum is performed in one pass** . Up to version 17, if `maintenance_work_mem` or `autovacuum_work_mem` were larger than 1GB, then 1GB was used to store TIDs , TIDs were stored as a list and searching was ineffective. Starting with version 17, a prefix tree (radix tree) is used With compact storage (path compression). The memory volume for storing TIDs has been reduced by ~ 20 times .

By default, vacuuming uses the INDEX_CLEANUP AUTO option . This means that if the number of blocks with at least one dead row is less than 2% (`BYPASS_THRESHOLD_PAGES = 0.02`) of all blocks in the table and at the same time the memory under the TID (6 bytes) of dead rows is less than 32 MB (2 ^ 25 / 6 = 5 592 400 lines up to version **17**) , then the remaining phases are not executed and indexes are not scanned. The second condition is needed for large tables. Condition in version 17:

```
TidStoreMemoryUsage ( vacrel -> dead_items ) < (32L * 1024L * 1024L))
```

Without this, it would be enough to update or delete a row in the table and all indexes on the table would be scanned , which is long and labor-intensive, and the result would be nothing. There is no visibility map (and freezing) for indexes , index records are not frozen and do not have xmin , xmax . A small number of dead rows are obtained when HOT cleanup is actively running (you need to strive to make it mainly free up space). If **INDEX_CLEANUP OFF or the VACUUM_INDEX_CLEANUP parameter is set at the table level OFF , then indexes are not scanned .** This makes sense if you want to scan the table blocks as quickly as possible and mark rows as LP_DEAD , to avoid overflow of the transaction counter. Disabled by : VACUUM (INDEX_CLEANUP ON) .

Calculation of memory for TID for vacuuming

- memory in the amount of is allocated for accumulation of TID of dead lines (LP_DEAD)
  - `autovacuum_work_mem` for auto vacuum
  - `maintenance_work_mem for` VACUUM command
  - default 64GB
- memory for storing string identifiers up to version 17 is calculated using the formula: `maintenance_work_mem = n_dead_tup * 6.` But no more than 1 gigabyte.
- Example: 1000000 *6 / 1MB =5.74 = 6 passes

```
select schemaname , relname , n_dead_tup , n_live_tup from pg_stat_user_tables where
relname = 'test';
 schemaname | relname | n_dead_tup | n_live_tup
------------+---------+------------+------
public | test | 1000000 | 10000000
set maintenance_work_mem =' 1MB ';
vacuum verbose test;
INFO: vacuuming " postgres.public.test "
INFO: finished vacuuming " postgres.public.test ": index scans: 6
```


Calculation of memory for TID for vacuuming

First phase: memory equal to `autovacuum_work_mem` (if it is set to -1, then `maintenance_work_mem`) is allocated for accumulation of dead line TIDs (LP_DEAD) .

The number of dead lines determines the upper limit, that is, the maximum number of lines that can be removed during vacuuming . This number is filled with all processes of updating table rows, does not require analysis and is relevant :

```
 select schemaname , relname , n_dead_tup , n_live_tup from pg_stat_user_tables
where relname = 'test';
  schemaname | relname | n_dead_tup | n_live_tup
 ------------+---------+------------+------
 public | test | 1000000 | 10000000
```

Memory for storing string identifiers up to version 17 is calculated using the formula: `maintenance_work_mem = n_dead_tup * 6` . But no more than 1 gigabyte.

In the example: maintenance_work_mem = 1000000*6=6000000 byte.

If you install:

```
set maintenance_work_mem ='1MB';
```

then it will be 6000000 / (1024*1024) = 5.74 which gives 6 passes . Rounding up, since the passes are an integer. Example:

```
vacuum verbose test;
INFO: vacuuming " postgres.public.test "
INFO: finished vacuuming " postgres.public.test ": index scans: 6
pages: 0 removed, 48712 remain, 48712 scanned (100.00% of total)
tuples : 1000000 removed , 10000000 remain, 0 are dead but not yet removable, oldest xmin :
22113886
removable cutoff: 22113886, which was 0 XIDs old when operation ended
frozen: 4427 pages from table (9.09% of total) had 1000002 tuples frozen
index scan needed: 4425 pages from table (9.08% of total) had 1000000 dead item identifiers removed
index " test_id_idx ": pages: 30163 in total, 0 newly deleted, 0 currently deleted, 0 reusable
I/O timings: read: 1185.969 ms, write: 317.146 ms
avg read rate: 79.722 MB/s, avg write rate: 20.861 MB/s
buffer usage: 96868 hits, 186041 misses, 48681 dirtied
WAL usage: 76277 records, 54147 full page images, 429319587 bytes
system usage: CPU: user: 3.41 s, system: 3.18 s, elapsed: 18.23 s
```

In version 17, memory correlates with n_dead_tup , but there is no exact relationship. Required memory size less by ~20 times .

The second and third phases of vacuuming

- TIDs are cleared in the indexes , stored in memory in the first phase
- TIDs with LP_DEAD are cleared in table blocks , collected in memory in the first phase: slots in the block header become usable
- frozen rows that can be frozen
- maps of free space, visibility, freezing are updated
- for clearing the block buffer handle is locked in Exclusive mode
  - If an Exclusive lock on a buffer handle cannot be obtained, the TID block with LP_DEAD is not cleared.
  - the process will attempt to acquire the lock and the block will be cleared if the VACUUM command uses the SKIP_LOCKED options false or FREEZE or DISABLE_PAGE_SKIPPING , and also if the autovacuum cycle is started in aggressive mode (for freezing rows)

The second and third phases of vacuuming

PostgreSQL version 17 , **if there are no indexes on the table ,** the options are ignored , there is no second phase, the first phase is combined with the third: instead of setting LP_DEAD , the space occupied by the row (along with the mask where the LP_DEAD bit is set) is freed, and the pointer in the block header becomes UNUSED , due to this, **the volume of the record in the WAL can decrease** (https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=c120550ed).

In the second phase, the indexes are cleared of references to TIDs stored in memory in the first phase.

TIDs with LP_DEAD are cleared in table blocks , collected in memory In the first phase: the slots in the block header become usable (UNUSED). If the slots at the end of the block header are cleared, the space may become free and the block header will shrink (truncate the line pointer array) .

To clear, the block buffer handle is locked in Exclusive mode . Blocks in which in the first phase LP_DEAD was not detected and is not read.

Rows that can be frozen are frozen , fs m and vm are updated or created if they did not exist. The comments to the lazy_scan_heap () function in the vacuumlazy.c file say: prunes each page in the heap, and considers the need to freeze remaining tuples with storage (not including pages that can be skipped using the visibility map). Also performs related maintenance of the FSM and visibility map.

If it is not possible to obtain an Exclusive lock on the buffer handle, then by default a Share lock is set , the block is not cleared, but read .

If the vacuum was started in aggressive mode , then vacuum will wait for Exclusive to be received on the buffer handle.

The aggressive mode of the VACUUM command is called the use of the parameter SKIP_LOCKED FALSE or FREEZE or DISABLE_PAGE_SKIPPING .

The goal of aggressive mode is to process all blocks with the purpose of freezing. SKIP_LOCKED FALSE is intended for cases when the visibility and freezing maps cannot be trusted, there is a suspicion of their corruption , it includes FREEZE only on all blocks .

The point of skipping blocks is to speed up table vacuuming, otherwise autovacuum may not have time to process all scheduled tables. Also, if the block descriptor is locked, there is a chance that the block contents are still changing.

The fourth and fifth phases of vacuuming

- the truncation phase of the table file(s) is called if the volume of empty blocks at the end of the last file of the main layer is more than 8 MB
  - An exclusive lock on the table is requested
  - if the lock is not received within 5 seconds, the phase is skipped
  - the phase can be disabled at the table level and at the level of its TOAST table
- the analysis is performed separately from the vacuum

The fourth and fifth phases of vacuuming

The fourth phase calls the documented (https://docs.tantorlabs.ru/tdb/ru/16_4/se/index-functions.html) function of the index access method amvacuumcleanup (..), which can perform some useful actions depending on the index type. For example, free empty index pages. The output value of the function is the index statistics, which is used to output VERBOSE or to the diagnostic log. This function is also called at the end of the ANALYZE phase.

The fifth phase is called only if the volume of empty blocks at the end of the last file of the main table layer is more than 8 MB (macro REL_TRUNCATE_MINIMUM) . It is executed by the `lazy_truncate_heap ( ..)` function , which tries to get an exclusive lock on the table, but if it cannot get the lock within VACUUM_TRUNCATE_LOCK_TIMEOUT =5 seconds, then truncation is not performed. The fifth phase can be disabled at the table level and separately for a TOAST table using `the VACUUM_TRUNCATE` and `TOAST.VACUUM_TRUNCATE parameters` , as well as for the `VACUUM (TRUNCATE false) command` . Setting the `old_snapshot_threshold configuration parameter` (removed in version 17) turns off the fifth phase.

Final actions: session parameters, if changed in the body of index functions, are restored. The session is returned to the original user. The transaction is committed. The lock is removed from the main table or materialized view.

If a TOAST table was processed, the lock is not released from the main table and TOAST processing is performed with the transaction opened and committed, after which the lock is released from the main table.

In the final phase it is updated `pg_database.datfrozenxid` and truncated if possible pg_xact . This can be disabled by the SKIP_DATABASE_STATS parameter. false. **Analysis is performed separately from vacuum , even if you combine** VACUUM (ANALYZE) in one command , the analysis will be performed after vacuum.

Aggressive vacuum mode

- if a regular autovacuum can promote `pg_class.relfrozenxid` and `pg_class.relminmxid` , then an aggressive autovacuum will not be needed
- aggressive autovacuum mode is started if `age ( pg_class.relfrozenxid ) > vacuum_freeze_table_age - vacuum_freeze_min_age` or `mxid_age ( pg_class.relminmxid ) > vacuum_multixact_freeze_table_age - vacuum_freeze_min_age`
- `age( pg_database.datfrozenxid ) > autovacuum_freeze_max_age` or `mxid_age ( pg_database.datminmxid ) > autovacuum_multixact_freeze_max_age`

```
\ dconfig *freeze*
Parameter | Value
-----------------------------------+------------
 autovacuum_freeze_max_age | 10000000000
 autovacuum_multixact_freeze_max_age | 20000000000
 vacuum_freeze_min_age |   50000000
 vacuum_freeze_table_age |  150000000
 vacuum_multixact_freeze_min_age |    5000000
 vacuum_multixact_freeze_table_age |  150000000
```

```
SELECT datname , age ( datfrozenxid ) , mxid_age ( datminmxid ) FROM pg_database ;
   datname | age | mxid_age
-----------+-----------+---------
postgres | 121346629 | 19
template1 | 121346629 | 19
template0 | 121346629 | 19
```


Aggressive vacuum mode

If regular autovacuum can promote `pg_class.relfrozenxid` and `pg_class.relminmxid` , then aggressive autovacuum will not be needed. It will be able to promote if it can get locks on the handles of the blocks it wants to clean.

Aggressive mode is invoked by the SKIP_LOCKED options FALSE , FREEZE , DISABLE_PAGE_SKIPPING of the VACUUM command . All three modes freeze the rows. Autovacuum There are no options, but there is also an aggressive mode.

Aggressive autovacuum mode is activated if `age( pg_class.relfrozenxid ) > vacuum_freeze_table_age - vacuum_freeze_min_age` or if `mxid_age ( pg_class.relminmxid ) > vacuum_multixact_freeze_table_age - vacuum_freeze_min_age` .

`vacuum_freeze_min_age parameter value` is too large It is not worth installing , since the auto vacuum for freezing will be launched frequently.

These parameters with similar names **can be set at the table level** .

And the aggressive mode is launched by the database even if autovacuum is disabled on the cluster , if `age( pg_database.datfrozenxid ) > autovacuum_freeze_max_age` or **`mxid_age`** `( pg_database.datmin m xid ) > autovacuum_multixact_freeze_max_age` .

Monitoring:

```
SELECT datname , age( datfrozenxid ), mxid_age ( datmin m xid ) FROM pg_database ;
   datname | age | mxid_age
-----------+-----------+---------
postgres | 121346629 | 19
template1 | 121346629 | 19
template0 | 121346629 | 19
```

It is given how far xid is transactions and multitransactions is in the past relative to the current transaction. **On 32-bit XIDs by default `autovacuum_freeze_max_age = 200000000`, so the result should not approach 200 million. If it approaches 2 billion, then you need to look for tables with large `age ( pg_class.relfrozenxid )` and freeze them.**

If the values in `pg_class` could not change, then `age(..)> vacuum_failsafe_age` autovacuum starts in a special mode: delays **`autovacuum_vacuum_cost_delay`** do not apply, **autovacuum is not limited by the buffer ring and uses all buffer cache blocks** , indexes are not processed (analog INDEX_CLEANUP off) .

VACUUM FULL automatically freezes all lines.

Freezing rows (FREEZE)

- You can vacuum and freeze TOAST separately:

```
select relname , relfrozenxid , age( relfrozenxid ), relminmxid , mxid_ age (
relminmxid ) from pg_class where relfrozenxid <>0 order by 3 desc ;

vacuum (freeze, verbose) pg_toast.pg_toast_1262 ;
INFO: finished vacuuming "postgres.pg_toast.pg_toast_1262": index scans: 0
tuples : 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin : 105637454
removable cutoff: 105637454, which was 3 XIDs old when operation ended
new relfrozenxid : 105637454, which is 105636725 XIDs ahead of previous value
new relminmxid : 37, which is 36 MXIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
```

- Regular vacuum if possible (if it processes all the blocks that it has not processed before and in total all the lines in all the blocks are frozen) performs freezing :

```
relname | relfrozenxid | relminmxid
---------+--------------+-----------
t3 | 67551 | 37
vacuum (verbose) t3;
 relname | relfrozenxid | relminmxid
---------+--------------+-----------
t3 | 105738379 | 37
```


Freezing rows (FREEZE)

It is not recommended to disable autovacuum during heavy load and to start vacuum during low load. You can execute VACUUM (FREEZE) manually on clusters with a 32-bit transaction counter when the moment of calling autovacuum in this mode is approaching, to avoid execution during heavy load. In this mode, the vacuuming process waits for an exclusive lock not only on blocks that have not been frozen for a long time, but also on actively used blocks, since such blocks may contain a string with an old XID.

When loading rows into a table with the COPY command **, it makes sense to use the FREEZE option** . If you don't use it, the first vacuum will most likely will send full block images to WAL if a checkpoint occurs between COPY and vacuum. The documentation disclaimer about " visibility violations " and " potential problems " can be ignored. The lines in TOAST for COPY WITH FREEZE are not frozen in the current version of PostgreSQL .

Example of vacuuming with freezing TOAST tables:

```
select relname , relfrozenxid , age( relfrozenxid ), relminmxid , mxid_ age (
relminmxid ) from pg_class where relfrozenxid <>0 and relname ='pg_toast_3394';
     relname | relfrozenxid | age | relminmxid | mxid_age
--------------+--------------+-----------+------------+---------
pg_toast_3394 | 723 | 140062561 | 1 | 19
vacuum (freeze, verbose) pg_toast.pg_toast_3394;
INFO: aggressively vacuuming "postgres.pg_toast.pg_toast_3394"
tuples : 0 removed, 0 remain, 0 are dead but not yet removable, oldest xmin : 140063284
removable cutoff: 140063284, which was 0 XIDs old when operation ended
new relfrozenxid : 140063284, which is 140062561 XIDs ahead of previous value
new relminmxid : 37, which is 36 MXIDs ahead of previous value
frozen: 0 pages from table (100.00% of total) had 0 tuples frozen
buffer usage: 22 hits, 1 misses, 1 dirtied
WAL usage: 1 records, 1 full page images, 8179 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
select relname , relfrozenxid , age( relfrozenxid ), relminmxid , mxid_ age (
relminmxid ) from pg_class where relfrozenxid <>0 and relname ='pg_toast_3394';
     relname | relfrozenxid | age | relminmxid | mxid_age
--------------+--------------+-----+------------+----------
pg_toast_3394 | 140063284 | 0 | 37 | 0
```

PostgreSQL version 17

- uses less memory

PostgreSQL version 17

In version 17, the LP_DEAD row TID is more efficient both in terms of memory usage and in searching for TIDs when scanning indexes in the second phase. If there is not enough memory, then several passes are used, and this is a sharp degradation of performance. When cleaning blocks of a test table of 3.4 GB in size and the default instance settings, maintenance_work_mem = 64 MB :

```
CREATE TABLE test AS SELECT * FROM generate_series (1, 100000000) x(id);
CREATE INDEX ON test(id);
UPDATE test SET id = id - 1;
```

vacuum up to version 17 will perform cleaning in 9 passes (64 MB of memory) and in 770 seconds:

```
LOG: automatic vacuum of table " postgres.public.test ": index scans: 9
WAL usage: 2316372 records, 1189127 full page images , 2689898432 bytes
system usage: CPU: user: 78.21 s, system: 30.06 s, elapsed: 773.23 s
```

in version 17 in 1 pass , 37MB and 620 seconds (20% faster) :

```
WAL usage: 2316363 records, 1431435 full page images , 2586981769 bytes
system usage: CPU: user: 78.84 s, system: 47.42 s, elapsed: 619.04 s
```

The vacuum runtime is the database horizon hold time.

With more memory in version 16, it would be possible to get 1 pass and less time. After " checkpoint settings " , i.e. increasing `checkpoint_timeout` and `max_wal_size` number of full page images will be the same. It is worth **analyzing the log activity of vacuuming only after setting the frequency of checkpoints.**

The number of passes is indicated in the **index_vacuum_count column.** pg_stat_progress_vacuum views :

```
select * from pg_stat_progress_vacuum ;
-[ RECORD 1 ]------+---------------
pid | 559
datid | 5
datname | postgres
relid | 16422
phase | vacuuming heap
heap_blks_total | 884956
heap_blks_scanned | 884956
heap_blks_vacuumed | 441819
index_vacuum_count | 9
max_dead_tuples | 11184809
num_dead_tuples | 10522080
```

https://pganalyze.com/blog/5mins-postgres-17-faster-vacuum-adaptive-radix-trees

Comparative testing of vacuum 16 and 17 versions of PostgreSQL

- ~20 times less memory
- memory for creating radix tree is limited by `autovacuum_work_mem` or `maintenance_work_mem`
- WAL usage is the same in both versions.
- radix construction and scanning tree (instead of array) and the duration of vacuuming in version 17 more by ~20%
- there is no increase in the load on the input/output, only on the processor
  - on avg read / write rate in vacuum statistics affects CPU usage
- when enabled `( data_checksums=on OR wal_log_hints=on ) AND ( full_page_writes=on )` the volume of log entries increases because if a block changes, then once after the checkpoint an image of this block is written to the log (unused space is not recorded)
  - the less frequent the checkpoints, the less likely it is that the entire block image will be written multiple times

Comparative testing of vacuum 16 and 17 versions of PostgreSQL

Example test:

```
alter system set max_wal_size = '8GB';
alter system set checkpoint_timeout ='30min';
select pg_reload_conf ();
drop table 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;
```

Comparison of versions 16 and 17 with 1 pass through indexes:

Version 17:

```
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: 752, which was 0 XIDs old when operation ended
new relfrozenxid : 752, 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: 74.961 MB/s, avg write rate: 81.952 MB/s
buffer usage: 106583 hits, 169651 misses, 185473 dirtied
WAL usage: 364397 records, 143159 full page images, 1005717614 bytes
system usage: CPU: user: 10.84 s, system: 1.00 s, elapsed: 17.68 s
```

Version 16:

```
avg read rate: 90.693 MB/s, avg write rate: 99.367 MB/s
WAL usage: 364398 records, 143159 full page images, 1005540671 bytes
system usage: CPU: user: 10.77 s, system: 1.78 s, elapsed: 14.57 s
```

Comparison of versions 16 and 17 with 1 pass through indexes without checksums and without `wal_log_hints` :

```
buffer usage: 107081 hits, 169152 misses, 185287 dirtied
WAL usage: 231640 records, 98897 full page images, 277373985 bytes
```

Version 16: `system usage: CPU: user: 10.01 s, system: 1.03 s, elapsed: 12.29 s`

Version 17: `system usage: CPU: user: 10.18 s, system: 0.70 s, elapsed: 14.72 s`

Without checksums, the volume of log records **is exactly** 3 times less .

with checksums with reading the table with the command before checkpoint : **`explain (analyze) select * from test;`**

```
buffer usage: 121140 hits, 155093 misses, 143159 dirtied
WAL usage: 187405 records, 143159 full page images, 576514722 bytes
```

Version 16:

```
avg read rate: 108.505 MB/s, avg write rate: 100.156 MB/s
system usage: CPU: user: 8.81 s, system: 1.03 s, elapsed: 11.16 s
```

Version 17:

```
avg read rate: 81.625 MB/s, avg write rate: 75.001 MB/s
system usage: CPU: user: 10.50 s, system: 0.66 s, elapsed: 14.91 s
```

without checksums with table reading (explain is used for reading) before checkpoint :

```
buffer usage: 120474 hits, 155760 misses, 143159 dirtied
WAL usage: 187391 records, 98897 full page images, 215117884 bytes
```

Version 16:

```
avg read rate: 118.284 MB/s, avg write rate: 109.183 MB/s
system usage: CPU: user: 8.62 s, system: 0.74 s, elapsed: 10.24 s
```

Version 17:

```
avg read rate: 93.486 MB/s, avg write rate: 85.923 MB/s
system usage: CPU: user: 10.24 s, system: 0.50 s, elapsed: 13.01 s
```

Without checksums, the volume of log records **is exactly** 2 times less .

Conclusions:

1) WAL usage is the same in both versions.

2) CPU usage for building and scanning radix tree instead of array and vacuuming duration in version 17 increased by **~20 %** . `avg read/write rate` correlates with `CPU usage` .

3) When enabled `( data_checksums =on OR wal_log_hints =on) AND ( full_page_writes =on)`

the volume of journal entries increases over due to the fact that when the hint bit or checksum changes (if the bit changes, then the checksum also changes) the block changes. If a block changes, then **once** after a checkpoint an image of this block is written to the journal (except for unused space). The less frequent the checkpoints, the less likely it is that the full block image will be written multiple times.

In the given example, the table was created, changed and vacuumed . In real work, the blocks stop changing after some time and checksums with full page writes do not affect the volume of logs as much as in artificial tests. Parameters `data_checksums =on` and `full_page_writes =on` there is no need to turn it off.

With checksums enabled and full page writes **repeated** block changes after checkpoint generate **100 megabytes** instead of 1 gigabyte and **31** full page images instead of `143159 full page images` :

```
checkpoint; vacuum verbose test;
WAL usage: 364398 records, 143159 full page images, 1005 540671 bytes
delete from test; vacuum verbose test;
WAL usage: 296887 records, 204 full page images, 80 286553 bytes
insert into test SELECT * FROM generate_series (1, 10000000) x(id);
UPDATE test SET id = id - 1;
vacuum verbose test;
avg read rate: 126.019 MB/s, avg write rate: 96.746 MB/s
buffer usage: 161648 hits, 224141 misses, 172076 dirtied
WAL usage: 231668 records, 31 full page images , 100 307466 bytes
system usage: CPU: user: 10.54 s, system: 1.91 s, elapsed: 13.89 s
```

With full_page_writes unused space in block (`lp_upper - lp_lower`) is not written to the log. If the block is completely filled, then there is no unused space in it. In artificial tests, blocks are completely filled in accordance with `fillfactor = 100%` .

Checksums and WAL

- the complexity of calculating and checking checksums is insignificant
- Enabling checksums " decreases " performance because:
  - if the expression `( data_checksums =on OR wal_log_hints =on) AND ( full_page_writes =on)` is true, then any change in the contents of a block between checkpoints will write the block to WAL in full (8Kb minus unallocated space), but only once
  - when checksums are enabled, the block from the buffer is copied to the local memory of the process and from there to the linux page cache ; when it is copied from the buffer cache and there is no cost for copying the block to local memory
- by default `wal_log_hints =off` , `full_page_writes =on` and there is no need to change the values
- the host's performance is " reduced " by the instance's operation itself, but this does not mean that the instance needs to be stopped

Checksums and WAL

It is incorrect to think that checksums reduce performance because their calculation and verification consume processor resources . The overhead costs are negligible. For example, for WAL blocks , checksumming is always enabled. Enabling checksums on data blocks reduces performance because:

1) Changing hint bits and any bit in a block changes the block checksum. The block image must be written once after the checkpoint to the WAL according to `full_page_writes =on` . The increase in WAL size is significant compared to disabled checksums , but in real applications the increase is not as high compared to synthetic tests.

2) The block is located in shared memory - the instance buffer cache buffer. When checksums are enabled, the block is copied from the buffer to the local memory of the process, which calculates and inserts the checksum into the header of the block image in the local memory. From the local memory, the image is sent to the operating system for writing - to its page (4Kb) cache. When checksums are disabled, the block image is sent from the instance buffer cache buffer without copying to the local memory of the process.

`wal_log_hints` configuration parameter is notable in that its value does not need to be changed, it is useful for the text of its description in the documentation:

1) The parameter can show how much the WAL volume will increase when checksums are enabled. Enabling checksums is a relatively long operation (reading and writing each cluster block), and changing the parameter is only a restart of the instance

2) When this parameter is enabled, the entire contents of each block are written to WAL **at the first** change of the block after the checkpoint, even if hint bits are changed. For **any subsequent** changes, the full block image is not written, only changes corresponding to the size of these changes are written to the log. Are hint bit changes written ? They are not written as a separate log entry " change bit in block " , because hint bits are calculated when log entries are applied to a block. Duplication would generate non-atomicity and the possibility of inconsistency. Non-atomicity is still present for some bits, which are derivatives of changes in blocks of PGDATA/ pg_xact , PGDATA/ pg_multixact files . And inconsistency ? The PostgreSQL program code eliminates it by the order of applying changes. Attempts are successful if there is no damage. For example, if a WAL record with a block image is overlaid, but the file does not exist (deleted, not reserved), then the file may be created empty.

VACUUM command parameters

- `DISABLE_PAGE_SKIPPING` processes all table blocks without exception
- `SKIP_LOCKED false` - does not allow skipping locked objects, table sections, blocks
- `INDEX_CLEANUP auto/on/off` specifies whether to process indexes. OFF is used if you need to remove dead rows from table blocks faster
- `PROCESS_TOAST false` - disables processing of TOAST tables
- `TRUNCATE false` - disables the fifth phase
- `PARALLEL` *n* . The number *n* limits the number of background processes.
- `FULL` – full cleanup, uses exclusive locks, sequentially set on each processed table

VACUUM command parameters

Vacuum can be called manually, it will be executed by the server process. The execution algorithm is the same as autovacuum and the program code is the same, only the command can be passed execution options. It makes sense to execute the VACUUM command after creating tables or loading data . Parameters:

`DISABLE_PAGE_SKIPPING` processes all table blocks without exception. If blocks are locked, waits for a lock to be acquired. Includes the FREEZE option.

`SKIP_LOCKED false` - does not allow skipping locked objects, table sections, blocks

`INDEX_CLEANUP auto/on/off` specifies whether to process indexes. OFF is used when approaching wrap around , when dead rows need to be removed from table blocks faster .

`PROCESS_TOAST false` - disables processing of TOAST tables

`PROCESS_MAIN false` - disables table processing and handles TOAST

`TRUNCATE false` - disables the fifth phase. In this phase, an exclusive lock is set. If the wait is longer than 5 seconds for each table, the phase is skipped. When queuing, an exclusive lock makes all commands that want to work with the table wait. You can set the vacuum_truncate off parameter at the table level.

`PARALLEL` *n* . The number *n* limits the number of background processes. They are also limited by the value of the max_parallel_maintenance_workers parameter . Parallel processes are used if the index size exceeds min_parallel_index_scan_size and there is more than one such index. Does not affect the analysis, only the index processing phase.

`FULL` - full cleanup, uses exclusive locks acquired sequentially on each table processed. Requires additional disk space because new files are created and old files are not deleted until the end of the transaction. It may be worth using the `CLUSTER command` , as it is the same but orders the rows.

VACUUM command parameters

- SKIP_DATABASE_STATS disables updating pg_database.datfrozenxid
  - allows you to avoid performing a full scan of the pg_class table
- VERBOSE - displays command execution statistics
- FREEZE - freezes rows in all blocks except those in which all rows are current and frozen
- BUFFER_USAGE_LIMIT buffer ring size instead of vacuum_buffer_usage_limit
  - Unlike the configuration parameter, BUFFER_USAGE_LIMIT can be set to zero and the buffer ring will not be used.

VACUUM Command Parameters (continued)

SKIP_DATABASE_STATS disables updating of pg_database.datfrozenxid number - oldest not frozen XID in database objects. To get the value, query relfrozenxid , relminmxid from pg_class using full scan (there is no index on these columns and it is not needed) . If the size pg_class is large , then the query wastes resources. You can disable this and leave it for any VACUUM on any table, for example, once a day, or use:

VACUUM (ONLY_DATABASE_STATS VERBOSE) which will not clear anything, but will only update the value of pg_database.datfrozenxid .

VERBOSE - displays command execution statistics. Does not add additional load, recommended for use.

ANALYZE - updates statistics. The update is performed separately. Combining vacuuming and analysis in one command does not provide any performance advantages.

FREEZE - freezes rows in all blocks except those in which all rows are current and frozen. This is called "aggressive" mode. Adding the FREEZE hint is equivalent to running the VACUUM command with the vacuum_freeze_min_age=0 and vacuum_freeze_table_age=0 parameters . In FULL mode, using FREEZE is redundant since FULL also freezes rows .

BUFFER_USAGE_LIMIT buffer ring size instead of vacuum_buffer_usage_limit configuration parameter (range from 128 KB to 16 MB, default 256 MB). Unlike the configuration parameter, BUFFER_USAGE_LIMIT can be set to zero . In this case, the buffer ring is not used 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 and, if the buffer cache is large, will load the processed blocks into it. Command example:

```
VACUUM( ANALYZE , BUFFER_USAGE_LIMIT 0 );
```

If autovacuum is started to protect against transaction counter overflow, the buffer ring is not used and autovacuum is performed in aggressive mode.

Extension `pg_visibility`

- standard extension
- for each table block in the visibility map (layer `_vm`) two bits are stored:
  - `all_visible` all rows in the block are relevant and visible to all transactions
  - `all_frozen` all rows in the block are frozen
- examples of functions:

```
select * from pg_visibility_map_summary (' pg_class ');
 all_visible | all_frozen
-------------+------------
14 | 12
select * from pg_visibility ('pg_class',0);
 all_visible | all_frozen | pd_all_visible
-------------+-----------+----------------
f | f | f
select * from pg_visibility_map (' pg_class ');
 blkno | all_visible | all_frozen
-------+-------------+------------
0 | f | f
1 | f | f
...
```


Extension `pg_visibility`

Standard extension. Allows you to view the visibility map and recreate it. For each table block in the visibility map (layer `_vm`) two bits are stored (BITS_PER_HEAPBLOCK = 2). One visibility map block stores data about 32672 table blocks.

1) all_visible all rows in the block are relevant and visible to all transactions, both current and future. **The block header contains a hint bit PD_ALL_VISIBLE, it matches the bit in the visibility map** . When restoring from WAL , it may happen that the bit is not set in the visibility map, but is set in the block.

2) all_frozen all rows in the block are frozen until the block is added, modified, deleted or locked (xmax is changed) line, vacuum treatment of the block is not necessary

The function shows the summary information, useful to determine how many blocks the vacuum will read:

```
select * from pg_visibility_map_summary (' pg_class ');
 all_visible | all_frozen
-------------+------------
14 | 12
```

You can also check the effect of autovacuum - whether the rows in the blocks were frozen or completely cleared.

Information on the first block:

```
select * from pg_visibility ('pg_class',0);
 all_visible | all_frozen | pd_all_visible
-------------+-----------+----------------
f | f | f
```

For all blocks:

```
select * from pg_visibility_map (' pg_class ');
 blkno | all_visible | all_frozen
-------+-------------+------------
0 | f | f
1 | f | f
...
```

Re-creating the map (resetting) visibility:

```
select pg_truncate_visibility_map (' pg_class ');
```

The card will be filled during the first vacuum .

The functions `pg_check_visible ( ' pg_class ' )` and `pg_check_frozen ( ' pg_class ' )` check if the visibility map is corrupted. If there is no corruption, they do not return anything.

Auto vacuum monitoring

- The following may indicate that the autovacuum is not coping with the current settings:
  - tables with dead lines that significantly exceed the threshold of the automatic vacuum ;
  - long cleaning of some tables ;
  - warnings in log
- **pg_stat_progress_vacuum , pg_stat_activity , pg_stat_all_tables , pg_class** are used for monitoring

```
relation |dead(%)| reltuples|n_dead_tup | effective_settings | last_vacuumed | status |..
---------+-------+---------+---------+----+-----------------+--------+--
pgbench_ | 19400 | 1 | 194 | vt : 50, vsf : 0.2, | 2024-10-07 12:0 | queued |
branches | | | | DISABLED | 2:33 (auto) | |
pgbench_ | 1160 | 10 | 116 | vt : 50, vsf : 0.2, | 2024-10-07 12:0 | queued |
tellers | | | | DISABLED | 2:33 (auto) | |

pid | duration | waiting | mode | database | table | phase | table_size | total_size | scanned | vacuumed
----+----------+---------+------+----------+-------+-------+------------+------------+---------+--------
123 01:00:00.0        f regular postgres test vacuuming indexes 1 GB 2 GB 500 MB 400 MB
```


Auto vacuum monitoring

Autovacuum frees up space for rows that are not visible in any snapshot (the same as going beyond the database horizon) and are not held back by physical replicas. Autovacuum processes tables, materialized views, table sections, system catalog tables. Autovacuum does not process temporary tables, but they can be vacuumed and analyzed by the VACUUM, ANALYZE commands in the session where the temporary table was created.

How to find out if autovacuum needs to be adjusted ? The following may indicate that autovacuum is not coping with the current settings: tables with dead rows that significantly exceed the autovacuum threshold ; long cleaning of some tables ; warnings in the log.

Example request:

```
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_size ,
pg_size_pretty ( pg_total_relation_size ( relid )) AS total_size ,
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 scanned_pct ,
round(100.0 * p.heap_blks_vacuumed / p.heap_blks_total , 1) AS vacuumed_pct ,
p.index_vacuum_count ,
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_size | total_size | scanned | vacuumed
----+----------+---------+------+----------+-------+-------+------------+------------+---------+--------
123 01:00:00.0        f regular postgres test vacuuming indexes 1 GB 2 GB 500 MB 400 MB
```

In the example , regular autovacuum has been working for 1 hour , is in the index cleaning phase, not blocked (`waiting= f`) .

https://dataegret.de/2017/10/deep-dive-into-postgres-stats-pg_stat_progress_vacuum/

A relatively simple query to show if a table needs vacuuming :

```
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_dea | v_thre |n_mod _ | a_thre|do_va|do_an|relsize|total
| | d_tup | shold |since_ | shold | cuum | alyze | |
| | | |analyze| | | | |
----------+---------------+-----+-------+-------+------+-----+-----+-------+------
pg_catalog|pg_statistic | 84 | 140.8 |150927 | 95.4 | no | yes |352256 |589824
public | pgbench_branches | 210 | 50.2 | 615 | 50.1 | yes | yes | 16384 |106496
public | pgbench_tellers | 291 | 52 | 615 | 51 | yes | yes | 73728 |466944
```

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 inaccurate .
The query shows that three tables have v_threshold higher than the default 20% autovacuum threshold

The query that takes into account table-level settings is shown on the next page and the query is quite long. This query returns a realistic list of tables that will be vacuumed in the next autovacuum cycle. An example of the result of this query:

```
relation |dead(%)| reltuples|n_dead_tup | effective_settings | last_vacuumed | status |..
---------+-------+---------+---------+----+-----------------+--------+--
pgbench_ | 19400 | 1 | 194 | vt : 50, vsf : 0.2, | 2024-10-07 12:0 | queued |
branches | | | | DISABLED | 2:33 (auto) | |
pgbench_ | 1160 | 10 | 116 | vt : 50, vsf : 0.2, | 2024-10-07 12:0 | queued |
tellers | | | | DISABLED | 2:33 (auto) | |
```

dead(%) = 19400% indicates that the estimated number of dead lines that can be vacuumed significantly exceeds the default autovacuum threshold of 20%. Last vacuum time `last_vacuumed` shows whether the table was actively changed (if it was recently vacuumed) or the problem is that autovacuum cannot cope: it could not clear the table (due to constant locks on it) or it cannot process all tables .
https://gitlab.com/-/snippets/1889668

```
with table_opts as (
select pg_class.oid, relname , nspname , array_to_string ( reloptions , '') as relopts
from pg_class join pg_namespace ns on relnamespace = ns.oid
), vacuum_settings as (
  select oid , relname , nspname ,
case
when relopts like '% autovacuum_vacuum_threshold %' then
        regexp_replace ( relopts , '.* autovacuum_vacuum_threshold =([0-9.]+).*', e'\\1')::int8
else current_setting (' autovacuum_vacuum_threshold ')::int8
end as autovacuum_vacuum_threshold ,
case
when relopts like '% autovacuum_vacuum_scale_factor %'
then regexp_replace ( relopts , '.* autovacuum_vacuum_scale_factor =([0-9.]+).*', e'\\1')::numeric
else current_setting (' autovacuum_vacuum_scale_factor ')::numeric
end as autovacuum_vacuum_scale_factor ,
case
when relopts ~ ' autovacuum_enabled =( false|off )' then false else true
end as autovacuum_enabled
   from table_opts
), p as (select * from pg_stat_progress_vacuum )
select
coalesce(
coalesce( nullif ( vacuum_settings.nspname , 'public') || '.', '') || vacuum_settings.relname , --this DB
format('[something in "%I"]', p.datname ) --another DB
) as a relation,
round((100 * psat.n_dead_tup ::numeric / nullif ( pg_class.reltuples , 0))::numeric, 0) as "dead(%)",
  pg_class.reltuples ::numeric, psat.n_dead_tup ,
format (
' vt : %s, vsf : %s, %s', -- ' vt ' - vacuum_threshold , ' vsf ' - vacuum_scale_factor
    vacuum_settings.autovacuum_vacuum_threshold ,
    vacuum_settings.autovacuum_vacuum_scale_factor ,
(case when autovacuum_enabled then 'DISABLED' else 'enabled' end)
) as effective_settings ,
case
when last_autovacuum > coalesce( last_vacuum , '0001-01-01')
then left( last_autovacuum ::text, 19) || '(auto)'
when last_vacuum is not null then left( last_vacuum ::text, 19) || '(manual)'
else null
end as last_vacuumed ,
coalesce( p.phase , 'queued') as status,
p.pid as pid ,
case
when a.query ~ '^autovacuum.*to prevent wraparound' then 'wraparound'
when a.query ~ '^vacuum' then 'user'
when a.pid is null then null
else 'regular'
end as mode,
case
when a.pid is null then null
else coalesce( wait_event_type || '.' || wait_event , 'f')
end as waiting,
round(100.0 * p.heap_blks_scanned / nullif ( p.heap_blks_total , 0), 1) as scanned_pct ,
round(100.0 * p.heap_blks_vacuumed / nullif ( p.heap_blks_total , 0), 1) as vacuumed_pct ,
  p.index_vacuum_count ,
case
when psat.relid is not null and p.relid is not null then
(select count(*) from pg_index where indrelid = psat.relid )
else null
end as index_count
from pg_stat_all_tables psat
join pg_class on psat.relid = pg_class.oid
left join vacuum_settings on pg_class.oid = vacuum_settings.oid
full outer join p on p.relid = psat.relid and p.datname = current_database ()
left join pg_stat_activity a using ( pid )
where
  psat.relid is null
or p.phase is not null
or ( autovacuum_vacuum_threshold + ( autovacuum_vacuum_scale_factor ::numeric * pg_class.reltuples )
< psat.n_dead_tup
)
order by status, relation;
```

Performance `pg_stat_progress_vacuum`

- contains one row for each server process executing the VACUUM command and each autovacuum worker executing a vacuum at the time the view is accessed
- phase column reflects the current vacuum phase: `initializing` (preparatory, goes quickly) , `scanning heap, vacuuming indexes, vacuuming heap, cleaning up indexes, truncating heap, performing final cleanup` (final)
- by columns `heap_blks_total , heap_blks_scanned , heap_blks_vacuumed` evaluate the progress of cleaning
- `num_dead_tuples` - the number of TIDs currently placed in the memory structure. If `max_dead_tuples is reached` , the value in **`index_vacuum_count will be increased`**
- `VACUUM` **`FULL monitored`** via `pg_stat_progress_cluster`
- `ANALYZE` monitored via `pg_stat_progress_analyze`

Performance `pg_stat_progress_vacuum`

View **pg_stat_progress_vacuum** contains one row for each server process executing the VACUUM command and each autovacuum worker executing a vacuum at the time the view is accessed.

VACUUM FULL executions are tracked through the pg_stat_progress_cluster view . VACUUM FULL is a special case of the CLUSTER command and is executed by the same code. It is optimal to use CLUSTER instead of VACUUM FULL because it creates data files with rows in sorted order.

ANALYZE command is tracked through the pg_stat_progress_analyze view .

phase column reflects the current vacuum phase: `initializing` (preparatory, goes quickly) , `scanning heap, vacuuming indexes, vacuuming heap, cleaning up indexes, truncating heap, performing final cleanup` (final).

The columns heap_blks_total , heap_blks_scanned , heap_blks_vacuumed give values in blocks. The values can be used to estimate the table size and how many blocks have already been processed (estimate the progress of the cleaning).

max_dead_tuples - An estimate of the maximum number of row identifiers (TIDs) that will fit in the memory limited by the autovacuum_work_mem or maintenance_work_mem parameter in effect for the process to which the view row belongs .

num_dead_tuples - the number of TIDs currently placed in the memory structure. If the number reaches the value at which memory is exhausted (max_dead_tuples) the index cleaning phase will begin and the value in the index_vacuum_count field will be incremented .

, you can use the pg_stat_activity view , which also displays the activity of server processes and autovacuum workers . This view is useful because it shows whether a process is waiting for something.

When switching to table vacuuming, the parameters with which the process will operate are read and the values cannot be changed until the table vacuuming is completed .

Parameter `log_autovacuum_min_duration`

- default 10 minutes
- if the autovacuum exceeds this time during processing (vacuum or analysis) of the table, then a message will be written to the cluster log
- If such messages appear, it is worth finding out the reason for the long vacuuming of the table

```
LOG: automatic vacuum of table " postgres.public.pgbench_tellers ": index scans: 0
pages: 0 removed, 9 remain, 9 scanned (100.00% of total)
tuples : 125 removed, 10 remain, 0 are dead but not yet removable, oldest xmin : 87738094
removable cutoff: 87738094, which was 2 XIDs old when operation ended
new relfrozenxid : 87738066, which is 609 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
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 42 hits, 0 misses, 0 dirtied
WAL usage: 4 records, 0 full page images, 722 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
LOG: automatic analyze of table " postgres.public.pgbench_tellers "
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 52 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```


Parameter `log_autovacuum_min_duration`

You can analyze the operation of the autovacuum with using the log. To do this, you need to enable logging of table processing for a long time. A long time indicates either that the table has not been vacuumed for a long time for some reason. If autovacuum has not been disabled, the table has not been locked for a long time (for example, by non-stop execution of ANALYZE commands , the locking mode of which is incompatible with vacuum), high threshold of the autovacuum , it could happen that the autovacuum was busy vacuuming other tables and could not get to this table. **You cannot turn off the autovacuum, the longer the time it is turned off, the longer it will process the accumulated dead lines** when it is started.

To monitor the occurrence of long-term vacuuming of objects, the parameter can be used: `log_autovacuum_min_duration` , by default set to 10 minutes. If autovacuum exceeds this time while processing a table , a message will be written to the cluster log. If such messages appear, it is worth finding out the reason for the long vacuuming of the table.

Enable collector if messages are output to linux log or another inconvenient place:

```
alter system set logging_collector = on;
sudo systemctl restart tantor-se-server-16.service
```

Setting the table processing duration threshold, longer than which messages should be output to the log. Zero outputs all messages for all tables ; if there are a large number of tables, there will be a lot of messages .

```
alter system set log_autovacuum_min_duration = 0 ;
select pg_reload_conf ();
```

Autovacuum messages:

```
LOG: automatic vacuum of table " postgres.public.pgbench_tellers ": index scans: 0
pages: 0 removed, 9 remain, 9 scanned (100.00% of total)
...
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
LOG: automatic analysis of table " postgres.public.pgbench_tellers "
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 52 hits, 0 misses, 0 dirtied
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```

Autovacuum configuration parameters

- List of parameters:

```
select name, setting||coalesce(unit,'') unit, context, min_val , max_val , short_desc from
pg_settings where category=' Autovacuum ' or name like '% autovacuum %' order by 2 desc ;
name | unit | context | min_val | max_val
-------------------------------------+----------+----------+-------+---------------
autovacuum | on | sighup | |
autovacuum_naptime | 60 s | sighup | 1 | 2147483
log_autovacuum_min_duration | 600000 ms | sighup | -1 | 2147483647
autovacuum_analyze_threshold | 50 | sighup | 0 | 2147483647
autovacuum_vacuum_threshold | 50 | sighup | 0 | 2147483647
autovacuum_max_workers | 3 |postmaster| 1 | 262143
autovacuum_vacuum_cost_delay | 2 ms | sighup | -1 | 100
autovacuum_multixact_freeze_max_age |20000000000|postmaster| 10000 | 9223372036854775807
autovacuum_work_mem | -1 kB | sighup | -1 | 2147483647
autovacuum_freeze_max_age |10000000000|postmaster| 100000| 9223372036854775807
autovacuum_vacuum_insert_threshold | 1000 | sighup | -1 | 2147483647
autovacuum_vacuum_cost_limit | -1 | sighup | -1 | 10000
autovacuum_vacuum_insert_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_vacuum_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_analyze_scale_factor | 0.1 | sighup | 0 | 100
(15 rows)
```


Autovacuum configuration parameters

List of parameters: **`select name, setting||coalesce(unit,'') unit, context, min_val , max_val , short_desc from pg_settings where category=' Autovacuum ' or name like '% autovacuum %' order by 2 desc ;`**

```
name | unit | context | min_val | max_val
-------------------------------------+----------+----------+-------+---------------
autovacuum | on | sighup | |
autovacuum_naptime | 60s | sighup | 1 | 2147483
log_autovacuum_min_duration | 600000ms | sighup | -1 | 2147483647
autovacuum_analyze_threshold | 50 | sighup | 0 | 2147483647
autovacuum_vacuum_threshold | 50 | sighup | 0 | 2147483647
autovacuum_max_workers | 3 |postmaster| 1 | 262143
autovacuum_vacuum_cost_delay | 2ms | sighup | -1 | 100
autovacuum_multixact_freeze_max_age |20000000000|postmaster| 10000 | 9223372036854775807
autovacuum_work_mem | -1kB | sighup | -1 | 2147483647
autovacuum_freeze_max_age |10000000000|postmaster| 100000| 9223372036854775807
autovacuum_vacuum_insert_threshold | 1000 | sighup | -1 | 2147483647
autovacuum_vacuum_cost_limit | -1 | sighup | -1 | 10000
autovacuum_vacuum_insert_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_vacuum_scale_factor | 0.2 | sighup | 0 | 100
autovacuum_analyze_scale_factor | 0.1 | sighup | 0 | 100
(15 rows)
```

Configuration parameter values `autovacuum = on` and `track_counts = on` there is no need to change it, otherwise the auto vacuum will turn off.

`autovacuum_max_workers` specifies the maximum number of autovacuum background worker processes that can run simultaneously. on the instance. Since autovacuum processes do not wait for locks to be released by default (they do not wait or wait up to 5 seconds), they work quite actively and their number should not exceed the number of processor cores that you want to allocate for autovacuum processes. At the same time, if all autovacuum processes linger on tables for a long time, other cluster tables will not be cleared until a free worker process appears.

The duration of the autovacuum operation on one table depends on the number of leaf blocks of all indexes (for indexes of the btree type) on the table ; the number of table blocks to which changes have been made since the previous vacuuming (as a consequence, the frequency of vacuuming of this table) ; index_vacuum_count (number of index scan iterations). For large tables, a threshold of 2% of blocks with at least one dead row , below which indexes are not cleaned, reduces the problem of autovacuum latency on a large table.

https://docs.tantorlabs.ru/tdb/ru/16_4/se/routine-vacuuming.html

Setting up autovacuum

- to minimize autovacuum downtime `autovacuum_naptime` **should be set to the minimum value: 1 second**
- `log_autovacuum_min_duration` c it is necessary to select a value (similar to `checkpoint_warnings`) at which messages in the cluster diagnostic log will not be output frequently
- `autovacuum_vacuum_cost_limit` by default it is -1, which means it is equal to vacuum_cost_limit , which by default is 200
  - shared among all autovacuum workers
  - when the limit is reached, each autovacuum worker process goes to sleep for a time in the range: from `autovacuum_cost_delay` to its fourfold value
  - the parameter can be set at table level
- **`autovacuum_vacuum_cost_delay` should be set to zero at the cluster level and do not set at the table level**
  - for the VACUUM command `vacuum_cost_delay` defaults to zero

Setting up autovacuum

If the autovacuum is operating on a loaded instance, then to minimize downtime:

autovacuum_naptime should be set to the minimum value of 1 second . The parameter sets the minimum delay between two autovacuum runs for a separate database. If there was no activity in the database, autovacuum will not work on it.

`log_autovacuum_min_duration` defaults to 10 minutes. If autovacuum exceeds the threshold, a message will be written to the cluster log. Similar to `checkpoint_warnings` c it is necessary to select a value at which messages will not occur frequently.

`autovacuum_vacuum_cost_limit` by default -1, which means it is equal `to vacuum_cost_limit` , which by default is 200 and is divided by all autovacuum workers . When `autovacuum_max_workers is increased` it makes sense to increase the parameter value. When the limit is reached, each autovacuum worker process **goes to sleep for a time in the range from `autovacuum_cost_delay` to four times its value** . The parameter can be set at the table level. Before version 12, the default value was 10 times larger.

For the VACUUM command, vacuum_cost_delay =0 and there is no delay because if the VACUUM command is launched, you probably want it to be executed as quickly as possible. For autovacuum, there is a delay by default. The vacuum delay is harmful because the autovacuum transaction lasts longer. This means that the table lock and the database horizon are held longer. About exclusive locks, the documentation says: for "certain operations that hold critical locks" the delay is not performed.

The delay is performed by the vacuum_delay_point (void) function from vacuum.c which is called before scanning each block . In addition to falling asleep, the vacuum_delay_point (void) function calculates and updates cost, the process wait status, which takes up processor resources. A small delay is equivalent to underloading the processor cores, which is also undesirable. The calculation is not performed if `autovacuum_vacuum_cost_delay = 0` at the cluster level, and at the table level and TOAST was not set to a value greater than zero. Zeroing the delay does not result in non-stop operation of autovacuum, since the minimum value of `autovacuum_naptime` is 1 second. **It makes sense to set `autovacuum_vacuum_cost_delay = 0` at the cluster level and not set it at the table level** .

Changing the values of the parameters `autovacuum_vacuum_cost_delay` and `autovacuum_va with uum_cost_limit` act immediately even on processes that are already vacuuming tables .

autovacuum_naptime parameter

- to minimize autovacuum downtime, autovacuum_naptime should be set to the minimum value of ' 1 s'
- if autovacuum fails to lock the table, it skips it. The next attempt will be in a new autovacuum cycle
- with **a utovacuum_naptim e = '1s'** attempts every second after the end of the previous vacuum cycle :

```
22:29:07.466 [195773] LOG: statement: CREATE INDEX ON test(id);
22:29:0 7 .652 [195796] LOG: skipping vacuum of "test" --- lock not available
22:29:0 8 .653 [195797] LOG: skipping vacuum of "test" --- lock not available
22:29:0 9 .652 [195798] LOG: skipping vacuum of "test" --- lock not available
22:29:1 0 .653 [195799] LOG: skipping vacuum of "test" --- lock not available
```

- With **autovacuum_naptim e = '3s'** attempts are made once every

```
22:33:47.021 [195773] LOG: statement: CREATE INDEX ON test(id);
22:33:4 9 .366 [195978] LOG: skipping vacuum of "test" --- lock not available
22:33:5 2 .373 [195981] LOG: skipping vacuum of "test" --- lock not available
22:33:5 5 .373 [195985] LOG: skipping vacuum of "test" --- lock not available
22:33:5 8 .378 [195991] LOG: skipping vacuum of "test" --- lock not available
```


autovacuum_naptime parameter

If the autovacuum worker encounters a lock on an object, the object will not be processed in this cycle. Blocks on which autovacuum could not obtain an exclusive lock to free up space will not be cleaned until the next cycle. **The next cycle begins with autovacuum_naptim e** . The default value is 1 minute.

If the autovacuum workflow did not have time to complete the cycle (and there may be many objects) autovacuum_naptime , autovacuum process launcher will send another worker process to the same database if there is a free worker process. The second and subsequent worker processes will build their lists of tables to process and will work on their lists. The list includes tables for which the threshold values are exceeded. Workers do not coordinate lists. If a table is locked by another worker process, it is simply skipped.

Settings that the autovacuum does not have:

SKIP_LOCKED false - prevents the VACUUM command from skipping locked objects, table sections, blocks

DISABLE_PAGE_SKIPPING tells the VACUUM command to process all table blocks without exception. If the blocks are locked, the command waits until the lock is acquired.

For autovacuum, waiting for blocking is in aggressive mode. Aggressive mode is launched infrequently. Launch conditions:

if age(pg_class.relfrozenxid) > vacuum_freeze_table_age - vacuum_freeze_min_age or if

mxid_age (pg_class.relminmxid) > vacuum_multixact_freeze_table_age - vacuum_freeze_min_age .

Thus, reducing autovacuum_naptim e to a minimum (one second) and disabling delays by setting autovacuum_vacuum_cost_delay to zero increases the likelihood that autovacuum and autoanalysis will process tables in a timely manner . Increasing the value to more than a second may be optimal if autovacuum_max_workers is set to a large value.

With multi-transaction IDs you need to use the mxid_age (..) function , using the age(..) function with multi-transactions will give an incorrect result.

Selection of tables by autovacuum

- the table is vacuumed if:
  - the proportion of dead rows is greater than the scale factor for vacuum: `pg_stat_all_tables.n_dead_tup / pg_class.reltuples > autovacuum_vacuum_scale_factor`
  - the proportion of inserted strings since the last vacuuming is greater than the scale factor for inserts
- statistics for the table will be updated by autovacuum if:
  - the proportion of changed lines since the last analysis is greater than the scale factor for the analysis:
  - `pg_stat_all_tables.n_mod_since_analyze / pg_class.reltuples > autovacuum_analyze_scale_factor`
- the values of the parameters `autovacuum_vacuum_threshold` , `autovacuum_vacuum_insert_threshold` , `autovacuum_analyze_scale_factor` can be ignored, since tables with a small number of rows rarely require attention

Selection of tables by autovacuum

Formulas that determine whether the autovacuum will vacuum the table.

Condition of changed lines:

`pg_stat_all_tables.n_dead_tup > autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor * pg_class.reltuples`

Condition of inserted rows:

`( pg_stat_all_tables.n_tup_ins > 0) && ( pg_stat_all_tables.n_ins_since_vacuum > autovacuum_vacuum_insert_threshold + autovacuum_vacuum_insert_scale_factor * pg_class.reltuples )`

Will the table be analyzed in the autoanalysis cycle :

`pg_stat_all_tables.n_mod_since_analyze > autovacuum_analyze_threshold + autovacuum_analyze_scale_factor * pg_class.reltuples`

`autovacuum_vacuum_threshold` And `autovacuum_analyze_threshold` default 50 lines for substitution into the formula. If there are many rows in the table and _ scale_factor is not equal to zero, then weakly affect the result.

`autovacuum_vacuum_insert_threshold` = 1000 , the number of rows inserted.

`autovacuum_vacuum_insert_scale_factor` the default is 0.2 , that is, **20 % of the table's rows** . Vacuuming performs freezing. The parameter was added in PostgreSQL version 13 based on the fact that in some tables rows were only inserted and autovacuum did not process them. and when the time for freezing rows approached on the 32-bit XID counter, autovacuum had to process a large number of rows, which was slow and due to locks, not all blocks could be processed quickly. Therefore, freezing of rows in blocks not processed by freezing should also be performed periodically.

`autovacuum_analyze_scale_factor` default 0.1 (10%)

ANALYZE command sets a lock that is incompatible with vacuum, and it is not worth analyzing the table continuously. Autoanalysis does not interfere with autovacuum , since it is performed after autoanalysis .

`autovacuum_vacuum_scale_factor` by default 0.2 (**20 % of table rows**) This is the main parameter that determines whether the table will be vacuumed . If we neglect the absolute number of rows (`autovacuum_vacuum_threshold`) , then the formula is reduced to condition:

the table is vacuumed if the proportion of dead rows is greater than the scale factor

(**`pg_stat_all_tables.n_dead_tup`** / `pg_class.reltuples > autovacuum_vacuum_scale_factor`)

Recommendations for setting up autovacuum

- the ability to purge old row versions in blocks is determined by the database horizon
- if the horizon does not move for a long time, then neither HOT cleanup nor autovacuum will be able to clear the space in blocks from old versions of lines
- it is better to reduce delays in autovacuum process cycles than to increase the number of autovacuum work processes
- watch " `index scans:` " in the autovacuum log:

```
INFO: finished vacuuming " postgres.public.test ": index scans: 29
```

  - if the value is greater than 1, then either increase autovacuum_work_mem or the autovacuum operation frequency according to the table
- monitor the duration of the autovacuum operation:

```
system usage: CPU: user: 8.90 s, system: 1.04 s, elapsed: 29.53 s
```

  - the database horizon is held for this time
- monitor the efficiency of the autovacuum:

```
tuples : 10000000 removed , 10000000 remain, 0 are dead but not yet removable
```


Recommendations for setting up autovacuum

The ability to clean old row versions in blocks is determined by the database horizon. If the horizon does not shift for a long time, then neither HOT cleanup nor autovacuum will be able to clean the space in the blocks from old row versions. Vacuuming works within the buffer ring and scanning table blocks can create a load on input/output. But it is impossible to conclude that it is better for autovacuum to work less often, because if the blocks contain old row versions, it means that the blocks were changed relatively recently and are in the buffer cache . On loaded instances, the buffer cache size is large. The vacuum buffer ring does not matter - a block cannot be in two buffers at the same time. If autovacuum vacuums the table less often (even without much result), then it may happen that some of the table blocks are displaced from the buffer cache and autovacuum will load them from the disk, increasing input/output. If autovacuum runs through table blocks without delay, it increases the probability of keeping them in the buffer cache until the next scan.

And if the table is large ? The vacuum will run through blocks that have irrelevant rows. What are these blocks ? In which there were changes to rows and / or these blocks were then held by readers. The set of blocks in the buffer cache that are changed or requested is called the " active set " of blocks. Autovacuum scans the active set. Autovacuum will enter the " inactive set " of blocks for freezing purposes, but their number will be small, since often when the all_visible bit is set it is possible to set the all_frozen bit as well . The exact number can be obtained by querying the visibility and freezing map:

```
select * from pg_visibility_map_summary (' table ');
 all_visible | all_frozen
-------------+------------
113 | 113
```

If the buffer cache size is larger than the " active set " , the extra I/O operations that may occur due to the autovacuum are unlikely to occur.

It is more optimal to reduce the delays in the cycles of autovacuum processes than to increase the number of autovacuum worker processes. Autovacuum processes can be assigned by the operating system scheduler to processor cores and through memory buses, without interfering with other cores reading blocks in the buffer cache .

The Importance of Monitoring the Database Horizon

- You need to monitor the database horizon to find the reasons why it is being held back
- horizon holds requests on replicas if feedback is enabled (`hot_standby_feedback =on`)
- autovacuum should be configured after selecting `checkpoint_timeout` and `shared_buffers`
- the time of database processing by autovacuum should preferably not be longer than the time to horizon
- autovacuum processes one table in one transaction, it should not hold the horizon (be the longest in time)
  - on such tables it is worth reducing `autovacuum_vacuum_scale_factor`
- You can reduce the database processing time by reducing this parameter at the cluster level
- if you can't reduce it, then increase `autovacuum_max_workers`

Database Horizon

It is necessary to monitor the database horizon in order to find the reasons why it is held or not shifted for a long time.

Cluster database horizon in the number of transaction numbers separated from the current one:

```
select datname , greatest(max(age( backend_xmin )), max(age( backend_xid ))) from pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname order by datname ;
```

The duration of the longest query or transaction that holds the horizon :

```
select datname , extract(epoch from max( clock_timestamp ()- xact_start )) from pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname order by datname ;
```

Horizon hold (held on all bases) by physical replication slots if feedback is enabled (`hot_standby_feedback =on`) :

```
select max(age( xmin )) from pg_replication_slots ;
select backend_xmin , application_name from pg_stat_replication order by age( backend_xmin ) desc ;
```

In the replicas themselves, you can search for processes executing commands that maintain the horizon in the same way as on the master - by querying `pg_stat_activity` :

```
select backend_xmin , backend_xid , pid , datname , 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 ;
```

Autovacuum should be configured after selecting `checkpoint_timeout` and `shared_buffres` . The time it takes to process a database with autovacuum should not be very long (not much longer than the time to the horizon). These intervals can be estimated by the time of the last table processing or by the table being mentioned in autovacuum messages in the cluster log. The second thing you can pay attention to is the tables that are processed in one autovacuum cycle the longest. The database horizon is held for the time it takes to process one table. That is why you should not use `autovacuum_vacuum_cost_delay` . If it is autovacuum that holds the horizon, and not server processes, then you should reduce `autovacuum_vacuum_scale_factor` on such tables . At the cluster level, this parameter should be reduced (not increased) to reduce the database processing time to the time it takes for server processes to hold the horizon. If this fails, then increase the number of `autovacuum_max_workers` .

Monitoring the database horizon

- cluster database horizon in the number of transaction numbers separated from the current one:

```
select datname , greatest(max(age( backend_xmin )), max(age( backend_xid ))) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname
order by datname ;
```

- the duration of the longest query or transaction that holds the horizon:

```
select datname , extract(epoch from max( clock_timestamp () - xact_start )) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname
order by datname ;
```

- enabled (hot_standby_feedback=on

```
select max(age( xmin )) from pg_replication_slots ;
select backend_xmin , application_name from pg_stat_replication order by age( backend_xmin )
desc ;
```

- that maintain the horizon in the same way as on the master - by querying pg_stat_activity

```
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 ;
```


Monitoring the database horizon

It is necessary to monitor the database horizon in order to find the reasons why it is held or not shifted for a long time.

Cluster database horizon in the number of transaction numbers separated from the current one:

```
select datname , greatest(max(age( backend_xmin )), max(age( backend_xid ))) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group by
datname order by datname ;
```

The duration of the longest query or transaction that holds the horizon :

```
select datname , extract(epoch from max( clock_timestamp ()- xact_start )) from
pg_stat_activity where backend_xmin is not null or backend_xid is not null group by
datname order by datname ;
```

Horizon hold (held on all bases) by physical replication slots if feedback is enabled (`hot_standby_feedback =on`) :

```
select max(age( xmin )) from pg_replication_slots ;
select backend_xmin , application_name from pg_stat_replication order by age(
backend_xmin ) desc ;
```

In the replicas themselves, you can search for processes executing commands that maintain the horizon in the same way as on the master - by querying `pg_stat_activity` :

```
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 ;
```

Autovacuum should be configured after selecting `checkpoint_timeout` and `shared_buffres` . The time it takes to process a database with autovacuum should not be very long (not much longer than the time to the horizon). These intervals can be estimated by the time of the last table processing or by the table being mentioned in autovacuum messages in the cluster log. The second thing you can pay attention to is the tables that are processed in one autovacuum cycle the longest. The database horizon is held for the time it takes to process one table. That is why you should not use `autovacuum_vacuum_cost_delay` . If it is autovacuum that holds the horizon, and not server processes, then you should reduce `autovacuum_vacuum_scale_factor` on such tables . At the cluster level, this parameter should be reduced (not increased) to reduce the database processing time to the time it takes for server processes to hold the horizon. If this fails, then increase the number of `autovacuum_max_workers` .

Autovacuum parameters at table level

- You can determine which parameters are set by querying:

```
 select nspname , relname , reloptions from pg_class join pg_namespace ns on
relnamespace = ns.oid where reloptions is not null and relkind in (' r','t','m','p ')
order by 2;
      nspname | relname | reloptions
 -----------------+-----------------+-------------------------------------------------------
-------------
 pg_toast_temp_0 | pg_toast_16394 | { vacuum_truncate =false}
  pg_toast | pg_toast_16399 | { autovacuum_vacuum_cost_delay =23}
 pg_temp_0 | reloptions_test | { vacuum_truncate = false,autovacuum_enabled =false}
 public | test | { autovacuum_enabled =off}
```

- return default values:

```
alter table test reset ( autovacuum_enabled , fillfactor , .. | all );
```

- set values :

```
alter table test set ( fillfactor =100,..)
```


Autovacuum parameters at table level

At the table level, you can set autovacuum parameters for a table and for its TOAST table. You can determine which parameters are set by querying:

```
 select nspname , relname , reloptions from pg_class join pg_namespace ns on
relnamespace = ns.oid where reloptions is not null and relkind in ('
r','t','m','p ' ) order by 2 ;
      nspname | relname | reloptions
 -----------------+-----------------+-------------------------------------------------------------------------
 pg_toast_temp_0 | pg_toast_16394 | { vacuum_truncate =false}
  pg_toast | pg_toast_16399 | { autovacuum_vacuum_cost_delay =23}
 pg_temp_0 | reloptions_test | { vacuum_truncate = false,autovacuum_enabled =false}
 public | test | { autovacuum_enabled =off}
```

set the values using the command:

```
alter table test set (
AUTOVACUUM_ENABLED
AUTOVACUUM_ANALYZE_SCALE_FACTOR
AUTOVACUUM_VACUUM_INSERT_SCALE_FACTOR
AUTOVACUUM_VACUUM_SCALE_FACTOR
AUTOVACUUM_VACUUM_THRESHOLD
...
FILLFACTOR= N range from 10 to 100
LOG_AUTOVACUUM_MIN_DURATION
PARALLEL_WORKERS
TOAST.AUTOVACUUM* ...,
TOAST_TUPLE_TARGET
TOAST.VACUUM_INDEX_CLEANUP
TOAST.VACUUM_TRUNCATE
VACUUM_INDEX_CLEANUP
VACUUM_TRUNCATE
```

Reset to default values: `alter table test reset ( autovacuum_enabled , fillfactor , .. | all );`

Examples:
https://github.com/postgres/postgres/blob/master/src/test/regress/sql/reloptions.sql

Parameter `default_statistics_target`

- sets
  - number of most common values in table columns (`pg_stats . most_common_vals`)
  - number of bins in histograms of distribution of values in columns (`pg_stats . histogram_bounds`)
  - number of rows (`default_statistics_target * 300`) random sample for which statistics are collected
- default 100
- maximum value 10000
- can be set for a specific table column or index by expression:

```
alter table test alter column id set statistics 10000;
alter index test alter column 1 set statistics 10000;
```


Parameter `default_statistics_target`

`300* default_statistics_target` rows is used to collect statistics . The default value is 100. The maximum value is 10000. The default value is sufficient for a representative sample and sufficient accuracy. In addition, the parameter sets the number of most frequently occurring values in table columns (`pg_stats . most_common_vals`) and the number of bins in the histograms of the distribution of values in columns (`pg_stats . histogram_bounds`). If the table has many rows and the distribution of values is uneven, you can increase the value for the table column using the command:

`alter table test alter column id set statistics 10000;`

and the planner will calculate the cost more accurately.

The higher the value, the more time it will take for autoanalysis. and the volume of statistics will be larger .

A value of -1 reverts to using the `default_statistics_target parameter` . The command acquires a `SHARE UPDATE EXCLUSIVE lock on the table` .

For indexes where expressions are indexed (function-based index), the value can be set with the command:

`alter index test alter column 1 set statistics 10000;`

Since expressions do not have unique names, the ordinal number of the column in the index is specified . Range values: 0..10000; A value of -1 returns to the use of the `default_statistics_target` parameter .

A parameter value in the range from 100 to 10000 does not affect the duration of the autoanalysis cycle .

Testing the ANALYZE command on a temporary table:

https://wiki.astralinux.ru/tandocs/vliyanie-nastrojki-default_statistics_target-na-skorost-vypolneniya-komandy-analyze-294400566.html

Bloat tables and indexes

- autovacuum may not process the table due to the following:
  - the database horizon has not moved for a long time
  - at the moment the autovacuum accessed the table, a lock incompatible with the autovacuum was installed on it
- After the autovacuum has worked, the file sizes are unlikely to decrease
- blocks will be used in the future for new versions of lines
- there is no need to monitor too often, it is more important to check the free space on the disks
- an example of what the report might look like:

```
Is N/A | Table | Size | Extra |Bloat estimate| Live | Last Vaccum | Fillfactor
-------+-----------------+-----+--------------+--------------+--------+------------------+----------
| pgbench_history |2832kB|~16 kB (0.56%)|~16 kB (0.56%)|~2816 kB |  11:11:11 (auto)| 100
```

```
create extension pgstattuple ;
\ dx + pgstattuple
select relname , b.* from pg_class , pgstattuple_approx ( oid ) b WHERE relkind ='r';
select relname , b.* from pg_class , pgstatindex ( oid ) b WHERE relkind = ' i ' order
by 10;
```


Bloat tables and indexes

Old row versions are stored in table blocks. Indexes store references to row versions, including old versions. Autovacuum may fail to process a table because the database horizon has not been shifted for a long time or because an incompatible lock was set on the table when the table was accessed . In the second case, autovacuum skips processing the table. This leads to an increase in the size of table and index files. After autovacuum has completed its work, the file sizes are unlikely to decrease. The blocks will be used in the future for new row versions. Bloat of tables and indexes can be considered an increase in size so that the free space will not be used in the near future. If the object size is large, the unused space may be noticeable to the administrator. You can find tables with unused space and run maintenance tasks using the Tantor Platform.

You can estimate unused space based on basic statistics collected by autoanalysis . Objects are unlikely to bloat quickly, so you don't need to monitor them often. Monitoring free disk space is more relevant. The accuracy of the estimate can be verified (compared with reality) by performing a full vacuum (`CLUSTER` or `VACUUM FULL`) and compare the result with the estimate.

Example queries:

```
https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/b1_table_estimation.sql
https://raw.githubusercontent.com/NikolayS/postgres_dba/refs/heads/master/sql/b2_btree_estimation.sql
```

Example of query result:

```
Is N/A | Table | Size | Extra |Bloat estimate| Live | Last Vaccum | Fillfactor
-------+-----------------+-----+--------------+--------------+--------+------------------+----------
| pgbench_history |2832kB|~16 kB (0.56%)|~16 kB (0.56%)|~2816 kB |  11:11:11 (auto)| 100
```

You can use the functions of the standard pgstattuple extension :

```
create extension pgstattuple ;
\ dx + pgstattuple
select relname , b.* from pg_class , pgstattuple_approx ( oid ) b WHERE relkind ='r' order by 9
desc ;
select relname , b.* from pg_class , pgstatindex ( oid ) b WHERE relkind = ' i ' order by 10;
```

You can estimate by `dead_tuple_percent` for tables and `avg_leaf_density` for indexes.

Practice

- `Vacuum` command parameters
- Part 2: Vacuum Observation
- Part 3. Extension for viewing visibility map and freezing `pg_visibility`
- Part 4. Interval between autovacuum cycles, `autovacuum_naptime` `parameter`
- Part 5. Comparison of vacuum version 17 with previous versions
- Part 6. Number of index scans during vacuuming
- Part 7. Autovacuum logging
- `pgststtuple` extension
- Part 9: Index Processing Condition: 2% of Rows

Practice

The practice examines in detail the operation of the vacuum and its diagnostics.
You will compare the vacuum operation in versions 16 and 17.

13

Using the diagnostic log

Diagnostic log

- `logging_collector ( off` by default) It is recommended to set the value to `on`
- a background process logger will start , which collects messages sent to stderr and writes them to log files
- `log_min_messages = WARNING` by default , which means logging messages with levels `ERROR, LOG, FATAL, PANIC`
- `log_min_error_statement =` ERROR by default. Minimum severity level for SQL statements that fail with an error.
- `log_directory = log` (PGDATA/log) by default . Specifies the path to the log file directory, can be changed to a mount point, the speed and volume of which are sufficient to receive logs . There are parameters that configure the rotation of log files.

```
psql -c " altern system set logging_collector = on; "
sudo systemctl restart tantor-se-server-16
ps - ef | grep logger
postgres 21861 21860 0 09:37 00:00:00 postgres : logger
```


Diagnostic log

PostgreSQL code contains function calls of the following type:
`ereport (WARNING, ( errcode ( MESSAGE_CODE ), errmsg ( " message text ")));`
The first parameter is the severity level (error level codes) . In elog.h 15 levels are defined.
Enabling the log collector process:
`logging_collector = on` (default off). It is recommended to set the value to on. By default, messages are transmitted to syslog and written in its format, which is inconvenient for analysis . With a large number of messages that cannot be handled (the speed of writing to the file is lower than the speed of generation) syslog does not record some messages (and rightly so), logger does not clear the errlog buffer and the instance processes generating messages are blocked until the logger writes down everything that has accumulated (which is also correct). In other words, the logger does not lose messages, which can be important for diagnostics. This situation can occur due to a failure to write to log files or enabling high logging level .
If `logging_collector = on` , a background process logger is started , which collects messages sent to stderr and writes them to log files.
The level of messages written to the cluster log is specified by the parameters:
`log_min_messages` , by default WARNING , which means logging messages with levels `ERROR, LOG, FATAL , PANIC` .
`log_min_error_statement` , defaults to ERROR . Sets the minimum severity level for SQL commands that failed
`log_destination = stderr` no need to change
`log_directory = log` (PGDATA/log) by default . Specifies the path to the log file directory. You can specify an absolute path (/u01/log) or relative to PGDATA (../log) .
The name of the current log file (or files) is specified in the text file `PGDATA/ current_logfiles`
Levels of importance from most to least detailed:
`DEBUG5 DEBUG4 DEBUG3 DEBUG2 DEBUG1` for debugging
`INFO` messages typically requested by the command option (VERBOSE)
`NOTICE` Helpful messages for customers
`WARNING` Warnings about possible problems
`ERROR` an error that caused the current command to be terminated
`LOG` messages useful for administrators
`FATAL` error due to which the server process was stopped (session ended)
`PANIC` stopping server processes by main process

Diagnostic parameters

- logging configuration parameters more than 35
- there are also more than 8 parameters for debugging SQL commands
- logging parameters
- Main parameters:

```
alter system set logging_collector =on;
alter system set log_min_duration_statement ='8s';
alter system set log_statement = ddl ;
alter system set log_min_error_statement =ERROR;
alter system set log_temp_files ='1MB';
alter system set cluster_name ='main';
alter system set log_autovacuum_min_duration ='10s';
alter system set log_disconnections =on;
alter system set log_connections =on;
alter system set log_lock_waits =true;
alter system set deadlock_timeout ='60s';
alter system set log_recovery_conflict_waits =on;
select pg_reload_conf ();
```

```
\ dconfig *debug*
debug_assertions | off
debug_discard_caches | 0
debug_io_direct |
debug_logical_replication_streaming | buffered
debug_parallel_query | off
debug_pretty_print | on
debug_print_parse | off
debug_print_plan | off
jit_debugging_support | off
(10 rows)
\ dconfig log*
log_autovacuum_min_duration | 10min
log_checkpoints | on
log_connections | off
log_disconnections | off
log_duration | off
log_error_verbosity | default
log_executor_stats | off
logging_collector | off
log_lock_waits | off
log_file_mode | 0600
log_filename | postgresql -%Y-%m-% d_%H%M%S.log
logging_collector | on
log_hostname | off
logical_decoding_work_mem | 64MB
log_line_prefix | %m [%p]
log_lock_waits | off
log_min_duration_sample | -1
log_min_duration_statement | -1
```


Diagnostic parameters

What parameters can be used to monitor potential performance issues ?

`log_min_duration_statement= '8s'` all commands that take the specified amount of time or longer to execute will be logged. If the value is zero, the duration of all commands is logged. By default, -1 does not log anything. It is recommended to set this to detect long-running commands (they hold the database horizon) ; cases of performance degradation due to which the duration of command execution increases ; problems with commands: for example, an index is no longer used and the command execution time increases sharply. Example:

```
LOG: duration: 21585.110 ms
STATEMENT: CREATE INDEX ON test(id);
```

Duration and command are given.

`log_duration=o ff` logs the duration of all commands after they are executed. Disadvantage: all commands are logged (without text), one line per command. It is not worth enabling at the cluster level. Advantage: the text of commands is not logged . The parameter can be used to collect statistics for all commands, but for this you will need some program to process the log file to analyze the collected data. It is not necessary to enable it for the entire cluster, the parameter can be enabled at any level. Example:

```
LOG: duration: 21585.110 ms
```

`log_statement= ddl` what types of SQL commands will be logged . Values: `none` (disabled), `ddl` , `mod` (what ddl is plus dml commands), `all` (all commands). Default is none . Recommended to set to ddl . ddl commands usually set a higher blocking level, which increases contention . The parameter can be used to detect or exclude the execution of the ddl command as a reason for the decline in productivity. Commands with syntax errors are not logged by default. If you need to log commands with syntax errors, you need to set `log_min_error_statement = ERROR` (or more detailed) . Should you log commands with syntax errors ? Commands do not load the server process, but can significantly increase network traffic. The cause of errors may be in the application code, which continuously repeats the command in a loop. You can **periodically** enable logging of erroneous commands. Example of a record with `log_statement= ddl set` :

```
LOG: statement: drop table test;
```

Monitoring temporary file usage

- `cluster_name = 'main'` Default is empty. Recommended to set. The value is appended to the instance process name, making it easier to identify. On a replica, `wal_receiver is used for identification by default.`
- `log_temp_files = '1MB'` (disabled by default) logs the names and sizes of created temporary files **when they are deleted**
- if the value is zero, files of any size are logged
- Temporary files are created in the directory of tablespaces specified in the `temp_tablespaces parameter.`
  - can be limited by the `temp_file_limit parameter`
  - It is recommended to set `log_temp_files` and `temp_file_limit`
- example of a message that the temporary file has grown to 97 MB :

```
STATEMENT: CREATE INDEX ON test(id);
LOG: temporary file: path "base/ pgsql_tmp
/pgsql_tmp137894.0.fileset/0.0",
size 101 810176
```


Monitoring temporary file usage

If there are a lot of commands and the log is cluttered, you can use `the log_min_duration_sample parameters and log_statement_sample_rate` . Parameter `log_transaction_sample_rate` has high overhead as all transactions are processed.
`cluster_name = 'main'` Empty by default. Recommended to set. The value is appended to the instance process name, making it easier to identify. On a replica, `wal_receiver is used for identification by default` .

`log_temp_files = '1MB'` logs the names and sizes of temporary files created at the time of their deletion. Why at the time of deletion ? Because files grow in size and the size to which they have grown is known only at the time of file deletion. How can I prevent files from growing ? The size of temporary files (**including temporary table files**) can be limited by the `temp_file_limit parameter` . If the size is exceeded, the commands will return an error. Example:

```
insert into temp1 select * from generate_series (1, 1000000);
ERROR: temporary file size exceeds temp_file_limit (1024kB)
```

Setting `temp_file_limit` will help identify errors that cause the execution plan to be suboptimal, such as not being able to use an index and instead sorting huge amounts of rows.
If the value is zero , files of any size are logged , and if the value is positive, files of a size not less than the specified value are logged. The default value is -1, logging is disabled. It is recommended to set `log_temp_files` to a relatively large value to detect the occurrence of commands that load the disk system. The disk system is the most loaded resource in the DBMS.

```
LOG: temporary file: path "base/ pgsql_tmp /pgsql_tmp36951.0", size 71 835648
STATEMENT: explain (analyze) select p1.*, p2.* from pg_class p1, pg_class p2
order by random();
```

Temporary files are created in the directory of the tablespaces specified by the `temp_tablespaces parameter` .
https://docs.tantorlabs.ru/tdb/en/16_4/se/runtime-config-logging.html

Monitoring the operation of autovacuum and autoanalysis

- `log_autovacuum_min_duration` , by default set to 10 minutes. If autovacuum exceeds this time while processing a table, a message will be written to the cluster log. If such messages appear, it is worth finding out the reason for the long vacuuming of the table

```
LOG: automatic vacuum of table " postgres.public.test ": index scans: 37
pages: 0 removed, 88496 remain, 88496 scanned (100.00% of total)
tuples : 10000000 removed , 10000000 remain, 0 are dead but not yet removable
removable cutoff: 799, which was 0 XIDs old when operation ended
new relfrozenxid : 798, which is 2 XIDs ahead of previous value
frozen: 1 pages from table (0.00% of total) had 82 tuples frozen
index scan needed: 44249 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
avg read rate : 518.021 MB/s, avg write rate : 47.473 MB/s
buffer usage: 385747 hits, 1864788 misses, 170895 dirtied
WAL usage: 231678 records, 83224 full page images , 248448578 bytes
system usage: CPU: user: 25.72 s, system: 0.38 s, elapsed: 28.12 s

LOG: automatic analyze of table " postgres.public.test "
avg read rate: 498.808 MB/s, avg write rate: 0.018 MB/s
buffer usage: 2906 hits, 27199 misses, 1 dirtied
system usage: CPU: user: 0.42 s, system: 0.00 s, elapsed: 0.42 s
```


Monitoring the operation of autovacuum and autoanalysis

`log_autovacuum_min_duration` , by default set to 10 minutes. If autovacuum exceeds this time while processing a table , a message will be written to the cluster log. 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.

The message is written if " **`elapsed:`** " > `log_autovacuum_min_duration`

The total duration of processing the table and its indexes is specified in " **`elapsed:`** ". The value will be greater than user + system . user and system is the processor usage time. When vacuuming, processes can send blocks for writing and wait for the I/O operation to be performed, without loading the processor.

First of all, it is worth looking at " **`elapsed:`** " - this is the duration of the autovacuum transaction, that is, holding the horizon. For TOAST , there will be a separate entry about vacuuming (including aggressively) in the log with its own indicators, like a regular table. There will be no entry about autoanalysis for TOAST , since **TOAST is not analyzed** :

```
analyze pg_toast.pg_toast_25267;
WARNING: skipping "pg_toast_25267" --- cannot analyze non-tables or special system tables
```

Secondly, it is worth pay attention to the number of index scans : . A value greater than 1 indicates that there was not enough memory to build the TID list . In this case, it is worth increasing the value of the parameter:

```
alter system set autovacuum_work_mem ='1000MB'; select pg_reload_conf ();
```

Thirdly , the efficiency indicators of the autovacuum cycle are " `tuples :` " and " `frozen:` ". " `scanned` " will be less than 100% if the blocks were cleared in the previous vacuum cycle, this is normal.

The value of " `full page images` " (and " `bytes` " proportional to it) do not relate to the efficiency of the vacuum and are determined by chance: how long ago the checkpoint was, or whether it is necessary to increase `checkpoint_timeout` . Even the opposite, if the value " `full page images` " large, then this may explain the long cycle (the value in " **`elapsed:`** ") . Large values of " `full page images` " and " `bytes` " together with " `tuples : number removed` " mean the efficiency of the autovacuum cycle or that it has not processed the table for a long time (for example, it could not lock).

" `avg read rate` " and " `avg write rate` " I/O cannot be estimated because it may not be a bottleneck.

Monitoring checkpoints

- the first entry is written to the log when the checkpoint begins
- total = 09:31:35.070 - 09:27:05.095
  - roughly corresponds 270 seconds, which are obtained by multiplying checkpoint_completion_target * checkpoint_timeout (0 .9 *300 = 270)
- total = write + sync time of writing to **WAL** - files
- sync = time spent on fdatasync calls on **WAL** files

```
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%) ; 0 WAL file(s) added, 0 removed,
6 recycled; write=269.938 s , sync=0.009 s , total=269.976 s ; sync files=15, longest=0.003 s, average=0.001
s ;
distance=109699 kB , estimate=109699 kB ; lsn =8/1164B2E8, redo lsn =8/BC98978
```

- sync files=15 (files synced) - number of data files (in tablespaces whose blocks are located in the buffer cache) to which the write was made and on which the fsync call was sent at the end of the checkpoint
- longest=0.003 s (the_longest sync) - the longest duration of processing one file

```
LOG: checkpoint complete: wrote 8596 buffers (52.5%); 0 WAL file(s) added, 0 removed, 33 recycled;
write=25.057 s, sync=9.212 s, total=35.266 s; sync files=4, longest=9,181 s, average=2,303 s;
distance=540552 kB , estimate=550280 kB ;  lsn =9/16BC03F0, redo lsn =8/F82504B0
```


Monitoring checkpoints

log_checkpoints **by default on since version 15** . It is not worth disabling, as it allows you to track the frequency of checkpoints. More frequent checkpoints lead to a temporary increase in the load on the journal system (WAL).

log_checkpoints creates log entries of this type:

09:27:05.095 LOG: checkpoint starting: time

09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%) ;

0 WAL file(s) added, 0 removed, 6 recycled; write=269.938 s , sync=0.009 s , total=269.976 s ; sync files=15, longest=0.003 s, average=0.001 s ; distance=109699 kB , estimate=109699 kB ; lsn =8/1164B2E8, redo lsn =8/BC98978

How to read the entries:

1) The first entry is written to the log when the checkpoint starts. There may be many entries between this entry and the checkpoint end entry . Meaning total = 09:31:35.070 - 09:27:05.095 which roughly corresponds to 270 seconds, which are obtained by multiplying checkpoint_completion_target * checkpoint_timeout (0 .9 *300 = 270). The number of blocks that checkpointer should send for writing is calculated quite often, but towards the end of the interval the I/O load may suddenly increase and checkpointer may not make it in time for the specified interval. To minimize the probability of not fitting into the interval between checkpoints (checkpoint_timeout) , the default value for checkpoint_completion_target is 0.9, which leaves a gap of 10% (0.1) .

2) total = write + sync . sync is the time spent on fsync calls . Big sync time indicates high I/O load. These indicators relate to data files.

LOG: checkpoint complete: wrote 8596 buffers (52.5%); 0 WAL file(s) added, 0 removed, 33 recycled; write=25.057 s, sync=9.212 s, total=35.266 s; sync files=4, longest=9.181 s, average=2.303 s; distance=540552 kB , estimate=550280 kB ; ..

3) sync files=15 (files synchronized) - the number of processed files whose blocks are located in the buffer cache (relations). The checkpoint at the beginning writes the buffer blocks slru caches , but their sizes are small. longest=0.003 s (the_longest sync) - the longest duration of processing one file . average=0.001 s - the average time of processing one file . These indicators apply to tablespace files.

Description of `log_checkpoints entries`

- `wrote 4315 buffers` number of dirty buffers written by checkpoint. At the same time as checkpointer, dirty blocks can be written by server processes and bgwriter
- `(26.3%)` percentage of the total number of buffer cache buffers specified by the `shared_buffers parameter`
- `file(s) added, 0 removed, 6 recycled` number of created, deleted, recycled WAL segments
- `distance=109699 kB` (distance) - the volume of WAL records between **the start of the previous** checkpoint and the start of the current one

```
09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6
recycled;
write=269.951 s, sync=0.009 s, total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB , estimate=108859 kB ; lsn =8/AA004C8, redo lsn =8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed, 6 recycled
;
write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB , estimate=109699 kB ; lsn =8/1164B2E8, redo lsn =8/BC98978
```


Description of `log_checkpoints entries`

`log_checkpoints` creates log entries of this type:

```
09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%);
0 WAL file(s) added, 0 removed, 6 recycled; write=269.951 s, sync=0.009 s,
total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB , estimate =108859 kB ; lsn =8/AA004C8, redo lsn =8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%);
0 WAL file(s) added, 0 removed, 6 recycled ; write=269.938 s, sync=0.009 s,
total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s; distance
=109699 kB , estimate =109699 kB ; lsn =8/1164B2E8, redo lsn =8/BC98978
```

How to read the entries (continued):

4) `wrote 4315 buffers` - the number of dirty blocks written by checkpoint. At the same time as checkpointer, dirty blocks can be written by server processes and bgwriter . `(26.3%)` percentage of the total number of buffers in the buffer cache , specified by the `shared_buffers parameter` . In the example, `4315 / 16384 * 100% = 2 6.33 66699 %`

5) `file(s) added, 0 removed, 6 recycled` number of created, deleted, recycled WAL segments (by default, the size of each segment is 16 MB).

6) `distance=109699 kB` (distance) - the volume of WAL records between the start of the previous checkpoint and the start of the completed checkpoint

`select ' 8/BC98978 '::pg_lsn-' 8/5177990 ':: pg_lsn ;` = 112332776 = 109699kB

Description of log_checkpoints entries

- estimate=109699 kB (distance expected) is calculated to estimate how many WAL segments will be used at the next checkpoint
- if zeros in " 0 WAL file(s) added, 0 removed " , then estimate correct. How many files to delete is determined by the parameters min_wal_size , max_wal_size , wal_keep_size , max_slot_wal_keep_size , wal_init_zero =on, wal_recycle = on

```
09:22:05.087 LOG: checkpoint starting: time
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6
recycled;
write=269.951 s, sync=0.009 s, total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB , estimate=108859 kB ; lsn =8/AA004C8, redo lsn =8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed, 6 recycled
;
write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB , estimate=109699 kB ; lsn =8/1164B2E8, redo lsn =8/BC98978
```


Description of log_checkpoints entries (continued)

7) After checkpoint starting: the properties of the checkpoint are specified. time means that the checkpoint was called " by time " after checkpoint_timeout . If the WAL size exceeds max_wal_size there will be a message:

LOG: checkpoint starting: wal

If the checkpoint is wal starts earlier than checkpoint_warning , then the message will be displayed:

LOG: checkpoints are occurring too frequently (23 seconds apart)
HINT: Consider increasing the configuration parameter " max_wal_size ".

23 seconds less than set checkpoint_warning = '30 s '

For checkpoints after instance restart:

LOG: checkpoint starting: end-of-recovery immediate wait

8) estimate=109699 kB (distance that was expected) - updated according to the formula:

if (**estimate** < **distance**) **estimate** = **distance**
else estimate=0.90*estimate+0.10*distance; (numbers are fixed in PostgreSQL code)

Estimate indicator is calculated by the checkpoint code to estimate how many WAL segments will be used at the next checkpoint . Based on estimate at the end of the checkpoint, it is determined how many files to rename for reuse and the remaining ones to delete. How many files to delete is determined by the parameters min_wal_size , max_wal_size , wal_keep_size , max_slot_wal_keep_size , wal_init_zero = on, wal_recycle = on . File reuse should not be disabled, it is optimal for the ext4 file system. Other file systems (zfs , xfs , btrfs) should not be used. If zeros in " 0 WAL file(s) added, 0 removed " , then estimate true. Such values should be most of the control points. The purpose of displaying the estimate value in this. The volume of log records between checkpoints is **distance** .

9) Between checkpoints passed 09:27:05 .095 - 09:22:05.087 = 30 0 .0 08 seconds, which with high accuracy equals checkpoint_timeout =300 s

Utility pg_waldump and records log_checkpoints

- To view records in WAL files, use the pg_waldump utility . By default, the utility searches for WAL files in " . " (the current directory from which it is launched) , further in ./pg_wal , $PGDATA/pg_wal
- in the log and output of pg_controldata in LSN leading zeros after "/" are not printed
- pg_ waldump output in lsn and prev zero it prints , but in redo it doesn't

```
pg_controldata | grep check | head -n 3
Latest checkpoint location: 8/1164B2E8
Latest checkpoint's REDO location: 8/ 0 BC98978
Latest checkpoint's REDO WAL file: 000000010000000800000000B
```

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
rmgr : XLOG len ( rec /tot): 148/148, tx : 0,
lsn : 8/1164B2E8 , prev 8/1164B298 , desc : CHECKPOINT_ONLINE redo 8/ 0 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
pg_waldump : error: error in WAL record at 8/1361C488: invalid record length at 8/1361C4B0:
expected at least 26, got 0
```

```
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed,
6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB , estimate=109699 kB ; lsn =8/1164B2E8, redo lsn =8/ 0 BC98978
```


Utility pg_waldump and log_checkpoints entries

The data about the last checkpoint is written to the control file. To view the contents of the control file, use the **pg_controldata utility** :

```
pg_controldata | grep check | head -n 3
Latest checkpoint location: 8/1164B2E8
Latest checkpoint's REDO location: 8/ 0 BC98978
Latest checkpoint's REDO WAL file: 000000010000000800000000B
```

The zero after the slash ("/") is not printed; in the examples on the slide and below the slide, the zeros were added manually.

The data corresponds to the last checkpoint entry in the log.

To view records in WAL files, use the utility pg_waldump . By default, the utility searches for WAL files in the current directory from which it is launched, then in the directories ./pg_wal, $PGDATA/pg_wal . An example of viewing a log entry about the end of a checkpoint:

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
```

or

```
pg_waldump -s 8/BC98978 | grep CHECKPOINT
rmgr : XLOG len ( rec /tot): 148/148, tx : 0,
lsn : 8/1164B2E8, prev 8/1164B298, desc : CHECKPOINT_ONLINE redo 8/ 0 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
```

The utility does not specify an LSN to scan the log to (the -e parameter), so when it reaches the very last log entry that was written to the log, the utility displays a message that the next entry is empty:

```
pg_waldump : error: error in WAL record at 8/1361C488: invalid record length at
8/1361C4B0: expected at least 26, got 0
```

In the log, output of the pg_controldata utility in LSN, leading zeros after "/" are not printed in the output of pg_waldump in lsn and prev zero is printed, but in redo it is not printed . Before the number 8, zeros are also invisibly present, but their absence does not create confusion. You can remember that after the slash there should be eight HEX symbols.

Utility pg_waldump and records log_checkpoints

- lsn 8/1164B2E8 end of checkpoint record
- red 8/0 BC98978 record of the start of the checkpoint from which recovery will begin in the event of an instance failure
- prev 8/1164B298 address of the start of the previous log entry
- distance volume magazine from start of previous to start of completed checkpoint ' 8/ 0 BC98978 ':: pg_lsn -' 8/ 0 5177990 '::

```
pg_controldata | grep check | head -n 3
Latest checkpoint location: 8/1164B2E8
Latest checkpoint's REDO location: 8/BC98978
Latest checkpoint's REDO WAL file: 000000010000000800000000B
```

```
pg_waldump -s 8/0B000000 | grep CHECKPOINT
rmgr : XLOG len ( rec /tot ): 148/148 , tx : 0,
lsn : 8/1164B2E8 , prev 8/1164B298 , desc : CHECKPOINT_ONLINE redo 8/BC98978 ; ...
```

```
09:26:35.066 LOG: checkpoint complete: wrote 3019 buffers (18.4%); 0 WAL file(s) added, 0 removed, 6
recycled;
write=269.951 s, sync=0.009 s, total=269.980 s; sync files=14, longest=0.004 s, average=0.001 s;
distance=99467 kB , estimate=108859 kB ; lsn =8/AA004C8, redo lsn =8/5177990
09:27:05.095 LOG: checkpoint starting: time
09:31:35.070 LOG: checkpoint complete: wrote 4315 buffers (26.3%); 0 WAL file(s) added, 0 removed,
6 recycled; write=269.938 s, sync=0.009 s, total=269.976 s; sync files=15, longest=0.003 s, average=0.001 s;
distance=109699 kB , estimate=109699 kB ; lsn =8/1164B2E8 , redo lsn =8/BC98978
```


Utility pg_waldump and log_checkpoints entries (continued)

lsn 8/1164B2E8 , end of checkpoint record.

red 8/0 BC98978 record of the beginning of the checkpoint from which recovery will begin in the event of an instance failure. The address of the record that was formed at the time of the beginning of the checkpoint (redo) is selected from the record, and this record is read. All records from redo to lsn must be read and overlaid on the cluster files. After applying lsn The cluster files are considered consistent.

prev 8/1164B298 address of the beginning of the previous record in the log. You can slide " backward " through the log . In this case, the LSN of the next log record is missing in the log records . Why ? The address of the next log record can be calculated by the len field (rec /tot): 148/148 which stores the length of the log record. The minimum length of a log record is 26 bytes (expected at least 26). In this case, the actual length of the log record is padded . up to 8 bytes. The actual length of the record in the example will be **152** bytes, not 148 . Example:

```
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; ...
rmgr : Heap len ( rec /tot): 86/ 86, tx : 8232886, lsn : 8/1164B380 , prev 8/1164B2E8, desc : HOT_UPDATE ...
```

lsn + len + padding to 8 bytes = LSN of start of next record

The size of the journal entries can be determined from the log or control file entry. The recovery time depends on it.

WAL written at a checkpoint is calculated from these fields :

select pg_wal_lsn_diff ('8/1164B2E8','8/BC98978'); = 94054768 = 91850kB.

Volume WAL from start to end of checkpoint 91850 kB.

Volume from the beginning of the previous checkpoint before the beginning of the completed one, that is, the distance between the control points :

select ' 8/BC98978 ':: pg_lsn - ' 8/5177990 ':: pg_lsn ; = 112332776 = 109699kB

For calculations, you can use the pg_wal_lsn_diff function or the "-" operator , the results are the same. To use the operator, you need to cast the string to the pg_lsn type .

Diagnostics of database connection frequency

- `log_disconnections = on` logs a session termination event. The same information as `log_connections` plus the session duration is logged .
  - the advantage is that it outputs one line, which does not clutter the log
  - allows you to identify short-term sessions
  - example of session duration message session 4 seconds:

```
LOG: disconnection: session time: 0:00:0 4 .056 user= oleg database=db1 host=[vm1]
```

  - the disadvantage is that unsuccessful attempts are distinguished only by an additional line :

```
LOG: connection received: host=[local]
LOG: connection authorized: user= postgres database=db2 application_name = psql
FATAL: database "db2" does not exist
LOG: connection received: host=[local]
LOG: connection authorized: user= alice database= alice application_name = psql
FATAL: role " alice " does not exist
```


Diagnostics of database connection frequency

`log_disconnections= on` logs the session termination event. The same information is logged as `log_connections` plus session duration . The advantage is that one line is output, which does not clutter the log. Allows you to identify short-term sessions. Short sessions lead to frequent spawning of server processes, which increases the load and reduces performance:

```
LOG: disconnection: session time: 0:00:0 4 . 0 56 user= oleg database=db1
host=[vm1]
```

In the example the session duration is 4 seconds.

`log_connections= on` logs attempts to establish a session. The disadvantage is that for many types of clients, two lines are output to the log : the first line about determining the authentication method (without a password, with a password), the second line - authentication. If a connection balancer (pgbouncer) is not used, then a server process is spawned before authentication, which is a labor-intensive operation. The parameter is useful for identifying problems when a client tries to connect non-stop with an incorrect password or to a non-existent database or with a non-existent role. The disadvantage is that unsuccessful attempts differ only by an additional line :

```
LOG: connection received : host=[local]
LOG: connection authorized : user=postgres database=db2 application_name = psql
FATAL: database "db2" does not exist
LOG: connection received : host=[local]
LOG: connection authorized : user= alice database= alice application_name = psql
FATAL: role " alice " does not exist
```

`log_hostname = off` . It is not worth enabling, as it introduces significant delays in logging session creation.

Diagnostics of blocking situations

- `log_lock_waits =true` . Disabled by default. Recommended to enable to get messages in the diagnostic log when a process waits longer than : `deadlock_timeout`
- `deadlock_timeout = '60s'` . The default is 1 second, which is too low and creates significant overhead on busy instances.
- `log_startup_progress_interval ='10s'` don't turn it off
- `log_recovery_conflict_waits=on` . Default `is off` . Process `startup` will write a message to the replica log if it cannot apply WAL to the replica for longer than `deadlock_timeout`

```
LOG: recovery still waiting after 60.555 ms: recovery conflict on lock
DETAIL: Conflicting process: 5555.
```

- The presence of conflicts can be seen in the presentation (few details):

```
select * from pg_stat_database_conflicts where datname ='postgres';
datid|datname | tblspc|confl_lock|confl_snapshot|confl_bufferpin|deadlock
---------+--------+------+---------+-------------+--------------+--------
13842|postgres| 0 | 0 | 1 | 1 | 0
```


Diagnostics of blocking situations

`log_lock_waits =true` . Disabled by default. Recommended to enable to get messages in the diagnostic log when a process waits longer than : `deadlock_timeout = '60s'` . The default is 1 second, which is too low and creates significant overhead on busy instances. It is recommended to configure the `deadlock_timeout` value . so that messages about waiting for a lock to be received occur rarely. As a first approximation, you can focus on the duration of a typical transaction (for a replica - the longest request).

In version 15, a parameter appeared `log_startup_progress_interval ='10s'` which **should not be disabled** (set to zero). If the `startup process` (performs recovery) encounters a long operation, a message about this operation will be written to the log. The messages will allow you to identify either problems with the file system or a high load on the disk system. Example of startup process messages during recovery:

```
LOG: syncing data directory ( fsync ), elapsed time: 10.07 s, current path: ./base/4/2658
LOG: syncing data directory ( fsync ), elapsed time: 20.16 s, current path: ./base/4/2680
LOG: syncing data directory ( fsync ), elapsed time: 30.01 s, current path: ./base/4/PG_VERSION
```

`log_recovery_conflict_waits=on` . Default `is off` . **The parameter appeared in version 14** . The `startup process` will write a message to the replica log if it cannot apply WAL to the replica for longer than `deadlock_timeout` . The delay may occur because the server process on the replica is executing a command or transaction (for read repeatability) and blocks WAL application due to the `max_standby_streaming_delay parameter` (default 30 s) . Allows you to identify cases of replica lagging. Effective on the replica, can be set in advance on the master. It is recommended to set this to `on` .

```
LOG: recovery still waiting after 60 .555 ms: recovery conflict on lock
DETAIL: Conflicting process: 5555.
CONTEXT: WAL redo at 0/3044D08 for Heap2/PRUNE: latestRemovedXid 744 nredirected
0 ndead 1; blkref #0: rel 1663/13842/16385, blk 0
```

The presence of conflicts can be seen in the view , but it has little detail:

```
select * from pg_stat_database_conflicts where datname ='postgres';
datid|datname | tblspc|confl_lock|confl_snapshot|confl_bufferpin|deadlock
----------+--------+------+---------+-------------+--------------+ -- -- ---
-
13842|postgres|0|0 | 1 | 1 |     0
```

Practice

- Part 1. Reading vacuum and autovacuum messages
- Part 2. Reading checkpoint messages
- Part 3. Reading pg_waldump checkpoint messages
- Part 4. Size of the PGDATA/pg_wal directory

Practice

The practice covers some of the topics covered in this chapter.

14

Cumulative statistics

Cumulative statistics

- accumulates statistics about the operation of the instance. Collecting statistics creates overhead, so there are parameters that can be used to enable the collection of additional statistics
- parameters that are enabled should not be disabled, since their result is used by the instance processes

```
select name, setting, unit, context, min_val , max_val from pg_settings where
name like 'track%';
 name | setting | unit | context | min_val | max_val
--------------------------+---------+------+------+---------+---------
 track_activities | on | | superuser | |
 track_activity_query_size | 1024 | B | postmaster | 100 | 1048576
 track_commit_timestamp | off | | postmaster | |
 track_counts | on | | superuser | |
 track_functions | none | | superuser | |
 track_io_timing | off | | superuser | |
 track_wal_io_timing | off | | superuser | |
(7 rows)
```


Cumulative statistics

accumulates statistics about the operation of the instance. Collecting statistics creates overhead, so there are parameters that can be used to enable the collection of additional statistics :

```
select name, setting, unit, context, min_val , max_val from pg_settings where name like 'track%';
 name | setting | unit | context | min_val | max_val
--------------------------+---------+------+------+---------+---------
 track_activities | on | | superuser | |
 track_activity_query_size | 1024 | B | postmaster | 100 | 1048576
 track_commit_timestamp | off | | postmaster | |
 track_counts | on | | superuser | |
 track_functions | none | | superuser | |
 track_io_timing | off | | superuser | |
 track_wal_io_timing | off | | superuser | |
(7 rows)
```

Table and index usage statistics (`track_counts = on`) is used autovacuum and there is no need to disable its collection. What is disabled by the instance is not used. `The track_commit_timestamp statistics` can be used in the conflict resolution logic for logical replication.

`track_functions` enables counting of function calls and their execution time. The value pl enables tracking of functions on in `plpgsql` language , and `all` also functions in SQL and C languages . The data can be used when optimizing the code of subroutines.

`track_io_timing` accumulates not only wait events, but also the duration of events. This parameter is useful for monitoring and during performance tuning. It is disabled by default, since it will require frequent access to the time counter, which slows down the work. Statistics can be viewed in the pg_stat_database , pg_stat_io views .

`track_wal_io_timing` enables measuring the duration of writing to logs. Statistics can be viewed in the pg_stat_wal view . Disabled by default, since it requires frequent access to the time counter, which slows down the work. The speed of access to the counter can be measured with the command line utility `pg_test_timing` . If you do not use `pg_stat_wal` and do not analyze the speed of the logs, then it is not worth turning on.

`stats_fetch_consistency` by default cache , no need to change . If you open a transaction, then the first time you access statistics for an object , it will not be updated until the end of the transaction. You can update by calling `pg_stat_clear_snapshot ()` . The value none will not cache statistics. The value snapshot caches all statistics accumulated on all database objects, which costs resources. Without a transaction, there is no difference.

Utility pg_test_timing

- allows you to estimate the costs of obtaining time
- any statistics with time is a consequence of accessing the time counter
- An example of the overhead of the EXPLAIN ANALYZE command , which makes 2 timing measurements per row:

```
CREATE TABLE t AS SELECT * FROM generate_series
(1.100000);
\ timingcv
SELECT COUNT(*) FROM t;
Time: 15.526 ms
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
Time: 310.682 ms
```

```
pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 1424.21 ns
Histogram of timing durations:
< us % of total count
1 0.03043 641
     2 64.30326 1354500
4 35.46601 747065
8 0.01234 260
16 0.09889 2083
```

- depends on the time counter used
- The delay is measured by the command lin
- the fastest tsc counter :

```
cat /sys/devices/system/ clocksource /clocksource0/ current_clocksource
tsc
```


Utility pg_test_timing

The utility allows you to estimate the costs of obtaining time. Any statistics with time is a consequence of accessing the time counter. How much do accesses affect the execution time of commands ?

You can estimate the overhead by comparing the execution speed of the command and the explain analyze command:

```
CREATE TABLE t AS SELECT * FROM generate_series (1.100000);
\timing
SELECT COUNT(*) FROM t;
Time: 15.526 ms
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
Time: 310.682 ms
```

295 milliseconds is the overhead of 2 measurements per row , which are performed by the explain analyze command.

The result of running the **pg_test_timing utility** :

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 1424.21 ns
Histogram of timing durations:
< us % of total count
1 0.03043 641
     2 64.30326 1354500
4 35.46601 747065
8 0.01234 260
16 0.09889 2083
```

1424.21 nanoseconds (1.4 microseconds) is the delay (overhead) for the time getting operation. 1.424 µs * 100000 rows * 2 dimensions = 284.8 ms, which is 295 ms.

If the time counter is fast, then the utility distribution histogram should give an overhead within 100 ns and more than 90% of requests should fit into the first row (1 us microsecond) in the distribution histogram.

The speed of the time counter depends on its type. The type can be viewed using the command:

```
cat /sys/devices/system/ clocksource /clocksource0/ current_clocksource
tsc
```

The fastest counter tsc . acpi_pm slower.

Viewing process statistics

- statistics are transferred by server processes to shared memory before idle time, i.e. after executing a command and no more often than once per second
- superusers and roles pg_read_all_stats , pg_monitor see data about all sessions. Other users see data about their session, about other sessions in the fields null
- To view cumulative statistics, more than 30 views and more than 100 functions named pg_stat are used *
- pg_stat_database view contains one row for each database:

```
select datname , numbackends , sessions, round(100.0* blks_hit /NULLIF(blks_read+blks_hit,0),2) hitratio ,
temp_files , temp_bytes , round((100.0* active_time /NULLIF( session_time+active_time , 0))::numeric,2)
activeratio , idle_in_transaction_time idleintrans , blk_read_time , blk_write_time from pg_stat_database ;
   datname | numbackends | sessions | hitratio | temp_files | temp_bytes | active ratio | idleintrans | blk_read_time | blk_write_time
-----------+-------------+----------+----------+------------+------------+--------------+-------------+---------------+----------------
 | 0 | 0 | 99.75 | 0 | 0 | | 0 | 0 | 0
postgres | 1 | 13 | 97.96 | 1 | 1400000 | 0.07 | 36177.453 | 8194.412 | 9.947
 my_test_db | 0 | 0 | | 0 | 0 | | 0 | 0 | 0
template1 | 0 | 0 | | 0 | 0 | | 0 | 0 | 0
template0 | 0 | 0 | | 0 | 0 | | 0 | 0 | 0
```


Viewing process statistics

Statistics are transferred by server processes to shared memory before idle time, that is, after executing a command and no more often than once per second. Exceptions:

1) the `pg_stat_activity view` , which displays information about the current query (if `track_activities =on`).

2) in views `pg_stat_xact_all_tables` , `pg_stat_xact_sys_tables` , `pg_stat_xact_user_tables` , `pg_stat_xact_user_functions` the transaction has access to its own statistics, which have not yet been transferred to shared memory

`)` `pg_stat_database` column .`numbackends`

Superusers and roles `pg_read_all_stats` ,`pg_monitor` see data on all sessions. Other users see data on their session, on other sessions in null fields. To view cumulative statistics, more than 30 views and more than 100 functions with the name `pg_stat * are used` .

View **`pg_stat_database`** contains one row for each database plus a row with `datid =0` with statistics on global objects.

Main columns:

`stats_reset` time and date of the last statistics reset

`numbackends` number of sessions to the database at the moment

`sessions` the number of sessions that have been created since the last statistics reset

`blks_read` - blocks were missing from the buffer cache , `blks_ hit` - blocks were present in the buffer cache . There is no need to overestimate this statistic, since there is a linux page cache . In the results of the `explain ( analyze, buffers) command` similar data is given in the lines:

`Buffers: shared hit =4073 read =56839`

`temp_files , temp_bytes` the number of temporary files created and how much is written in them

`blk_read_time , blk_write_time` are filled if `track_io_timing = on` time in milliseconds spent by server processes on input/output (reading or writing blocks of data files)

`active_time / session_time` session activity share

`idle_in_transaction_time` idle time if the server process has an open transaction. The lower the value, the better .

Performance pg_stat_database

- The pg_stat_database view contains one row for each database.
- if the xact_rollback number is growing , this may indicate errors in the application code
- if there are deadlocks , this indicates errors in the application logic

```
 select datname , round(100* xact_commit ::numeric/ nullif (xact_commit+xact_rollback,0),2) as
cratio , xact_rollback rollbacks, deadlocks, conflicts, temp_files , pg_size_pretty ( temp_bytes
) as tempsize from pg_stat_database ;
  datname | cratio | rollbacks | deadlocks | conflicts | temp_files | tempsize
 ---------+--------+-----------+-----------+-----------+------------+----------
 | | 0 | 0 | 0 | 0 | 0 bytes
 postgres | 100.00 | 10 | 0 | 0 | 0 | 0 bytes
```

```
 select datname , numbackends , sessions, round(100.0* blks_hit /NULLIF(blks_read+blks_hit,0),2) hitratio ,
temp_files , temp_bytes , round((100.0* active_time /NULLIF( session_time+active_time , 0))::numeric,2)
activeratio , idle_in_transaction_time idleintrans , blk_read_time , blk_write_time from pg_stat_database ;
   datname | numbackends | sessions | hitratio | temp_files | temp_bytes | active ratio | idleintrans | blk_read_time | blk_write_time
-----------+-------------+----------+----------+------------+------------+--------------+-------------+---------------+----------------
 | 0 | 0 | 99.75 | 0 | 0 | | 0 | 0 | 0
 postgres | 1 | 13 | 97.96 | 1 | 1400000 | 0.07 | 36177.453 | 8194.412 | 9.947
```


Performance pg_stat_database

Example request:

```
 select datname , numbackends , sessions, round(100.0* blks_hit /NULLIF( blks_read +
blks_hit,0),2) hitratio , temp_files , temp_bytes , round((100.0* active_time /NULLIF(
session_time+active_time , 0))::numeric,2) activeratio , idle_in_transaction_time
idleintrans , blk_read_time , blk_write_time from pg_stat_database ;
```

Percentage of hits in the buffer cache: `round(100.0* blks_hit /NULLIF(blks_read+blks_hit,0),2) hitratio`

The metric gives an estimate of what portion of blocks are taken from the buffer cache, and what from the disk or page cache. The hit percentage should be at least 90%. A lower value indicates that the " active set " - the data that queries work with - does not fit in the buffer cache . Because of this, blocks are most likely displaced and loaded again. In this case, it is worth increasing the value of `shared_buffers` and make sure there is enough physical memory.

You can also check if there are a large number of transaction rollbacks:

```
 select datname , round(100* xact_commit ::numeric/ nullif
(xact_commit+xact_rollback,0),2) as cratio , xact_rollback rollbacks, deadlocks,
conflicts, temp_files , pg_size_pretty ( temp_bytes ) as tempsize from pg_stat_database ;
  datname | cratio | rollbacks | deadlocks | conflicts | temp_files | tempsize
 ---------+--------+-----------+-----------+-----------+------------+----------
 | | 0 | 0 | 0 | 0 | 0 bytes
 postgres | 100.00 | 10 | 0 | 0 | 0 | 0 bytes
```

If the **xact_rollback number increases** , this may indicate errors in the application code.

If there are `deadlocks` , this indicates errors in the application logic.

`conflicts` are associated with replication.

When writing queries, you can use the `NULLIF() function` to avoid division by zero. To calculate decimal values, use `100 * integer::numeric` or `100.0* integer` .

Progress of command execution

- Six views contain data about the commands being executed:
- pg_stat_progress_analyze - `analyze` and autoanalysis
- pg_stat_progress_basebackup - `pg_basebackup` utilities
- pg_stat_progress_cluster - `cluster` and `vacuum full`
- pg_stat_progress_copy - `copy`
- pg_stat_progress_create_index - `create index` and `reindex`
- pg_stat_progress_vacuum
- you can estimate how long it will take for the command to be executed and observe the phases of command execution

```
pg_basebackup -D $HOME/backup/1 -P --max-rate=1000
28130/4564784 kB (0%), 0/1 tablespace
select * from pg_stat_progress_basebackup \watch 5
pid | phase |backup_total|backup_streamed|tablespaces_total|tablespaces_streamed
---+---------+------------+---------------+-----------------+--------------------
414|streaming database files| 4674334720 | 6502912 | 1 | 0
```


Progress of command execution

Six views **of** `pg_stat_progress_ *` contain data about the commands being executed: `analyze` and autoanalysis , `create index` and `reindex` , `vacuum` and autovacuum, `cluster` and `vacuum full` , `copy` , as well as the `pg_basebackup command line utilities` :

```
\ dv pg_stat_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)
```

Based on these representations, it is possible to estimate how long it will take for the command to be executed and to observe the phases of command execution.

Example of reservation monitoring:

```
pg_basebackup -D $HOME/backup/1 -P --max-rate=1000
28130/4564784 kB (0%), 0/1 tablespace
select * from pg_stat_progress_basebackup \watch 5
pid | phase |backup_total|backup_streamed|tablespaces_total|tablespaces_streamed
---+- ----- -----------------+-----------+---------------+-----------------+--------------------
414|streaming database files| 4674334720 | 6502912 | 1 | 0
```

Before version 17, the **`--max-rate option`** (reservation speed) loaded the processor core by 100%. In version 17 , the error was fixed and there is no load on the processor.

If in psql If you need to execute a command in a few seconds, then instead of " ; " at the end of the command you can use **`\watch seconds`** and the command will be executed at the specified interval.

Example of queries to views:

https://dev.to/bolajiwahab/progress-reporting-in-postgresql-1i0d

https://docs.tantorlabs.ru/tdb/ru/16_4/se/progress-reporting.html

Performance pg_stat_io

- input/output statistics
- statistics for each type of process
- object column can contain the following values:
  - relation (persistent storage object)
  - temp relation (temporary object)
- context column may contain the following values:
  - normal
  - bulkread , bulkwrite , vacuum (buffer rings)
- It is convenient to group by these columns:

```
select * from pg_stat_bgwriter \
gx
-[ RECORD 1 ]--------+--------
checkpoints_timed | 151
checkpoints_req | 2
checkpoint_write_time | 5074064
checkpoint_sync_time | 250
buffers_checkpoint | 93237
buffers_clean | 622
maxwritten_clean | 0
buffers_backend | 18716
buffers_backend_fsync | 0
buffers_alloc | 21328
```

```
select backend_type name, sum(writes) buffers_written , sum( write_time ) write_time , sum( writebacks )
writebacks , sum( writeback_time ) writeback_time , 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;
 name | buffers_written | write_time | writebacks | writeback_time | evictions | fsyncs | fsync_time
------------------+-----------------+------------+------------+----------------+-----------+--------+------------
client backend | 0 | 0 | 0 | 0 | 370 | 0 | 0
autovacuum worker | 0 | 0 | 0 | 0 | 57 | 0 | 0
background writer | 622 | 6.401 | 576 | 83.795 | | 0 | 0
checkpointer | 93035 | 1386.3310 | 93028 | 323.601 | | 304 | 215.698
```


Performance pg_stat_io

The view contains statistics for each process. The rows are duplicated because of the object and context columns. The object column can have values: relation (a persistent storage object) and temp relation (a temporary object). The context column can contain the following values: normal , bulkread , bulkwrite , vacuum . The last three are buffer rings. To reduce the number of rows in the result, it is convenient to use grouping:

```
select backend_type name, sum(writes) buffers_written , sum( write_time ) write_time , sum( writebacks ) writebacks , sum( writeback_time ) writeback_time , 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;
```

```
 name |buf_written|write_time|writebacks|writeback_time|evictions|fsyncs|fsync_time
 -----------------+-----------+----------+----------+--------------+---------+------+----------
 client backend | 0 | 0 | 0 | 0 | 370 | 0 | 0
 autovacuum worker| 0 | 0 | 0 | 0 | 57 | 0 | 0
 background writer| 622 | 6.401| 576 | 83.795 | | 0 | 0
 checkpointer | 93035 | 1386.331| 93028 | 323.601 | | 304 | 215.698
```

To avoid complicating the query, empty values are not filtered and `object='temp relation' is not excluded .`

```
select * from pg_stat_bgwriter \ gx
-[ RECORD 1 ]------ ------------------- ---+--------
checkpoints_timed     ( num_timed )     | 151
checkpoints_req       ( num_requested ) | 2
checkpoint_write_time ( write_time )       | 5074064
checkpoint_sync_time  ( sync_time )        | 250
buffers_checkpoint    ( buffers_written )  | 93237
buffers_clean                            | 622
maxwritten_clean                         | 0
buffers_backend                          | 18716
buffers_backend_fsync                    | 0
buffers_alloc                            | 21328
```

In version 17, statistics on server processes `buffers_backend` and `buffers_backend_fsync` from `pg_stat_bgwriter` removed , considering that it can be viewed in `pg_stat_io` . Other columns were renamed and moved to the `pg_stat_checkpointer` view that appeared in version 17 .

https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=74604a37f

Statistics `buffers_backend_fsync` And `fsyncs`

- statistics `pg_stat_bgwriter . buffers_backend_fsync` and `pg_stat_io . fsyncs , pg_stat_io . fsync_time by backend_type = 'client backend'` shows how many fsync calls executed server processes
- if statistics are greater than zero
  - It's worth checking the statistics and parameters of the bgwriter process
  - I/O performance may be low and be a bottleneck
  - if `pg_stat_bgwriter . maxwritten_clean` has a large value, then this indicates that it is worth increasing `bgwriter_lru_maxpages`

Statistics `buffers_backend_fsync` And `fsyncs`

Statistics `pg_stat_bgwriter . buffers_backend_fsync` and `pg_stat_io . fsyncs , pg_stat_io . fsync_time by backend_type = ' client backend '` shows how many fsync calls executed server processes.

During normal operation, processes (except checkpointer) do not perform fsync by data files (data files are located in tablespaces) , and send block identifiers to the shared memory structure of the checkpoiner process and checkpointer sorts blocks by each file and passes writeback calls to the operating system by page ranges, corresponding blocks and fsyncs by data files . **If the checkpointer fails (the memory structure is full), then the process performs a write itself** . Processes initiate writing a block to disk if they need to load a block into a buffer and for this they evict a dirty buffer from the buffer cache. This happens when server processes, parallel processes, and autovacuum worker processes search for a free buffer . It is desirable that processes do not encounter delays, especially if `context=normal, vacuum` . Column values `context= bulkwrite , bulkread` indicate that a ring buffer was used.

If the statistics are greater than zero, then it is worth checking the statistics and parameters of the bgwriter process , perhaps it should work more actively. Also, the I/O performance may be low and be a bottleneck.

bgwriter scans shared pool buffer headers and dirty pages with `usagecount = 0` and `pincount =0` sends to write . The process ensures the need for clean pages in the buffers if they are needed in large quantities. If `pg_stat_bgwriter . maxwritten_clean` has a large value, then this indicates that b g writer is slow and it is worth increasing `bgwriter_lru_maxpages` , so that bgwriter in one cycle was able to clear more blocks. If the value is relatively large, then reduce `bgwriter_delay` .

Presentation strings `pg_stat_io`

```
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 ex, round( extend_time ::numeric,2) et, evictions ev ,
reuses ru , fsyncs fs , round( fsync_time ::numeric,2) fst from pg_stat_io ;
    backend_type | object | context | w | wt | wb | wbt | ex | et | ev | ru | fs | fst
-------------------+--------+-----------+---+----+----+-----+----+-------+----+----+----+-------
autovacuum launcher | relation | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
autovacuum launcher | relation | normal | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | 0.00
autovacuum worker | relation | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
autovacuum worker | relation | normal | 0 | 0.00 | 0 | 0.00 | 3 | 0.085 | 0 | | 0 | 0.00
autovacuum worker | relation | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
client backend | relation | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
client backend | relation | bulkwrite | 0 | 0.00 | 0 | 0.00 | 6 | 0.136 | 0 | 0 | |
client backend | relation | normal | 0 | 0.00 | 0 | 0.00 | 165 | 1.967 | 0 | | 0 | 0.00
client backend | relation | vacuum | 24 | 0.24 | 0 | 0.00 | 0 | 0 | 0 | 24 | |
client backend | temp relation | normal | 50 | 0.39 | | | 151 | 1.567 | 51 | | |
background worker | relation | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
background worker | relation | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
background worker | relation | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
background worker | relation | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
background worker | temp relation | normal | 0 | 0.00 | | | 0 | 0 | 0 | | |
background writer | relation | normal | 0 | 0.00 | 0 | 0.00 | | | | 0 | 0.00
checkpointer | relation | normal | 94 | 4.76 | 94 | 2.70 | | | | 62 | 160.88
standalone backend | relation | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
standalone backend | relation | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
standalone backend | relation | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
standalone backend | relation | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
startup | relation | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
startup | relation | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
startup | relation | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
startup | relation | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
walsender | relation | bulkread | 0 | 0.00 | 0 | 0.00 | | | 0 | 0 | |
walsender | relation | bulkwrite | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
walsender | relation | normal | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | | 0 | 0.00
walsender | relation | vacuum | 0 | 0.00 | 0 | 0.00 | 0 | 0 | 0 | 0 | |
walsender | temp relation | normal | 0 | 0.00 | | | 0 | 0 | 0 | | |
(30 rows)
```


Lines performances `pg_stat_io`

lines refer to the types of processes for which statistics are collected. The process names are duplicated because statistics are collected separately for two types of objects - permanent and temporary, and for contexts. The normal context is normal work with a block in the shared buffer cache and local cache (for temporary objects). The local buffer does not use buffer rings, so the lines with object=' temp_relation ' only have context='normal'.

The view has a column op_bytes , its value is always equal to the size of the data block (8192), so there is no point in displaying it in queries.

evictions is filled with any context - both rings and normal . evictions means that a block in the buffer was replaced.

reused is filled only for rings.

What is the difference between evictions and reused when working with rings?

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 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.

Example of a view query:

```
select backend_type , context, writes, round( write_time ::numeric,2) wt,
writebacks wb , round( writeback_time ::numeric,2) wbt , extends ex, round(
extend_time ::numeric,2) et, hits, evictions ev , reuses, fsyncs fs , round(
fsync_time ::numeric,2) fst from pg_stat_io where writes>0 or extends>0 or
evictions>0 or reuses>0 or hits>0;
   backend_type |context | writes|write_time | wb | wbt | ex | et | hits | ev | reuses|fsyncs | fst
----------------+--------+------+----------+----+-----+----+----+--------+----+-------+------+---------
autovacuum worker|normal | 4| 83.46 | 4| 0.06| 48| 0.59| 77932| 6| | 4| 73.67
autovacuum worker|vacuum | 3| 0.07 | 0| 0.00| 0| 0.00| 26747| 0| 3| |
client backend | bulkread | 0| 0.00 | 0| 0.00| | | 0| 320|723283| |
client backend |normal | 8024|126050.50 | 8024| 203.95|7537|57465.56|4019507|29556| | 12761|179428.24
background worker|bulkread | 0| 0.00 | 0| 0.00| | | 0| 640|356537| |
background worker|normal | 203| 6.51 | 203| 110.04| 0| 0.00| 9191| 1737| | 0| 0.00
background writer|normal | 17675|250172.10 |17675|4937.16| | | | | | 13779|228832.52
checkpointer |normal | 12250| 235.43 |12250|1853.66| | | | | | 33033|272578.26
(8 rows)
```

Characteristics `pg_stat_io`

- the main columns of interest are writebacks , writeback_time fsyncs , fsync_time since they are the ones that cause system calls to be sent to the operating system
- writebacks are performed:
  - server processes when expanding files (extends) and searching for a buffer for displacement
  - bgwriter processes and checkpointer , which only work in context normal
- fsyncs
  - the checkpointer process to be executed at the end of the checkpoint once for each file
  - if they cannot be transferred (there is no space in the memory structure that checkpointer clears), then they are executed by the process itself
  - at fsyncs rings do not count
- at bgwriter and checkpointer in columns reads, hits, evictions are always zero
- at bgwriter and autovacuum launcher in the extends column zeros

Characteristics `pg_stat_io`

writeback_time columns and fsync_time are filled only if the configuration parameter track_io_timing = on.

Writebacks and fsyncs are not performed on temporary objects , since temporary tables do not need a write guarantee.

reuses are non-empty only for rings, but are absent for normal .

At fsyncs rings do not count.

bulkread has no extends because files do not change size when read.

The autovacuum launcher and autovacuum worker processes do not work with bulkwrite rings . The autovacuum launcher process does not work with vacuum rings .

bgwriter processes and checkpointer with buffer rings do not work.

At bgwriter and checkpointer in columns reads, hits, evictions are always zero .

At bgwriter and autovacuum launcher have zeros in the extends column because they don't increase file sizes.

By process loggingg collector statistics are not collected because the loggingg process collector 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 Receiver and WAL Writer processes are missing from pg_stat_io (not implemented) .

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) the bgwriter (BgBufferSync () function) and checkpointer (BufferSync () function) processes in the normal context . Both functions call SyncOneBuffer (..) and use the checkpointer process queue by calling the ScheduleBufferTagForWriteback (..) function.

fsyncs execution rule by processes : If there is a local pending-ops table, just make an entry in it for ProcessSyncRequests to process later. Otherwise, try to pass off the fsync request to the checkpointer process. If that fails (there is no space in the memory structure that the checkpointer cleans up) , just do the fsync locally before returning (we hope this will not happen often enough to be a performance problem).

Presentation statistics `pg_stat_io`

- `Evictions` how many times the process cleared the buffer for a new block
  - whether the buffer was dirty or clean is not reflected
- `context= bulkwrite,bulkread` means that a ring buffer was used
- `extends, extend_time` - time spent on increasing relations files
- `buffers_alloc` how many times processes loaded blocks into a buffer when not using a buffer ring
- if there are no free buffers in the buffer cache , then `buffers_alloc` corresponds to `evictions`

```
select backend_type , context, reads, trunc ( read_time ) r_time , writes, trunc ( write_time ) w_time , extends,
extend_time , hits, evictions, reuses from pg_stat_io where reads<>0 or writes<>0 or extends<>0 or hits<>0 or evictions<>0;
    backend_type | context | reads | r_time | writes | w_time | extends | extend_time | hits | evictions | reuses
-----------------+---------+-------+--------+--------+--------+---------+-------------+------+-----------+-------
 autovacuum launcher | normal | 1 | 6 | 0 | 0 | | | 590 | 0 |
 autovacuum worker | normal | 217 | 310 | 0 | 0 | 2 | 0.038 | 799993 | 101 |
 autovacuum worker | vacuum | 47889 | 305 | 13 | 0 | 0 | 0 | 1636676 | 702 | 47171
 client backend | normal | 2420 | 2573 | 0 | 0 | 30914 | 764.359 | 459710941 | 12675 |
 client backend | vacuum | 1 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0
 background writer | normal | | | 2751 | 21 | | | | |
 checkpointer | normal | | | 169856 | 2338 | | | | |
```


Presentation statistics `pg_stat_io`

In the example, the statistics for server processes in `pg_stat_io` are zero, besides `evictions` , in `pg_stat_bgwriter . buffers_backend` is non-zero. Processes do not write themselves, they pass blocks for writing to the `checkpointer process` , except when using a buffer ring.

If a process (such as a server or autovacuum worker process) clears the buffer for a new block, `evictions is incremented` . Whether the buffer was dirty or clean is not reflected.

`context= bulkwrite,bulkread` means that a ring buffer was used.

`extends, extend_time` - time spent on increasing relations files.

in the `object column the relation` values or `temp relation` and you can group the result by obtaining data by temporary tables and indexes.

Performance is reduced by file extensions . In the example, 764 milliseconds were spent expanding files, and 2571 milliseconds were spent reading blocks.

`pg_stat_bgwriter.buffers_alloc` is not related to `bgwriter` , but these statistics were left in the view and not even removed. bgwriter does not analyze this statistic, although the purpose of this statistic was: "count buffer allocation requests so that the bgwriter can estimate the rate of buffer consumption".

`numBufferAllocs` statistic corresponding `to buffers_alloc` is increased when calling the `StrategyGetBuffer (..) function` if the buffer ring is not used. In this case, the function will return a buffer into which the block can be loaded. **If there are no free buffers in the buffer cache , then `buffers_alloc` corresponds to `evictions` .** If there are free buffers, they are taken from the free list, the statistic is increased, and evictions is not increased:

```
select backend_type , context, evictions from pg_stat_io where evictions<>0;
    backend_type | context | evictions
-------------------+---------+-----------
 autovacuum worker | normal | 60758
 autovacuum worker | vacuum | 23958
client backend | normal | 3259501
select buffers_alloc from pg_stat_bgwriter ;
      3365169
select * from pg_buffercache_summary ();
 buffers_used | buffers_unused | buffers_dirty | buffers_pinned | usagecount_avg
--------------+----------------+---------------+----------------+----------------
16384 | 0 | 642 | 2 | 4.1064453125
```

Presentations `pg_statio_all_tables and pg_statio_all_indexes`

- in `pg_statio_all_tables` statistics of reading table blocks, all indexes on this table, TOAST of the table and its index
- in `pg_statio_all_indexes` statistics for a specific index
  - " hit " means that the block was in the buffer cache.
  - " read " means that the 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.

```
 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 3;
 name | tab read | tab hit | idx read | idx hit | toast read | toast hit
------------------------+----------+----------+----------+----------+------------+-----------
  public.pgbench_accounts | 22702740 | 82951068 | 479143 | 36142117 | |
  pg_catalog.pg_statistic | 193579 | 2042126 | 2792 | 41643 | 5 | 1
  public.pgbench_history | 148559 | 5882517 | | | |
 select schemaname ||'.'|| relname table, indexrelname index, idx_blks_read idxread , idx_blks_hit idxhit from
pg_statio_all_indexes order by 4 desc limit 3;
 table | index | idx read | idx hit
------------------------+----------------------+----------+----------
  public.pgbench_accounts | pgbench_accounts_pkey | 483328 | 36178084
  public.pgbench_tellers | pgbench_tellers_pkey | 1128 | 13472177
  public.pgbench_branches | pgbench_branches_pkey | 1129 | 9042373
```


Presentations `pg_statio_all_tables and pg_statio_all_indexes`

These views contain statistics on reading table blocks, all indexes on this table, the TOAST table and its index (TOAST is always accessed through the TOAST index and therefore the data on the TOAST index is proportional to the data on the TOAST table) with loading from disk (columns `*_ blks_read`) and from the buffer cache (columns `*_ blks_hit`) .

`pg_statio_all_tables` view contains statistics for all indexes on a table. Statistics (reading with loading from disk and from the buffer cache) for a specific index can be viewed in the `pg_statio_all_indexes view` .

" **`hit`** " means that the block was in the buffer cache .

" **`read`** " means that the block was not in the buffer cache . The process had to clear the buffer (if the free list is empty), load the block into the cleared buffer, and only then read its contents.

Performance pg_stat_all_tables

- contains detailed statistics on tables and indexes:

```
 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 1 ;
 name | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
------------------------+----------+----------+---------------+------------------+-------------------
  public.pgbench_accounts | 0 | 11183162 | 11183162 | 1512 | 266
  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 1 ;
  name | ins | upd | del | hot_upd | newblock | live | dead | sv | sa
------------------+-----+---------+-----+---------+----------+--------+---------+----+------
  pgbench_tellers | 0 | 5598056 | 0 | 5497197 | 100859 | 10 | 1456051 | 0 | 165
```

- example of pgbench test if horizon was held and stopped holding:

```
progress: 92100.0 s,   9.4 tps , lat 105.960 ms stddev 4.277, 0 failed
progress: 922 0 0.0 s, 556.1 tps , lat  1.796 ms stddev 2.213, 0 failed
 name | ins | upd | del | hot_upd | newblock | live | dead | iv | ma
------------------+-----+---------+-----+---------+----------+--------+------+----+------
 pgbench_tellers | 0 | 5762566 | 0 | 5656271 | 106295 | 10 | 124 | 0 | 1201
```

- performance `pg_stat_xact_all_tables` has the same columns as `pg_stat_all_tables` , but shows only actions performed in the current transaction and not yet in `pg_stat_all_ * .` Columns for **`n_live_tup` , `n_dead_tup`** , and those related to vacuuming and analysis are missing

Performance pg_stat_all_tables

Detailed statistics by tables:

```
 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 3;
 name | seq_scan | idx_scan | idx_tup_fetch | autovacuum_count | autoanalyze_count
------------------------+----------+----------+---------------+------------------+-------------------
  public.pgbench_accounts | 0 | 11183162 | 11183162 | 1512 | 266
  public.pgbench_tellers | 906731 | 4684850 | 4684850 | 1524 | 1536
  public.pgbench_branches | 907256 | 4684327 | 4684327 | 1527 | 1536
 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 3;
 name | ins | upd | del | hot_upd | newblock | live | dead | sv | sa
------------------+-----+---------+-----+---------+----------+--------+---------+----+------
  pgbench_tellers | 0 | 5598056 | 0 | 5497197 | 100859 | 10 | 1456051 | 0 | 165
  pgbench_branches | 0 | 5598056 | 0 | 5589787 | 8269 | 1 | 1456044 | 0 | 175
  pgbench_accounts | 0 | 5598056 | 0 | 3923068 | 1674988 | 100001 | 1456032 | 0 | 7619
```

pgbench test if you hold the horizon and then move it:

after vacuum exhaustion the number `n_dead_tup` will decrease from 1456051 to the minimum, and TPS will increase from 9 to 556 :

```
progress: 92100.0 s,   9.4 tps , lat 105.960 ms stddev 4.277, 0 failed
progress: 922 0 0.0 s, 556.1 tps , lat  1.796 ms stddev 2.213, 0 failed
name | ins | upd | del | hot_upd | newblock | live | dead | iv | ma
------------------+-----+---------+-----+---------+----------+--------+------+----+------
 pgbench_tellers | 0 | 5762566 | 0 | 5656271 | 106295 | 10 | 124 | 0 | 1201
 pgbench_branches | 0 | 5762566 | 0 | 5753853 | 8713 | 1 | 185 | 0 | 1201
 pgbench_accounts | 0 | 5762566 | 0 | 4055493 | 1707073 | 100000 | 5687 | 0 | 1740
```

Statistics **`n_tup_hot_upd`** is not updated by vacuum. This and other accumulated statistics should be monitored dynamically, for example, using graphs. After the horizon was cleared, the share of HOT update increased from 14% to 54%.

View pg_stat_xact_all_tables has the same columns as pg_stat_all_tables , but shows only actions performed in the current transaction so far and not yet in pg_stat_all_ *. The columns for **`n_live_tup` , `n_dead_tup`** , and those related to vacuuming and analysis are missing from these views :

```
 select schemaname ||'.'|| relname name, seq_scan , idx_scan , idx_tup_fetch , n_tup_ins ins, n_tup_upd upd ,
n_tup_del del, n_tup_hot_upd hot_upd , n_tup_newpage_upd newblock from pg_stat_xact_all_tables where idx_scan is
not null order by 3 desc limit 3;
 name | seq_scan | idx_scan | idx_tup_fetch | ins | upd | del | hot_upd | newblock
------------------------+----------+----------+---------------+-----+-----+-----+---------+----------
 pg_catalog.pg_namespace | 1 | 2 | 0 | 0 | 0 | 0 | 0 | 0
```

Performance pg_stat_all_indexes

- contain detailed statistics on tables and indexes:

```
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 3;
 name | indexrelname | idx_scan | scan_sec | idx_tup_read | idx_tup_fetch
------------------------+----------------------+----------+----------+--------------+---------------
 public.pgbench_accounts | pgbench_accounts_pkey | 7080442 | 19489 | 20471311 | 7080442
 public.pgbench_tellers | pgbench_tellers_pkey | 2799175 | 19489 | 3740278460 | 124998
 public.pgbench_branches | pgbench_branches_pkey | 2798720 | 19489 | 3049729576 | 2798720
```

- idx_scan number of scans of this index
- last_idx_scan the time of the last scan of this index
- idx_tup_read number of index records returned by scans of this index
- idx_tup_fetch actual table rows fetched using a simple scan of this index (Index Scan)

Performance pg_stat_all_indexes

Detailed statistics on indices:

```
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 3;
 name | indexrelname | idx_scan | scan_sec | idx_tup_read | idx_tup_fetch
------------------------+----------------------+----------+----------+--------------+---------------
 public.pgbench_accounts | pgbench_accounts_pkey | 7080442 | 19489 | 20471311 | 7080442
 public.pgbench_tellers | pgbench_tellers_pkey | 2799175 | 19489 | 3740278460 | 124998
 public.pgbench_branches | pgbench_branches_pkey | 2798720 | 19489 | 3049729576 | 2798720
```

idx_scan The number of scans of this index. Helps determine which indexes are used frequently and which are rarely or never used.

last_idx_scan is the time this index was last scanned . This column was introduced in PostgreSQL version 16. It displays the end time of the transaction that used the index, not the exact time the index was used (which can vary significantly if it is a long transaction). By periodically querying the last use time and comparing it with idx_scan , you can identify rarely used indexes to remove.

Unused indexes take up space comparable to the table size ; slow down DML operations ; interfere with HOT updates.

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) .

Indexes can be used with Index Scan , with Bitmap Index Scan and scheduler. Bitmaps built on several indices can be combined by AND and OR operations. When Bitmap Index Scan increases the pg_stat_all_indexes.idx_tup_read counters of the used indexes and the pg_stat_all_tables.idx_tup_fetch counter for each tables, and pg_stat_all_indexes.idx_tup_fetch does not change.

the idx_tup_read and idx_tup_fetch counters may differ even if a bitmap scan is not used, because idx_tup_read counts the items retrieved from the index, while idx_tup_fetch counts the number of live rows fetched from the table. The difference will be smaller if dead or not yet committed rows are retrieved using the index, or if Index is used to retrieve table rows. Only Scan .

Database horizon retention duration

- duration of the longest query or transaction of all databases in the cluster:

```
 select datname database, max(now()- xact_start ) duration, greatest(max(age( backend_xmin )), max(age(
backend_xid ))) age from pg_stat_activity where backend_xmin is not null or backend_xid is not null group
by datname order by datname ;
 database | duration | age
----------+----------+-------
 postgres | 00:09:41.946193 | 79348
```

```
 select now()- xact_start duration, age( backend_xmin ) age_xmin , age( backend_xid ) age_xid , pid ,
datname , 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 ;
 duration | age_xmin | age_xid | pid | datname | usename | state | wait_event | query
----------+----------+---------+------+----------+----------+---------------------+------------+-------
 00:02:52 | 46732 | | 8806 | postgres | postgres | idle in transaction | ClientRead | select
 00:00:00 | | 1 | 8825 | postgres | postgres | active | WALSync | END;
 00:00:00 | 1 | | 9049 | postgres | postgres | active | |
```

- you can identify queries or transactions that are holding up the horizon and optimize them or move queries to replicas

Database horizon retention duration

The horizon is critical to performance because it determines whether autovacuum and HOT cleanup can clean up old row versions . You can identify queries or transactions that are holding up the horizon and optimize them or move queries to a replica.

Duration of the longest query or transaction of all databases in the cluster:

```
select datname database, max(now()- xact_start ) duration, greatest(max(age( backend_xmin )), max(age( backend_xid ))) age from pg_stat_activity where backend_xmin is not null or backend_xid is not null group by datname order by datname ;
 database | duration | age
----------+----------+-------
 postgres | 00:09:41.946193 | 79348
```

age - the number of transaction numbers that are distant from the current one. Horizon-keeping processes:

```
select now()- xact_start duration, age( backend_xmin ) age_xmin , age( backend_xid ) age_xid , pid , datname , usename , state, wait_event , left(query, 20) 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 ;
 duration | age_xmin | age_xid | pid | datname | usename | state | wait_event | query
----------+----------+---------+------+----------+----------+---------------------+------------+-------
 00:02:52 | 46732 | | 8806 | postgres | postgres | idle in transaction | ClientRead | select
 00:00:00 | | 1 | 8825 | postgres | postgres | active | WALSync | END;
 00:00:00 | 1 | | 9049 | postgres | postgres | active | |
```

If the query is executed on the master, then the data is given on the master processes. If the query is executed on the replica, then on the replica processes.

Performance `pg_replication_slots` contains the state of all replication slots. The `xmin column` contains the ID of the oldest transaction for which the horizon should be held. Example query:

```
select max ( age ( xmin )) from pg_replication_slots ;
```

`pg_stat_replication` view on the master contains one row for each `walsender` . The column `backend_xmin` contains the oldest transaction ID (" xmin ") of the replica if feedback is enabled (`hot_standby_feedback= on`) . Example query:

```
select age( backend_xmin ) age_xmin , application_name from pg_stat_replication order by age( backend_xmin ) desc ;
```

Performance pg_stat_wal

- V one line representation
- wal_records how many log records are written to WAL files
- wal_fpi how many full page images were written (full page images)
- wal_buffers_full how many times was a write to the logs caused by the log buffer being full
- linux page cache XLogWrite function
- if wal_sync_method = fdatasync , fsync or fsync_writethrough , then the XLogWrite function calls the function issue_xlog_fsync to flush blocks related to log files from the page cache to disk
- wal_write_time columns and wal_sync_time have values different from zero only if the parameter track_wal_io_timing = on

```
select * from pg_stat_wal ;
 wal_records | wal_fpi | wal_bytes | wal_buffers_full | wal_write | wal_sync |...
-------------+---------+------------+------------------+-----------+----------+
33373424 | 1158848 | 10892483009 | 0 | 5736124 | 5730955 |
```


Performance pg_stat_wal

IN one line representation .

wal_records how many log records are written to WAL files.

wal_fpi how many full page images (full page images) are written to WAL files.

wal_buffers_full The number of times that a write to the log files was caused because the log buffer was full.

wal_write How many times blocks from the log buffer were flushed to the page cache by the XLogWrite function

wal_sync (and wal_sync_time - time spent) how many times journal entries flushed to disk by the issue_xlog_fsync function if the fsync parameter = on and wal_sync_method fdatasync , fsync matters (and fsync_writethrough , which is not used in linux). If the wal_sync_method parameter = open_datasync or open_sync , then issue_xlog_fsync is not used and the wal_sync column has a value of zero.

Journal buffers are flushed to linux page cache function XLogWrite , which is called:

XLogInsertRecord function when there is no room in the WAL buffers for new records

XLogFlush function

walwriter process

If wal_sync_method = fdatasync , fsync or fsync_writethrough , the XLogWrite function calls the function issue_xlog_fsync to flush blocks related to log files from the page cache to disk .

wal_write_time columns and wal_sync_time have values different from zero only if the parameter track_wal_io_timing = on .

Extension pg_walinspect

- alternative to command line utility pg_waldump
- pg_get_wal_stats (lsn1 , lsn2 , true) returns log record statistics
  - If the third parameter is true , then statistics are also displayed on record_type , not the total for each resource_manager
- In the example, the full block images (fpi_size) occupy: 83% for tables and 76% for indexes . According to pgbench test tps=14.

```
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 ) ) ) .
 type | count | count% | record_size | size% | fpi_size | fpi % | combined_size | total%
----------------------+-------+--------+-------------+-------+----------+------+---------------+--------
  Heap / LOCK | 16223 | 15 | 1003405 | 11 | 48582684 | 81 | 49586089 | 71
  Btree / INSERT_LEAF | 16224 | 15 | 1049301 | 11 | 10692280 | 18 | 11741581 | 17
 Heap/ UPDATE | 15822 | 14 | 2865744 | 31 | 12620 | 0 | 2878364 | 4
 Heap/ HOT_UPDATE | 29779 | 27 | 2560740 | 27 | 5432 | 0 | 2566172 | 4
 Heap/ INSERT | 15215 | 14 | 1232365 | 13 | 7620 | 0 | 1239985 | 2
 XLOG/ FPI_FOR_HINT | 89 | 0 | 4539 | 0 | 719100 | 1 | 723639 | 1
 Transaction/ COMMIT | 15331 | 14 | 553376 | 6 | 0 | 0 | 553376 | 1
 ...
```


Extension pg_walinspect

Standard extension for viewing log records. Introduced in PostgreSQL version 15. Works with a running instance. Outputs data on the current timeline. The functionality is similar to the pg_waldump command line utility , but works via SQL . Available to superusers and the pg_read_server_files role . Installing the extension:

```
create extension pg_walinspect ;
```

The extension consists of four functions. You can see their list with the command:

```
\ dx + pg_walinspect
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 )
```

Function pg_get_wal_stats (lsn1,lsn2 , true) shows statistics on log records between two LSNs.

If the third parameter is true , then statistics are also given for record_type , and not the total for each resource_manager . The second LSN can be specified as maximum . Example of a pgbench test with horizon retention:

```
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 ))). 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%
---------+-------+--------+-------------+-------+----------+------+---------------+--------
Heap/LOCK | 16223 | 15 | 1003405 | 11 | 48582684 | 81 | 49586089 | 71
Btree /INSERT_LEAF | 16224 | 15 | 1049301 | 11 | 10692280 | 18 | 11741581 | 17
Heap/UPDATE | 15822 | 14 | 2865744 | 31 | 12620 | 0 | 2878364 | 4
...
```

Full block images (fpi_size) take up significant space: 69668516/83897623 = 83% for tables and 9634496/12734954= 76% for indexes .

resource_manager list can be viewed by request:

```
select * from pg_get_wal_resource_managers ();
```

Using the extension `pg_walinspect`

- if the horizon is not held, then the HOT share becomes larger:

```
type | count | count% | record_size | size% | fpi_size | fpi % | combined_size | total%
---------------------+---------+--------+-------------+-------+----------+------+---------------+--------
Heap/ HOT_UPDATE | 1125325 | 54 | 96690453 | 62 | 28860128 | 47 | 125550581 | 58
Heap/INSERT | 372681 | 18 | 30187111 | 19 | 8764 | 0 | 30195875 | 14
Heap2/PRUNE | 131095 | 6 | 11757645 | 8 | 4885592 | 8 | 16643237 | 8
Heap2/VISIBLE | 58699 | 3 | 3822780 | 2 | 11975568 | 20 | 15798348 | 7
XLOG/FPI_FOR_HINT | 1929 | 0 | 98379 | 0 | 15082820 | 25 | 15181199 | 7
Transaction/COMMIT | 375127 | 18 | 13509524 | 9 | 0 | 0 | 13509524 | 6
...
```

- TPS increases from 14 to 660
- the closer the horizon is to the current moment and the smaller the FILLFACTOR , the more likely it is that the HOT share will increase
- does reducing these values give an effect? You can ask by paying attention to the share Heap/ HOT_UPDATE
- HOT is possible if:
  - changes do not affect indexed columns
  - the new row version must fit in the same block as the row version being updated

Using the extension `pg_walinspect`

In the previous example TPS is small:

```
progress: 36590.0 s, 13.9 tps, lat 71.833 ms stddev 5.460, 0 failed
```

If the horizon is not held, the HOT share becomes larger :

```
type | count | count% | record_size | size% | fpi_size | fpi % | combined_size | total%
---------------------+---------+--------+-------------+-------+----------+------+---------------+-------
Heap/ HOT_UPDATE | 1125325 | 54 | 96690453 | 62 | 28860128 | 47 | 125550581 | 58
Heap/INSERT | 372681 | 18 | 30187111 | 19 | 8764 | 0 | 30195875 | 14
Heap2/PRUNE | 131095 | 6 | 11757645 | 8 | 4885592 | 8 | 16643237 | 8
Heap2/VISIBLE | 58699 | 3 | 3822780 | 2 | 11975568 | 20 | 15798348 | 7
XLOG/FPI_FOR_HINT | 1929 | 0 | 98379 | 0 | 15082820 | 25 | 15181199 | 7
Transaction/COMMIT | 375127 | 18 | 13509524 | 9 | 0 | 0 | 13509524 | 6
...
```

TPS increase from 14 to 660 :

```
progress: 38320.0 s, 660.6 tps, lat 1.511 ms stddev 0.360, 0 failed
```

HOT is possible if:

Indexed columns are not updated, including columns in partial indexes.

The new row version must fit into the same block as the row version being updated. By default, FILLFACTOR=100%, fast cleanup will be triggered if the previous update of any row in the block fit into the block and there is less than 10% left in the block. The new row version must fit into these 10%. If the previous version of any row in the block has gone beyond the horizon, its place will be cleaned and occupied by the new version.

If the row size in the table is less than 5-10% of the block size, then decreasing the FILLFACTOR will not increase the HOT percentage. If the row size is greater than 10% of the block size, then decreasing FILLFACTOR can increase the HOT share.

Conclusion: the closer the horizon is to the current moment and the smaller the FILLFACTOR , the more likely it is that the HOT share will increase. Whether the reduction of these values has an effect can be done by asking by paying attention to the Heap/HOT_UPDATE share .

Updating indexes when updating table rows is called "index write amplification". It is not related to "WAL write amplification".

https://www.adyen.com/knowledge-hub/postgresql-hot-updates

Performance `pg_stat_activity`

- one line for each instance process
- the lines reflect what the process is doing at the time the line is accessed
- `backend_type` - name of the process
- state current state of the process
- `wait_type = 'Activity'` , this is the normal state of the process, it is not blocked, idle and waiting to wake up in its main (Main) cycle
- `wait_type = 'Lock'` , then the process is waiting to receive a lock

```
select left( backend_type , 14) type, left(state,6) state, wait_event_type wait_type , wait_event , age(
backend_xid ) age_xid , age( backend_xmin ) age_xmin , round(extract('epoch' from clock_timestamp ()
xact_start )) sec, substring(query,0, 20) query from pg_stat_activity ;
 type | state | wait_type | wait_event | age_xid | age_xmin | sec | query
--------------+--------+-----------+--------------------+---------+----------+------+--------------------
autovacuum lau | | Activity | AutoVacuum Main | | | |
pg_wait_sampli | | Extension | Extension | | | |
client backend | idle i | Client | ClientRead | 92700 | | 1010 | lock table t;
client backend | active | Lock | relation | | 92700 | 967 | select * from t lim
client backend | active | | | 1 | 92700 | 0 | UPDATE pgbench_bran
...
```


Performance `pg_stat_activity`

In the pg_stat_activity view for each instance process there will be one line with information about what the process is doing when the view is accessed. It might be worth filtering out server processes that are not connected to the current database: `datname = current_database ()`

Column backend_type contains the name of the process. Example: client backend , autovacuum worker, checkpointer, background writer, startup, walreceiver , walsender .

Columns client_addr , client_hostname , client_port , application_name , username (user name in session) , usesysid (User OID) allows you to identify the client whose session is being serviced by the server process.

state is the current state of the process. Values:

active: The server process is executing a command. Spikes at certain times indicate synchronous logic in the application. If wait_event is not empty, it means that the process is blocked by something and is waiting for the block to be released.

idle: The server process is not in a transaction and is waiting for a command . A decrease compared to the normal value may indicate a shortage of connections in the pool and a long wait for new server processes to start to expand the connection pool. A monotonic increase may indicate a connection leak in the client connection pool.

idle in transaction: the server process has opened a transaction and is idle. There should be no long idle periods in open transactions in the application logic.

idle in transaction (aborted): transaction in a failed state.

fastpath function call: The server process called a function through the fastpath interface . This is a legacy way of calling functions. clients using the libpq library . Calling procedures, group, window functions is not supported by this method. It is recommended to use prepared commands instead of this method of calling.

disabled: `track_activities` parameter `=off`

If wait_type =Activity , this is the normal state of the process, it is not blocked, idle and waiting to wake up in its main (Main) cycle

If wait_type =Lock , the process is waiting to acquire a lock .

Example request:

```
select left( backend_type , 14) type, left(state,6) state, wait_event_type wait_type , wait_event , age(
backend_xid ) age_xid , age( backend_xmin ) age_xmin , round(extract('epoch' from clock_timestamp () - xact_start
)) sec, left(query,20) query from pg_stat_activity ;
```

Blocking processes and function `pg_blocking_pids ()`

- if pg_stat_activity . wait_type ='Lock' or pg_locks.granted ='f' , then the process is blocked, i.e. waiting to receive a lock
- `pg_blocking_pids ( pid )` shows blocking processes
  - the function gets a lightweight lock on all sections of the lock structure for a short time, so it should not be called often

```
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|blocker|wait | event | age|waitsec|bl_type | bl_wait | bl_sec | blocked_q | blocker_q
--------+-------+----+--------+---+-------+-------+----------+------+---------------+----------
 755637| 754580|Lock|relation| 1| 815| Client|ClientRead | 1126|select count(*)| select 1;
```

- For monitoring you can use the `pg_locks view` :

```
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 ();
 blocked| mode | wait | event | age_xmin | waitsec | blocked_query
--------+-----------------+------+----------+----------+---------+----------
 1234 | AccessShareLock | Lock | relation | 447502 | 11077 | select * from t limit 1;
```


Blocking processes and function `pg_blocking_pids ()`

If `pg_stat_activity . wait_type = 'Lock'` , then the process is waiting to acquire a lock . LWLock should not appear in `pg_stat_activity` , as these are short-lived (light) locks.

To find blocking processes, use the `pg_blocking_pids ( PID )` function . The function returns an array with the PIDs of blocking processes, or an empty array.

A process blocks another process if:

1) The blocking process holds a lock that conflicts with the lock that the blocked process wants to acquire.

2) is in the waiting queue before the one being blocked.

When using parallel processes, the function returns the PID of the parent server process, not the worker processes. This may result in duplicate PIDs being returned in the array. The function gets a lightweight lock on all sections of the lock structure for a short time , so you shouldn't call it often . In other words, do not engage in " monitoring " . To minimize calls, the following condition has been added to the request: `where a.wait_event_type = 'Lock' and cardinality( pg_blocking_pids (a.pid))>0` . By adding the condition `a.datname = current_database ( )` you can limit the result to sessions of the current database. The command `explain analyze` You can compare the execution time and cost of executing a command with and without a condition.

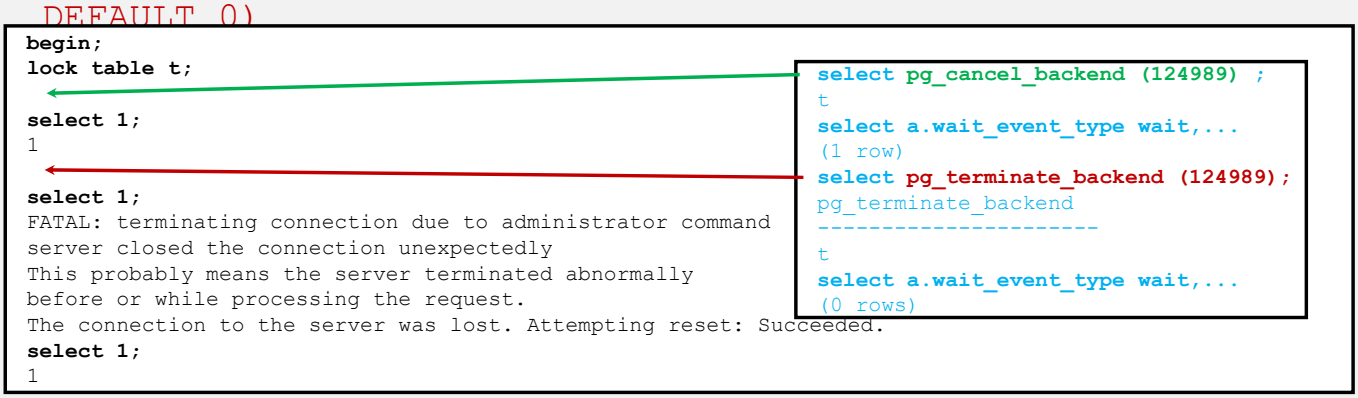
```
select a.pid blocked, bl.pid blocker, a.wait_event_type wait, a.wait_event event, age(
a.backend_xmin ) age, round(extract('epoch' from clock_timestamp () - a.query_start )) waitsec ,
bl.wait_event_type bl_type , bl.wait_event bl_wait , round(extract('epoch' from clock_timestamp () -
bl.query_start )) 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|blocker|wait | event | age|waitsec|bl_type | bl_wait | bl_sec | blocked_q | blocker_q
--------+-------+----+--------+---+-------+-------+----------+------+---------------+----------
 755637| 754580|Lock|relation| 1| 815| Client|ClientRead | 1126|select count(*)| select 1;
```

For monitoring, you can use the `pg_locks view` :

```
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 ();
 blocked| mode | wait | event | age_xmin | waitsec | blocked_query
--------+-----------------+------+----------+----------+---------+----------
 1234 | AccessShareLock | Lock | relation | 447502 | 11077 | select * from t limit 1;
```

pg_cancel_backend () and pg_terminate_backend ()

- You can cancel a command in a blocking session using the pg_cancel_backend (pid) function
- if the session is idle (wait_event = ' ClientRead '), then there is nothing to cancel and the function is useless
- a session and a transaction in a session can be terminated using the pg_terminate_backend function (pid , timeout milliseconds DEFAULT 0)



pg_cancel_backend () and pg_terminate_backend ()

You can cancel a command in a blocking (or arbitrary) session using the pg_cancel_backend (pid) function . You can cancel a command that is running. If the session is idle (wait_event ='ClientRead '), then there is nothing to cancel and the function will return true , but will do nothing.

Usually locks are released at the end of a transaction. Regardless of whether the transaction is open or not, you can terminate a session with the pg_terminate_backend (pid , timeout milliseconds DEFAULT 0) function . The transaction of the terminated session will be interrupted and will not be committed. If the timeout parameter is not specified or is zero, the function will return true without delay if a process with such PID exists. If timeout is greater than zero, the function will return true as soon as the process is terminated . In case of a timeout, a warning is issued and false is returned .

To call both functions, you only need the rights of the user under whom the session with the command was created. The functions are also available to users with the pg_signal_backend role. Only superusers can apply functions to superuser sessions .

PID and session properties are obtained from the pg_stat_activity columns . PID corresponds to the process number in the operating system.

The list of functions for administration can be found in the documentation:
https://docs.tantorlabs.ru/tdb/ru/16_4/se/functions-admin.html

Practice

- Part 1. I/O statistics in the pg_stat_io view
- Part 2. Running fsyncs when checkpointer is stopped
- Part 3. I/O Performance Testing
- Part 4. Choosing the size of temp_buffers when working with temporary tables using pg_stat_io
- Part 5. Example of analysis of statistics of vacuum command operation with a ring
- bgwriter operation and statistics comparison in pg_stat_bgwriter and pg_stat_io views
- Part 7. Running bgwriter on 128MB and 1GB buffer cache
- Part 8. Using the pg_walinspect extension
- Part 9. Monitoring Blocks

Practice

Examples of I/O monitoring and analysis of presentation statistics are discussed in detail. pg_stat_io .
In practice, the checkpointer and bgwriter processes will be suspended. and you will see how this will affect the operation of the instance.
You will compare the performance of an instance with different buffer cache sizes.
pg_walinspect extension will be given. and comparison with the pg_waldump utility .

15

pg_stat_statements and pg_stat_kcache extensions

Extension pg_stat_statements

- detailed instance performance statistics down to SQL commands
- for installation you need to download the library and install the extension:

```
alter system set shared_preload_libraries = pg_stat_statements ;
create extension pg_stat_statements ;
```

- The extension includes 3 functions and 2 views:

```
\ dx + pg_stat_statements
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
```

- commands are combined into a single line in `pg_stat_statements` when they are executed by the same user and have identical query structures, i.e. are semantically equivalent except for literals and substitution variables (literal constants)
- For example: `select * from t where id = 'a'` and `select * from t where id = 'b'` will be combined into `select * from t where id = $1`

Extension pg_stat_statements

Standard extension. Provides detailed statistics of the instance operation with the accuracy of SQL commands . To install you need to download the library and install the extension:

```
alter system set shared_preload_libraries = pg_stat_statements ;
create extension pg_stat_statements ;
```

The extension includes 3 functions and 2 views:

```
\ dx + pg_stat_statements
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
```

The extension collects command execution statistics, grouped by commands.

`compute_query_id` configuration parameter is used to group commands . The parameter value must be auto (the default value) or on.

Commands are combined into a single command in `pg_stat_statements` when they are executed by the same user and have identical structure, i.e. are semantically equivalent except for literals and substitution variables (literal constants) . For example, queries: `select * from t where id = 'a'` and `select * from t where id = 'b'` will be combined into a query: `select * from t where id = $1` . Queries with visually different texts can be combined if they are semantically equivalent. Different commands can be combined due to a hash collision , but the probability of this is low. And vice versa: commands with the same text can be considered different if they received a different parse tree, for example, due to a different `search_path` .

To keep statistics up-to-date, it is worth resetting statistics regularly, for example once a day. You can save data for the previous day. Statistics are reset by calling the `pg_stat_statements_reset ()` function .

Configuration pg_stat_statements

- pg_stat_statements_info view has two fields:
  - dealloc how many times records about rarely executed commands were discarded in pg_stat_statements . The number of commands tracked is specified by the pg_stat_statements.max parameter , by default 5000.
  - stats_reset the moment of the last reset of all extension statistics by calling the pg_stat_statements_reset () function without parameters
- With the function pg_stat_statements_reset (userid , dbid , queryid , minmax_only) you can reset not all, but only part of the statistics:
  - by commands executed by the user, executed in the database, by a separate

```
select pg_stat_statements_reset ();
   pg_stat_statements_reset
------------------------------
20 25 - 0 1- 01 1 1:11:11 . 111111 +03
(1 row)
select * from pg_stat_statements_info ;
 dealloc | stats_reset
---------+------------------------------
0 | 20 25 - 0 1- 01 1 1:11:11 . 111111 +03
(1 row)
```


Configuration pg_stat_statements

pg_stat_statements_info view has one row and two columns:

1) dealloc c is the number of times pg_stat_statements discarded records of rarely executed commands because more distinct commands were received for processing than were specified in the pg_stat_statements.max extension configuration parameter (default 5000). The pg_stat_statements.max parameter specifies the maximum number of commands tracked by the extension, that is, the maximum number of rows in the pg_stat_statements view .

2) stats_reset the moment of the last reset of all extension statistics by calling the pg_stat_statements_reset () function without specifying parameters:

```
select pg_stat_statements_reset ();
2025-01-01 11:11:11.111111+03
select * from pg_stat_statements_info ;
 dealloc | stats_reset
---------+------------------------------
0 | 2025-01-01 11:11:11.111111+03
```

The function returns the reset time in the format timestamp with time zone .

After a full reset in the **pg_stat_statements_info view** The reset moment is updated.

The function can reset not all, but only part of the statistics:

pg_stat_statements_reset (userid , dbid , queryid , minmax_only DEFAULT false)

by commands executed by the user, executed in the database, by a separate request. To do this, you need to pass the OID of the user and / or the cluster database and / or the request identifier.

If minmax_only=true , then only the minimum and maximum query planning and execution time values are reset (the values of the min_plan_time , max_plan_time , min_exec_time , max_exec_time columns). By default, minmax_only=false . The time of the last reset of the minimum/maximum values is displayed in the minmax_stats_since column of the pg_stat_statements view .

Configuration parameters `pg_stat_statements`

- `pg_stat_statements.max` specifies the maximum number of statements tracked by the extension, that is, the maximum number of rows in the `pg_stat_statements view`.
- `pg_stat_statements.save` determines whether statistics should be saved across server reboots
- `pg_stat_statements.track` determines which commands will be tracked: top only top-level commands are tracked
- pg_stat_statements.track_planning sets whether planning operations and the duration of the planning phase are tracked
- `pg_stat_statements.track_utility` whether statements other than `SELECT, INSERT, UPDATE, DELETE, MERGE will be tracked`

```
select name, setting, context, min_val , max_val from pg_settings where name like ' pg_stat_statements
%';
 name | setting | context | min_val | max_val
----------------------------------+---------+-----------+---------+------------
pg_stat_statements.max | 5000 | postmaster | 100 | 1073741823
 pg_stat_statements.save | on | sighup | |
 pg_stat_statements.track | top | superuser | |
 pg_stat_statements.track_planning | on | superuser | |
 pg_stat_statements.track_utility | on | superuser | |
```


Configuration parameters `pg_stat_statements`

```
select name, setting, context, min_val , max_val from pg_settings where name like ' pg_stat_statements %';
name | setting | context | min_val | max_val
----------------------------------+---------+-----------+---------+------------
pg_stat_statements.max | 5000 | postmaster | 100 | 1073741823
 pg_stat_statements.save | on | sighup | |
 pg_stat_statements.track | top | superuser | |
 pg_stat_statements.track_planning | on | superuser | |
 pg_stat_statements.track_utility | on | superuser | |
```

Extension configuration parameters:

`pg_stat_statements.max` specifies the maximum number of statements tracked by the extension, that is, the maximum number of rows in the `pg_stat_statements view` . Statistics about rarely executed statements are usually not needed, and there is no need to increase this value, as this increases the amount of shared memory allocated by the extension. The default value is 5000.

`pg_stat_statements.save` determines whether statistics should be saved across server reboots. If the value is `off` , statistics are not saved when the instance is stopped. The default value is `on` , which means statistics are saved when the instance is stopped or restarted.

`pg_stat_statements.track` determines which statements will be tracked. Accepts values:

1) `top` (default value) only top-level commands (passed by clients in sessions) are tracked
2) `all` - in addition to top-level commands, commands inside called functions are tracked
3) `none` - statistics collection is disabled.

`pg_stat_statements.track_planning` Sets whether planning operations and the duration of the planning phase will be monitored. The value is `on` may cause a noticeable performance penalty, especially when multiple sessions execute commands with the same query structure at the same time, resulting in those sessions attempting to modify the same rows in `pg_stat_statements at the same time` . The default value `is off` .

`pg_stat_statements.track_utility` determines whether the extension tracks utility commands. Utility commands are commands other than `SELECT, INSERT, UPDATE, DELETE, MERGE` . The default value `is on` .

Performance `pg_stat_statements`

- the view is filled with the function pg_stat_statements (true)
- The presentation has **49 columns** , 10 of which are related to `jit`
- 5 columns with execution time: total_exec_time , min_exec_time , max_exec_time , mean_exec_time , stddev_exec_time
- calls - how many times the command was executed
- rows - number of rows read or changed
- 3 columns are related to wal : `wal_records , wal_fpi , wal_bytes`
- 6 columns to execution plans
- 6 columns to buffer cache
- 10 columns to temporary files and objects
- 4 columns with team characteristics

```
 select calls, round( total_exec_time ) time, round( stddev_exec_time ) delta, rows, plans, round( total_plan_time
) ptime , shared_blks_hit hit, shared_blks_read read, shared_blks_dirtied dirty, shared_blks_written write,
wal_fpi , wal_bytes , left(query,20) query from pg_stat_statements order by 2 desc limit 10;
 calls | time | delta | rows | plans | ptime | hit | read | dirty | write | wal_fpi | wal_bytes | query
-------+-------+-------+--------+--------+-------+---------+------+-------+-------+---------+-----------+--------
 529554 | 13478 | 0 | 529554 | 529554 | 16851 | 3302629 | 6963 | 34309 | 0 | 38447 | 117746959 | UPDATE
 529554 | 12370 | 0 | 529554 | 529554 | 14759 | 2118690 | 5 | 15 | 0 | 9 | 46780041 | UPDATE
 529554 | 12246 | 0 | 529554 | 529554 | 15316 | 3046701 | 3 | 38 | 0 | 31 | 46927982 | UPDATE
```


Performance `pg_stat_statements`

To view the data collected by the extension, a view of the same name is used. The view is filled with the function of the same name pg_stat_statements (true). If you call a table function with an argument `pg_stat_statements ( false )` , then the function in column query will return null , not the command text . This allows to reduce the amount of data returned and can be convenient for monitoring programs that connect over a network. Why doesn't the monitoring system access the view removing the column from the selection ? When accessing the view, pg_stat_statements (true) is called, the function selects data for the query column and the I/O load is higher than calling pg_stat_statements (false).

The presentation has **49 columns** , 10 of which are related to `jit` .

2 columns `stats_since , minmax_stats_since` the moment of the last statistics reset.

3 columns are related to wal : `wal_records , wal_fpi , wal_bytes` number of log records, page images, number of bytes generated during command execution.

10 columns to temporary files and objects: `local_blks_hit , local_blks_read , local_blks_dirtied , local_blks_written , local_blk_read_time , local_blk_write_time , temp_blks_read , temp_blks_written , temp_blk_read_time , temp_blk_write_time` .

6 columns to the buffer cache : `shared_blks_hit , shared_blks_read , shared_blks_dirtied , shared_blks_written , shared_blk_read_time , shared_blk_write_time` .

6 columns for execution plans: `plans, total_plan_time , min_plan_time , max_plan_time , mean_plan_time , stddev_plan_time` . **They are filled if `pg_stat_statements.track_planning = on` .**

5 columns with execution time: `total_exec_time , min_exec_time , max_exec_time , mean_exec_time , stddev_exec_time` .

`calls` - how many times the command was executed

`rows` - number of rows read or changed

4 columns with command characteristics: `userid` - OID of the user who executed the command ; `dbid` - OID of the database in which the command was executed ; `toplevel = true` (if the parameter `pg_stat_statements.track = top` then always `true`) or false; `queryid` - command identifier, integer of type bigint .

```
 select calls, round( total_exec_time ) time, round( stddev_exec_time ) delta, rows, plans, round( total_plan _
time) ptime , shared_blks_hit hit, shared_blks_read read, shared_blks_dirtied dirty, shared_blks_written write,
wal_fpi , wal_bytes , left(query,20) query from pg_stat_statements order by 2 desc limit 10;
```

Requests for presentation `pg_stat_statements`

- `order by total_exec_time desc limit 10` returns "Top SQL" - the commands that load the instance the most
- `where plans<>0` removes auxiliary commands like : `BEGIN , END`
- `where rows/calls>1000 and toplevel =true` requests, returning a large number of rows to the client
- `temp_blks_written >0` may indicate a lack of work_mem
- `100*( blk_read_time+blk_write_time )/ total_exec_time` the share of I/O in the total query execution time
- `total_plan_time > total_exec_time` the time to create a plan is longer than the execution time

```
 select round( total_exec_time ::numeric, 2) time, calls, round( mean_exec_time ::numeric, 2) mean,
round((100 * total_exec_time / sum( total_exec_time ::numeric) OVER ())::numeric, 2) "% cpu ", left(query,70)
query from pg_stat_statements order by total_exec_time desc limit 10;
 time | calls | mean | % cpu | query
------------+--------+-------+-------+----------------------------------------
 2773963.94 | 227303 | 12.20 | 82.86 | UPDATE pgbench_branches SET bbalance = bbalance + $1 WHERE bid = $2
 556235.76 | 227303 | 2.45 | 16.62 | UPDATE pgbench_tellers SET tbalance = tbalance + $1 WHERE tid = $2
 7922.75 | 227303 | 0.03 | 0.24 | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE aid = $2
 3735.72 | 227302 | 0.02 | 0.11 | INSERT INTO pgbench_history ( tid , bid, aid, delta, mtime ) VALUES ($1,
 3692.31 | 227303 | 0.02 | 0.11 | SELECT abalance FROM pgbench_accounts WHERE aid = $1
```


Requests for presentation `pg_stat_statements`

Examples how can i select data from view .

`order by total_exec_time desc limit 10` returns "Top SQL" - the commands that load the instance the most. Example:

```
 select round( total_exec_time ::numeric, 2) time, calls, round( mean_exec_time
::numeric, 2) mean, round((100 * total_exec_time / sum( total_exec_time
::numeric) OVER ())::numeric, 2) "% cpu ", left(query,40) query from
pg_stat_statements order by total_exec_time desc limit 10;
 time | calls | mean | % cpu | query
------------+--------+-------+-------+----------------------------------------
 2690386.86 | 222903 | 12.07 | 82.87 | UPDATE pgbench_branches SET bbalance = b
 539192.95 | 222904 | 2.42 | 16.61 | UPDATE pgbench_tellers SET tbalance = tb
 7739.33 | 222904 | 0.03 | 0.24 | UPDATE pgbench_accounts SET abalance = a
 3656.78 | 222903 | 0.02 | 0.11 | INSERT INTO pgbench_history ( tid , bid, a
 3617.65 | 222904 | 0.02 | 0.11 | SELECT abalance FROM pgbench_accounts WH
```

`where plans<>0` removes auxiliary commands like : `BEGIN , END`

`order by stddev_exec_time desc` large spread in execution times, possibly suboptimal plans

`where rows/calls>1000 and toplevel =true` requests, returning a large number of rows to the client

`temp_blks_written >0` may indicate a lack of work_mem . **Temporary files are used when joining by hashing Hash Join (except `work_mem` affects the `hash_mem_multiplier parameter`); sorting externl merge; materialization Materialize .**

`local_blks_ *` refers to the local cache used to work with temporary tables.

In `the explain (analyze, buffers) execution plans` , the statistics correspond to the line after the plan node in which temporary files were used:

`Buffers: shared hit= read=, temp read= written=`

`100*( blk_read_time+blk_write_time )/ total_exec_time` the share of I/O in the total query execution time

`total_plan_time > total_exec_time` The time to create a plan is longer than the execution time. Prepared commands must be used .

Example of a view query:

https://habr.com/ru/articles/805813/

Examples of view queries `pg_stat_statements`

- `blk_read_time+blk_write_time` time spent on input-output
- `total_exec_time-blk_read_time-blk_write_time` not for input-output
- `(100* ( blk_read_time+blk_write_time )/ total_exec_time )` percentage of time spent on I/O
- `( total_exec_time ::numeric/calls)` average command execution time
- `(( total_exec_time-blk_read_time-blk_write_time )::numeric/ calls)` average non-I/O time
- `(( blk_read_time+blk_write_time )::numeric/calls)` average input/output time

```
 select ( blk_read_time+blk_write_time )::numeric(20,2) io_time , ( total_exec_time-blk_read_time -
blk_write_time )::numeric(20,2) non_io_time , (100* blk_read_time+blk_write_time / total_exec_time
)::numeric(20,2) " io_time %", ( total_exec_time ::numeric/calls)::numeric(20,2) avg_time , ((
total_exec_time-blk_read_time-blk_write_time )::numeric/calls)::numeric(20, 2) avg_non_io_time , ((
blk_read_time+blk_write_time )::numeric/ calls)::numeric(20, 2) avg_io_time , left(query,20) query from
pg_stat_statements where calls<>0 order by io_time desc limit 10;
  io_time | non_io_time | io_time_percent | avg_time | avg_non_io_time | avg_io_time | query
---------+-------------+-----------------+----------+-----------------+-------------+----------------------
 273.58 | 58.60 | 27343.66 | 332.19 | 58.60 | 273.58 | create extension pg_
 42.74 | -42.64 | 4274.07 | 0.10 | -42.64 | 42.74 | SELECT c.relname , $1
```


Examples of view queries `pg_stat_statements`

Examples of metrics:

`( blk_read_time+blk_write_time )::numeric(20,2)` time spent on input-output

`( total_exec_time-blk_read_time-blk_write_time )::numeric(20,2) non_io_time` not for input-output

`(100* blk_read_time+blk_write_time / total_exec_time )::numeric(20,2) " io_time %"` percentage of time spent on input/output

`(( total_exec_time + total_plan_time )/calls)::numeric(20,2)` average command execution time , also called " response time "

`(( total_exec_time-blk_read_time-blk_write_time )::numeric/calls)::numeric(20,2) avg_non_io_time`

`(( blk_read_time+blk_write_time )::numeric/calls)::numeric(20,2) avg_io_time`

`total_exec_time` statistic is not the total execution time of the command, only the execution phase . The total time includes the planning phase . Usually, the planning phase is less than the execution phase. `* blk _ * _time statistic` include I/O in both the execution phase and the planning phase. In the example, the request `has non_io_time` negative because the request was planned for 38ms , executed for 0.5ms and the time during planning was spent on 37ms reading blocks :

```
select * from pg_stat_statements where queryid =5827178464283574856\ gx
-[ RECORD 1 ]---+----------------------------------------------------
..
query |SELECT a.attname , .. FROM pg_catalog.pg_attribute a WHERE a.attrelid = $2 AND a.attnum >
                |$3 AND NOT a.attisdropped ORDER BY a.attnum
plans |3
total_plan_time |38.33984
min_plan_time |0.19539
max_plan_time |37.909368
mean_plan_time |12.779946666666667
stddev_plan_time|17.769191620608545
calls |3
total_exec_time |0.500417
min_exec_time |0.12470800000000001
max_exec_time |0.18976
mean_exec_time |0.16680566666666669
stddev_exec_time|0.0298081765553607
rows |147
shared_blks_hit |447
shared_blks_read|12
..
blk_read_time |37.228767
```

`total_plan_time` to the formulas , but the query size will increase and results with negative values are more visual.

Examples of view queries:

```
select
sum(calls) as calls,
sum(rows) as rows,
round(sum( total_exec_time )::numeric, 0) as ex_t ,
round((sum( mean_exec_time * calls) / sum(calls))::numeric, 2) as mean_ex_t ,
format('%s-%s', round(min( min_exec_time )::numeric, 2), round(max( max_exec_time
)::numeric, 2)) as minmax_ex_t ,
round(sum( total_plan_time )::numeric, 2) as pl_t ,
round((sum( mean_plan_time * calls) / sum(calls))::numeric, 2) as mean_pl_t ,
format('%s-%s', round(min( min_plan_time )::numeric,2), round(max( max_plan_time
)::numeric, 2)) as minmax_pl_t ,
left(query,10) query
from pg_stat_statements
where calls<>0 and (select datname from pg_database where oid = dbid )= current_database
()
group by userid , dbid , query
order by sum( total_exec_time ) desc
limit 10;
```

```
calls | rows | ex_t | mean_ex_t | minmax_ex_t | pl_t | mean_pl_t | minmax_pl_t | left
--------+---------+----------+-----------+-------------+----------+-----------+-------------+------------
2842766 | 2842766 | 15188975 | 5.34 | 0.00-156.50 | 87728.93 | 0.03 | 0.00-19.69 | UPDATE pgb
2842767 | 2842767 | 3116249 | 1.10 | 0.00-68.69 | 83591.63 | 0.03 | 0.00-22.82 | UPDATE pgb
109428 | 109428 | 486901 | 4.45 | 0.42-38.17 | 3607.42 | 0.03 | 0.00-8.80 | select cou
```

```
select
sum( shared_blks_hit ) as shared_hit ,
sum( shared_blks_read ) as sh_r ,
sum( shared_blks_dirtied ) as sh_d ,
sum( shared_blks_written ) as sh_w ,
round(sum( blk_read_time )::numeric,2) as r_t ,
round(sum( blk_write_time )::numeric,2) as w_t ,
sum( local_blks_hit ) as l_hit ,
sum( local_blks_read ) as l_r ,
sum( local_blks_dirtied ) as l_d ,
sum( local_blks_written ) as l_w ,
sum( temp_blks_read ) as t_r ,
sum( temp_blks_written ) as t_w ,
(select usename from pg_user where usesysid = userid ) as user,
left(query,15) query
from pg_stat_statements
where calls<>0 and (select datname from pg_database where oid = dbid )= current_database
()
group by userid , dbid , query
order by sum( total_exec_time ) desc
limit 10;
```

```
shared_hit|sh_r | sh_d|sh_w|r_t | w_t | l_hit|l_r|l_d|l_w|t_r|t_w | user | query
-- - -------+----+-----+----+----+-----+-----+---+---+---+---+---+--------+ ------------ ---
1123724917| 65 |2930 |2750|0.84|99.20| 0 | 0| 0 | 0 | 0 | 0 | postgres|UPDATE pgbench_
1431243053| 176|4270 |1314|3.22|47.02| 0 | 0| 0 | 0 | 0 | 0 | postgres|UPDATE pgbench_
 15558221| 0 | 0 | 0 | 0.00| 0.00| 0 | 0| 0 | 0 | 0 | 0 | postgres|select count(c)
```

Groups `group by userid , dbid , query` are used when commands with the same text are duplicated (different parse trees or artifacts of concurrent access to the shared extension buffer) and you want to combine the commands to simplify analysis.

Metrics `pg_stat_statements`

- you can optimize the performance of individual commands using the **`explain command (analyze, verbose, settings , buffers , wal )`**
- To monitor the execution of commands in general, use `pg_stat_statements , pg_stat_activity` wait events , cluster diagnostic log
- `pg_stat_activity` restrictions :
  - › does not provide information about the commands being executed
  - › unsuccessfully completed commands are not tracked, for example, those that stopped executing due to `statement_timeout`
- Three metrics can be calculated for each cumulative statistic:
  - › calculate the difference between the values of cumulative statistics and divide by the time interval: Δ M/ Δ t , that is, calculate the derivative with respect to time.
  - › divide not by the time interval, but by the difference in the calls column (how many times the command was executed): Δ M/ Δ c .
  - › The share (percentage) that the indicator occupies relative to all calls M/ sum (M)=%M

Metrics `pg_stat_statements`

You can optimize the performance of individual commands (micro-optimization) using the `explain (analyze, verbose, settings , buffers , wal )` .

The execution time measured by the `EXPLAIN ANALYZE command` may differ significantly from the execution time of the same query in normal mode . The results `of EXPLAIN` should not be generalized to situations significantly different from those in which you are testing. Cost estimates are nonlinear , and the planner may choose different plans depending on the size of the table.

command `( generic_plan )` allows you to use `$1` characters in the command that are present in the `pg_stat_statements.query column` .

To monitor the execution of commands in general , `pg_stat_statements , pg_stat_activity` wait events , and the cluster diagnostic log are used.

Limitations of `pg_stat_activity` : does not provide information about running commands. Also, unsuccessfully completed commands are not tracked, for example, those that stopped executing due to `statement_timeout` . Before the timeout expires, commands actively use host resources. Such commands are tracked in the diagnostic log. If a command contains comments (they can be used by `the pg_hint_plan extension`) , they are ignored during classification, and only one of the comments is saved in query .

Some of the indicators (statistics) in the presentation are cumulative, that is, the value increases over time or with the number of executed commands. For example, `total_exec_time` cumulative, and `min_exec_time` non-cumulative statistics. Absolute values of such indicators do not make sense, they need to be compared with some other indicator (normalized). What to compare with (how to normalize) ?

You need to make two samples from the representation at some interval. For example, after 10 minutes or an hour. The metric is called the derivative of the statistic. Obtaining the derivative is called " differential analysis " .

Three metrics can be calculated for each cumulative statistic:

1) Calculate the difference between the values of cumulative statistics and divide by the time interval: Δ M/ Δ t , that is, calculate the derivative with respect to time.

2) Divide not by the time interval, but by the difference in the calls column (how many times the command was executed): Δ M/ Δ c .

3) The percentage (share) that the indicator occupies relative to all calls M/sum(M)=%M.

Examples of presentation metrics `pg_stat_statements`

- `Δ calls/ Δ t` is the "QPS" metric - the number of calls per second
- `Δ( total_plan_time+total_exec_time )/ Δ calls` - moving average query execution time
- `Δ rows/ Δ t` - number of lines transferred to clients
- `Δ( shared_blks_hit+shared_blks_read ) *8192 / Δ t` - the volume of processed data loading the processor-memory bus
- `Δ( shared_blks_hit+shared_blks_read )/ Δ calls` - large values indicate that the query has potential for optimization
- `Δ wal_bytes / Δ t` - commands generating a lot of WAL
- `Δ wal_bytes / Δ calls` - the volume of WAL generated by the command
- `%time` - the share of the request among all :

```
select calls, round( total_exec_time ::numeric, 2) total_time , round( mean_exec_time ::numeric, 2)
mean_time , round((100 * total_exec_time / sum( total_exec_time ) OVER () )::numeric, 2) "%time" ,
left(query,40) query from pg_stat_statements order by total_exec_time desc limit 10;
 calls | total_time | mean_time | %time | query
---------+------------+-----------+-------+------------------------------------------
 2842766 | 15188975.46 | 5.34 | 75.80 | UPDATE pgbench_branches SET bbalance = b
 2842767 | 3116248.96 | 1.10 | 15.55 | UPDATE pgbench_tellers SET tbalance = tb
```


Examples of presentation metrics `pg_stat_statements`

Each metric: Δ M/ Δ t , Δ M/ Δ c , %M has its own value. If you periodically make selections from the view, you can build a graph of the change in each metric.

Metrics can be calculated not for individual queries (usually with the largest `total_exec_time`), but for `query` groups . For example, queries from one user (the role under which the application is running).

Examples of metrics:

1) `Δ calls/ Δ( t)` - is the "QPS" metric - the number of calls per second . Δ(t) is the time interval in seconds between two requests to the view.

2) `Δ( total_plan_time + total_exec_time )/ Δ t` - the share of time spent on servicing requests

`Δ( total_plan_time + total_exec_time )/ Δ calls` - moving average query execution time. Analogous to `mean_exec_time` , which is an average over the entire period from the statistics reset. The appearance of a significant (by an order of magnitude) deviation of the metric from `mean_exec_time` indicates the emergence of a bottleneck during this period of time.

3) `Δ rows/ Δ t` - the number of rows returned to clients. Returning too many rows may require significant resources on the client side. A large value may indicate that the client is the bottleneck. Spikes in the metric graph may indicate that the client is sending requests synchronously and it is worth inserting delays to spread the requests out over time (" frontend floods ").

4) `Δ( shared_blks_hit+shared_blks_read ) *8192 / Δ t` - the volume of processed data that loads the processor-memory bus. The memory bus bandwidth is limited to the range of 2-20 Gb / s . For large metric values, it is worth optimizing the queries by which the metric was calculated or transferring them to replicas.

`Δ( shared_blks_hit+shared_blks_read )/ Δ calls` - large values indicate that the request has potential for optimization

5) `Δ wal_bytes / Δ t` - commands (query) that generate a lot of log records. A large volume of logs creates a load on physical and logical replication and can increase the replication lag .

6) `Δ wal_bytes Δ calls` - the volume of log records generated by the command.

7) `round((100* total_exec_time /sum( total_exec_time ) over ( ) ::numeric,2)` - the share of the request among all tracked (up to 5000 by default) requests.

Extension pg_stat_kcache

- complements and depends on pg_stat_statements
- collects linux statistics , by executing the getrusage system call after each command is executed
- Unlike operating system utilities, the extension collects statistics down to the command level
- allows you to distinguish whether a block was read from disk or from the page cache

```
 select * from (select *,lead(off) over(order by off)-off as diff from pg_shmem_allocations ) as a where name
like 'pg_%';
 name | off | size | allocated_size | diff
------------------------+----------+-------+----------------+---------
  pg_stat_statements | 148162816 | 64 | 128 | 128
  pg_stat_statements hash | 148162944 | 2896 | 2944 | 2188544
  pg_stat_kcache | 150351488 | 992 | 1024 | 1024
  pg_stat_kcache hash | 150352512 | 2896 | 2944 | 1373056
 select name, setting, context, min_val , max_val from pg_settings where name like '% kcache %';
 name | setting | context | min_val | max_val
-------------------------------+---------+-----------+---------+------------
  pg_stat_kcache.linux_hz | 333333 | user | -1 | 2147483647
  pg_stat_kcache.track | top | superuser | |
  pg_stat_kcache.track_planning | off | superuser | |
```


Extension pg_stat_kcache

The extension complements pg_stat_statements and depends on it . It is not included in the standard delivery. The extension works stably and has insignificant overhead. Statistics shared_blks_read does not distinguish whether the pages (size 4Kb) that make up the block (size 8Kb) were in the page cache linux or read from disk. The extension allows you to distinguish this, it collects Linux statistics , by executing the getrusage system call after each command is executed. Statistics collected by the extension can be useful for determining the effectiveness of caching and possible bottlenecks . Data collected by the system call is written to shared memory.

Call getrusage also used by the configuration parameter log_executor_stats =on (disabled by default). This configuration parameter saves the collected operating system statistics to the cluster diagnostic log, which is less convenient for viewing and the need to monitor the log size.

Unlike operating system utilities, the extension collects statistics with detail down to the command. The number of commands for which statistics are collected and the size of shared memory structures are determined by the pg_stat_statements.max parameter. (default 5000) since this extension depends on the pg_stat_statements extension .

The extension uses two shared memory buffers:

```
 select * from (select *,lead(off) over(order by off)-off as diff from
pg_shmem_allocations ) as a where name like 'pg_%';
 name | off | size | allocated_size | diff
------------------------+----------+-------+----------------+---------
  pg_stat_statements  | 148162816 | 64 | 128 | 128
  pg_stat_statements hash | 148162944 | 2896 | 2944 | 2188544
  pg_stat_kcache | 150351488 | 992 | 1024 | 1024
  pg_stat_kcache hash | 150352512 | 2896 | 2944 | 1373056
```

The extension has the following parameters:

```
 \ dconfig * kcache *
```

pg_stat_kcache.linux_hz (default -1) is automatically set to the value of the linux parameter CONFIG_HZ and is used to compensate for sampling errors . No need to change.

pg_stat_kcache.track = top parameter - analog of pg_stat_statements.track

pg_stat_kcache.track_planning = off analogue of pg_stat_statements.track_planning

Statistics collected pg_stat_kcache

Statistics in pg_stat_kcache views and pg_stat_kcache_detail :

- reads_blks reads, in 8K blocks
- writes_blks writes, in 8K blocks
- user_time user CPU time used
- system_time system CPU time used
- minflts page reclaims (soft page faults)
- majflts page faults (hard page faults)
- nswaps swaps
- msgsnds IPC messages sent
- msgrcvs IPC messages received
- nsignals signals received
- nvcsws voluntary context switches
- nivcsws involuntary context switches

```
alter system set shared_preload_libraries = pg_stat_statements , pg_wait_sampling , pg_stat_kcache ;
create extension pg_stat_kcache ;
\ dx + pg_stat_kcache
function pg_stat_kcache ()
function pg_stat_kcache_reset ()
view pg_stat_kcache
view pg_stat_kcache_detail
```


Statistics collected pg_stat_kcache

Commands to install the extension:

```
apt install clang-13
wget https://github.com/powa-team/pg_stat_kcache/archive/REL2_3_0.tar.gz
tar xzf ./REL2_3_0.tar.gz
cd pg_stat_kcache-REL2_3_0
make
make install
alter system set shared_preload_libraries = pg_stat_statements , pg_wait_sampling , pg_stat_kcache ;
sudo systemctl restart tantor-se-server-16.service
create extension pg_stat_kcache ;
```

The extension consists of two views and two functions:

```
\ dx + pg_stat_kcache
function pg_stat_kcache ()
function pg_stat_kcache_reset ()
view pg_stat_kcache
view pg_stat_kcache_detail
```

pg_stat_kcache_detail view has columns: query, top, rolname and gives data with command accuracy . Statistics are given from 14 columns for planning and 14 columns for command execution.

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
```

View statistics pg_stat_kcache

- database statistics in the pg_stat_kcache view :

```
 select datname database, pg_size_pretty ( exec_minflts *4096) reclaims, 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 | reclaim | faults | reads | writes | sys | usr | vsw | isw
----------+----------+---------+--------+--------+-----+-----+------+------
 postgres | 183 MB | 0 bytes | 226 MB | 13 GB | 39 | 115 | 2717 | 2031
```

- To connect to pg_stat_statements it is convenient to use the **pg_stat_kcache ()** function :

```
 select d.datname database, round( s.total_exec_time ::numeric, 0) time, s.calls , pg_size_pretty ( exec_minflts
*4096) reclaims, 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 2 desc ;
 database|time |calls | reclaims| faults | reads | writes | user | sys | vsw | isw | query
---------+-------+------+---------+---------+-------+---------+------+------+-----+-----+--------------------
 postgres| 25407 | 1 | 3752 kB | 0 bytes | 33 MB | 1277 MB | 11.42 | 4.00 | 306 | 362 | insert into test.f
 postgres| 24817 | 1 | 3752 kB | 0 bytes | 15 MB | 1297 MB | 11.36 | 3.92 | 294 | 167 | insert into test.f
 postgres| 24187 | 1 | 3748 kB | 0 bytes | 35 MB | 1321 MB | 11.70 | 3.71 | 277 | 598 | insert into test.f
```


View statistics pg_stat_kcache

If exec_majflts is insignificant compared to exec_minflts , it means that There is enough RAM.

exec_user_time proportion 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.

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.

To connect to pg_stat_statements , it is more convenient to use the **pg_stat_kcache ()** function rather than views :

```
 select d.datname database, round( s.total_exec_time ::numeric, 0) time, s.calls
, pg_size_pretty ( exec_minflts *4096) reclaims, 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 2
desc ;
 database|time |calls| reclaims| faults| reads | writes | user | sys | vsw | isw | query
---------+-------+-----+---------+--------+-------+---------+------+------+-----+-----+--------------------
 postgres| 25407| 1| 3752 kB | 0 bytes| 33 MB | 1277 MB | 11.42 | 4.00 | 306 | 362 | insert into test.f
 postgres| 24817| 1| 3752 kB | 0 bytes| 15 MB | 1297 MB | 11.36 | 3.92 | 294 | 167 | insert into test.f
 postgres| 24187| 1| 3748 kB | 0 bytes| 35 MB | 1321 MB | 11.70 | 3.71 | 277 | 598 | insert into test.f
```

Difference in column values s.shared_blks_hit , s.shared_blks_read , k.exec_reads /8192 exec_reads_blk , k.exec_minflts cache,k.exec_majflts disk if you select these columns, it may be because of different statistics reset times. Therefore, when using columns from different extensions in calculations, you need to make sure that the statistics were reset at the same time. Statistics from different extensions are reset by different functions.

Practice

- Part 1. Installing the pg_stat_kcache extension
- Part 2. Using the pg_stat_kcache extension
- Part 3. Performance when using direct i / o

Practice

Consider what the statistics of `the pg_stat_kcache extension show` and what statistics make sense.

You will also compare the performance of the instance in normal I/O mode and in direct mode.

16

16

Extension `pg_wait_sampling`

Extension pg_wait_sampling

- included in all Tantor DBMS builds
- returns statistics on wait events of all processes in the instance
- for installation you need to download the library and install the extension:

```
alter system set shared_preload_libraries = pg_stat_statements , pg_stat_kcache , pg_wait_sampling ;
create extension if not exists pg_wait_sampling ;
```

- pg_wait_sampling library must be specified after pg_stat_statements to prevent the extension from overwriting identifiers queries (queryid)
- The extension uses the background process pg_wait_sampling collector
- the process polls the state of all processes in the instance
- The extension includes 4 functions and 3 views:

```
\ dx + pg_wait_sampling
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
```


Extension pg_wait_sampling

Tantor DBMS assemblies . It provides statistics on wait events for all instance processes . To install you need to download the library and install the extension:

```
alter system set shared_preload_libraries = pg_stat_statements , pg_stat_kcache , pg_wait_sampling , pg_qualstats , pg_store_plans ;
create extension if not exists pg_wait_sampling ;
```

pg_wait_sampling library must be specified after pg_stat_statements to prevent pg_wait_sampling from overwriting identifiers queries (queryid) that are used by pg_wait_sampling .

The extension includes 4 functions and 3 views:

```
\ dx + pg_wait_sampling
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
```

Current wait events are displayed in the pg_stat_activity view . Many wait events are short-lived and are unlikely to be " caught " . The extension uses a background process pg_wait_sampling collector , which, with a frequency specified by the parameter pg_wait_sampling.history_period or pg_wait_sampling.profile_period (default 10 milliseconds) polls the state of all processes in the instance, stores pg_wait_sampling.history_size (default 5000, maximum value determined by the int4 type) of events in the history, and groups them into a " profile " of events accessible through the pg_wait_sampling_profile view .

The history is used in a circular fashion: old values are overwritten in a circular fashion. Applications can save the collected history by requesting the history from the view:

```
select count(*) from pg_wait_sampling_history ;
count
-----
5000
```

History of waiting events

- The extension uses " sampling " with a frequency of 1 millisecond (default 10 milliseconds)
- history is available through the view:

```
select count(*) from pg_wait_sampling_history ;
count
-----
5000
```

- by default, the history of the last 5000 wait events is saved
- The extension also uses shared memory to store its three structures:
  - queue (MessageQueue) fixed size 16Kb
  - PID list
  - memory for command identifiers (`queryid`) executed by processes

```
 select * from (select *, lead(off) over(order by off)-off as diff from pg_shmem_allocations ) as a
where name like '%wait%';
 name | off | size | allocated_size | diff
------------------+----------+-------+----------------+-------
 pg_wait_sampling | 148145920 | 17536 | 17536 | 17536
```


History of waiting events

And the history of waiting events can be viewed through the view:

\ sv pg_wait_sampling_history

CREATE OR REPLACE VIEW public.pg_wait_sampling_history AS SELECT pid , ts , event_type , event, queryid FROM pg_wait_sampling_get_history () pg_wait_sampling_get_history (pid , ts , event_type , event , queryid)

`pg_wait_sampling_get_history ()` function produces the same data and has no input parameters.

Obtaining information about what a process is currently executing by polling its state at some frequency is used in Oracle Database ASH (Active Session History) , which is part of AWR (Automatic Workload Repository) .

On an instance with many active sessions, the 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. For example, when running the standard test: **`pgbench -T 100`** among 5000 events in the history, you can sometimes see one line:

```
select * from pg_wait_sampling_history where queryid <>0;
  pid | ts | event_type | event | queryid
-------+----------------------------+------------+---------------------------+---------------------
53517 | 20 3 5-11-1 1 11:18:19.676412+03 | IPC | MessageQueueReceive | 6530354471556151986
```

The extension also uses shared memory to store its three structures:

```
select * from (select *, lead(off) over(order by off)-off as diff from
pg_shmem_allocations ) as a where name like '%wait%';
 name | off | size | allocated_size | diff
------------------+----------+-------+----------------+-------
 pg_wait_sampling | 148145920 | 17536 | 17536 | 17536
```

Most of it is occupied by a queue (MessageQueue) of a fixed size of 16Kb, memory for the list of PIDs , memory for command identifiers (`queryid`) executed by processes. The size of the structure for storing the list of PIDs of processes is determined by the maximum number of instance processes. The number is determined by the configuration parameters and is approximately equal to: `max_connections` , `autovacuum_max_workers+1` (launcher) , `max_worker_processes` , `max_wal_senders +5` (main background processes). Memory for queryid equal to the maximum number of PIDs multiplied by 8 bytes (type size `bigint` used by `queryid`).

History of waiting events

- the history stores the events of all processes' waits
- if server processes do not encounter locks, then 99.98% of wait events will be filled with waits for background processes not related to requests

```
select * from pg_wait_sampling_history where queryid <>0 ;
 pid | ts | event_type | event | queryid
-----------+-------------------------------+----------+--------------------+---------------------
53718 | 20 3 5-11-1 1 11:41:55.629229+03 | IO | BufFileWrite | 6530354471556151986
53718 | 20 3 5-11-1 1 11:41:59.513407+03 | IO | BufFileRead | 6530354471556151986
```

Monitoring programs can request history content and store it in their storage.
Determining the maximum frequency of history polling:

```
 select max( ts ), min( ts ), max( ts )-min( ts ) duration from pg_wait_sampling_history ;
max |              min |    duration
-----------------------------+-----------------------------+-----------------
20 35 -11-1 1 11:58:19.691753+03 | 20 35 -11-1 1 11:55:41.153914+03 | 00:02:38 .537839
```

pg_wait_sampling.history_period and pg_wait_sampling.history_size

History of waiting events (continued)

Shared memory, used by the extension, probably because it was supposed to have multiple handler processes, but they were not needed.

As the number of records increases :

```
alter system set pg_wait_sampling.history_size = 100000;
```

history will be displaced less often and will be retained several times more rows with **queryid <>0** :

```
select * from m pg_wait_sampling_history where queryid <>0;
 pid | ts  | event_type | event | queryid
-----------+-------------------------------+----------+--------------------+---------------------
53718 | 20 3 5-11-1 1 11:41:55.629229+03 | IO | BufFileWrite | 6530354471556151986
...
53718 | 20 3 5-11-1 1 11:41:59.513407+03 | IO | BufFileRead | 6530354471556151986
53718 | 20 3 5-11-1 1 11:42:54.618133+03 | IPC | MessageQueueReceive | 6530354471556151986
(14 rows)
```

Monitoring programs can request the contents of the history and save it in their storage. You can determine the maximum frequency for polling the wait history by requesting:

```
 select max( ts ), min( ts ), max( ts )-min( ts ) duration from pg_wait_sampling_history ;
max |              min |    duration
-----------------------------+-----------------------------+-----------------
20 35 -11-1 1 11:58:19.691753+03 | 20 35 -11-1 1 11:55:41.153914+03 | 00:02:38 .537839
```

In the example 2 minutes 38 seconds . If you request less often, some rows will be lost.

The query can be used to determine the value of the `pg_wait_sampling.history_period` and `pg_wait_sampling.history_size parameters` .

With default values, rows are retained in history for a few seconds.

`pg_wait_sampling.history_period` parameter sets the frequency with which the process state is polled. The default is 10 milliseconds. The minimum value is 1 millisecond. If you reduce the frequency by 10 times (to 1 ms), then the `pg_wait_sampling_history history` will be overwritten 10 times more often and it is worth increasing the `pg_wait_sampling.history_size value by 10 times` so that the event retention time remains the same.

Extension Options `pg_wait_sampling`

- `profile_period` and `history_period` interval in milliseconds for sampling from the history of events and for adding to the profile
  - default 10 milliseconds
  - the probability of catching an event is determined only by these parameters
  - **data for history and profile are selected and saved independently of each other**
- `pg_wait_sampling.history_size` (default 5000, maximum value determined by `int4 type`) of events in history
- `profile_pid` and `profile_queries` break down events in a profile by process and query

```
select name, setting, context, min_val , max_val , boot_val , pending_restart from pg_settings where name like '% pg_wait_sampling %';
                name                 | setting | context | min_val | max_val | boot_val | pending_restart
-------------------------------------+---------+---------+---------+---------+----------+----------------
pg_wait_sampling.history_period      | 100000  | superuser | 1     | 2147483647 | 10   | f
pg_wait_sampling.history_size        | 100     | 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     | on      | superuser |       |            | on   | f
```


Extension Options `pg_wait_sampling`

`pg_wait_sampling.profile_period` parameter sets the interval in milliseconds for sampling events from history for grouping and adding to the profile. The default is 10 milliseconds. **Data for history and profile are selected and stored independently of each other .** Wait events can be caught in the profile, but not in history, and vice versa. The probability of catching an event is determined only by two parameters: `pg_wait_sampling.profile_period` and `pg_wait_sampling . history _period` . The smaller 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` .

`pg_wait_sampling.history_size` (default 5000, maximum value is determined by the `int4 type`) of events stored in the history. The history can be viewed through the `pg_wait_sampling_history view` . The parameter value determines the number of rows in the view. The history is rarely used for queries by administrators, since it does not store the number of wait events, unlike the profile. If the history is not used by monitoring applications, then it is worth increasing the collection interval `profile_period` .

`pg_wait_sampling.profile_pid` and `pg_wait_sampling.profile_queries` rank events in the profile by processes and queries, which increases the local memory of the collector process. `profile_pid =on` has a greater impact on memory, since the profile stores data on completed processes.

Changing the values of all parameters does not require restarting the instance, only rereading the configuration select `pg_reload_conf ();`

The number of lines in the history can be changed either upwards or downwards. Calling **`the pg_wait_sampling_reset_profile () function`** does not clear the history. The history is only overwritten . Calling the function clears the data accumulated in the wait event profile.

`pg_wait_sampling.sample_cpu` by default `on` . Processes that are not waiting for anything will also be polled, the values in the `event_type columns` and `event` will be empty.

The extension appeared in 2016 and is not a standard PostgreSQL extension. The idea of the extension is interesting, but there are no examples where the extension would help solve practical problems. Unlike Oracle Database, frequently executed queries in PostgreSQL do not require sampling and are reflected in `pg_stat_statements` . Implementation of ideas used in DBMS of other families, often meaningless when applied to PostgreSQL .
https://akorotkov.github.io/blog/2016/03/25/wait_monitoring_9_6/

Profile `pg_wait_sampling`

- To view the number of events, use the view:

```
select * from pg_wait_sampling_profile where queryid <>0 order by count desc ;
  pid | event_type | event | queryid | count
-------+-----------+---------------+---------------------+-------
55206 | IO | DataFileRead | -650897274631253140 | 1290
55206 | IO | DataFileExtend | -877373571429139692 | 6
```

- data is saved for all processes, including server processes of completed sessions that are no longer present in the operating system
- if `pg_wait_sampling.profile_pid =true` , then the number of rows will steadily increase
- **the extension does not clear the memory used by the profile**
- **you need to periodically free up memory by calling `the pg_wait_sampling_reset_profile () function`**
  - The function clears the data accumulated in the wait event profile.
  - The function does not clear data in history

Profile `pg_wait_sampling`

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) command types (`queryid`) , if `pg_wait_sampling.profile_queries =true` .
To view the number of events, use the `pg_wait_sampling_profile view` :

```
select * from pg_wait_sampling_profile where queryid <>0 order by count desc ;
  pid | event_type | event | queryid | count
-------+-----------+---------------+---------------------+-------
55206 | IO | DataFileRead | -650897274631253140 | 1290
55206 | IO | DataFileExtend | -877373571429139692 | 6
```

`pg_wait_sampling_profile ()` function produces the same data and has no input parameters.
Data is saved for all processes, including server processes of terminated sessions that are no longer in the operating system. If `pg_wait_sampling.profile_pid =true` , then the number of rows in the view only increases. The rows are in a memory structure, and if the number of rows is large, a lot of memory may be occupied. **The extension does not clear the memory used by this structure. Therefore, you either need to periodically free the memory by calling `the pg_wait_sampling_reset_profile () function`** , or set `pg_wait_sampling.profile_pid =false` . If the cluster serves arbitrary queries, and not a set of predefined ones, then you will also have to set `pg_wait_sampling.profile_queries =false` . It is easier to periodically reset the collected statistics with the function :

```
select pg_wait_sampling_reset_profile () ;
```

An example of how history and profile are independent:

```
\ dconfig *_period
List of configuration parameters
Parameter | Value
---------------------------------+-------
 pg_wait_sampling.history_period | 10000
 pg_wait_sampling.profile_period | 1
(2 rows)
```

Different process polling intervals (" semiling ") are set.

```
postgres=# select * from pg_wait_sampling_history where queryid <>0;
 pid | ts | event_type | event | queryid
-----------+-------------------------------+-----------+------------+-----------------
59651 | 20 35 -11-1 1 1 1 : 11 :37.770929+03 | Client | ClientRead | 193615527688292385
(1 row)
```

There is only one event in history.

```
postgres=# select * from pg_wait_sampling_profile where queryid <>0;
  pid | event_type | event | queryid | count
-------+-----------+----------+----------+-------
59651 | IO | DataFileRead | 193615527688292385 | 27
59651 | IPC | MessageQueueInternal | 3255387186388375512 | 1
59651 | IPC | MessageQueueInternal | 193615527688292385 | 1
59651 | IPC | MessageQueueInternal | -6238149883089971617 | 1
59651 | Client | ClientRead | 193615527688292385 | 3841
59651 | IPC | MessageQueueInternal | 6530354471556151986 | 2
(6 rows)
```

The profile " caught " six wait events that occurred when executing commands (**queryid <>0**).

The history buffer is not completely filled, which means that events were not pushed out of the buffer:

```
postgres=# select count(*) from pg_wait_sampling_history ;
count
-------
126
(1 row)
```

Profile Requests `pg_wait_sampling`

- The profile shows wait events if the wait was caught at least once during the time from the reset:

```
 select * from pg_wait_sampling_profile where pid =55549;
 pid | event_type |        event | queryid | count
------+-----------+--------------------+---------------------+--------
55549 | Client | ClientRead |                   0 | 615444
55549 | IPC | MessageQueueInternal | 3255387186388375512 |      1
55549 | IO | DataFileRead |                  0 |      6
```

- pid column allows you to link rows to the `pg_stat_activity view` :

```
 select p.pid, left( a.backend_type , 14) process_type , a.application_name app, p.event_type , p.event
,
  p.count from pg_wait_sampling_profile p join pg_stat_activity a on p.pid = a.pid
 where event_type <>'Activity';
  pid | process_type | app | event_type | event | count
-----------+----------------+---------+------------+----------------------+--------
 60577 | client backend | psql | Client | ClientRead | 398415
 60683 | client backend | pgbench | IO | WALSync | 246986
 60569 | checkpointer | | Timeout | CheckpointWriteDelay | 171242
 60573 | walwriter | | LWLock | WALWrite | 20021
 60683 | client backend | pgbench | Timeout | SpinDelay | 2
```

- category `event_type = 'Activity'` is used if the background process has nothing to do
- `ClientRead` event ok if it's not in an open transaction

Queries to the pg_wait_sampling profile

The extension has a function **`pg_wait_sampling_get_current (PID)`** . The function can be used to get the current wait event and command ID. Example:

```
 select * from pg_wait_sampling_get_current (55549);
  pid | event_type | event | queryid
-------+-----------+-------+-------------------
55549 | | | 1936155276882923 85
```

Short wait events cannot be caught by this function (the wait columns are empty), so the function is impractical. A profile query will show wait events if the wait was caught at least once during the time from the reset:

```
 select * from pg_wait_sampling_profile where pid =55549;
 pid | event_type |        event | queryid | count
------+-----------+--------------------+---------------------+--------
55549 | Client | ClientRead |                   0 | 615444
55549 | IPC | MessageQueueInternal | 3255387186388375512 |      1
55549 | IO | DataFileRead |                  0 |      6
```

pid column allows you to link rows to the `pg_stat_activity view` :

```
 select p.pid, left( a.backend_type , 14) process_type , a.application_name app,
p.event_type , p.event , p.count from pg_wait_sampling_profile p join
pg_stat_activity a on p.pid = a.pid where event_type <>'Activity';
  pid | process_type | app | event_type | event | count
-----------+----------------+---------+------------+----------------------+--------
 60577 | client backend | psql | Client | ClientRead | 398415
 60683 | client backend | pgbench | IO | WALSync | 246986
 60569 | checkpointer | | Timeout | CheckpointWriteDelay | 171242
 60573 | walwriter | | LWLock | WALWrite | 20021
 60683 | client backend | pgbench | Timeout | SpinDelay | 2
```

event_type = 'Activity' category is used if the background process has nothing to do (category description in `wait_event.h file`), this is the normal state of background processes.

'Timeout' category includes wait events that are usually normal , but not all (WAIT_EVENT_SPIN_DELAY) . You should not exclude events in this category from the sample.

`ClientRead` event `It` 's normal if it's not present in an open transaction. It's waiting for a command from the client and is the most common .

View **`pg_stat_activity`** contains rows only for running processes. View **`pg_wait_sampling_profile`** contains rows for all processes, including those that have terminated. If you use LEFT JOIN , then rows for such processes will be output, but the columns related to **`pg_stat_activity`** will be empty.

Profile Requests `pg_wait_sampling`

- An example of grouping to obtain data on what events the instance processes were most waiting for:

```
 select event_type , event, sum(count) count from pg_wait_sampling_profile where event_type <>'Activity'
group by event_type , event order by count desc ;
  event_type | event | count
------------+-----------------------+--------
  IO | WALSync | 84388
  LWLock | WALWrite | 4549
```

- example of connection with `pg_stat_statements` view :

```
 select calls, round( total_exec_time+total_plan_time ) time, rows, shared_blks_hit hit, shared_blks_read
read, shared_blks_dirtied dirty, p.event_type ev_t , p.event , p.count , left(query,20) query from
pg_stat_statements join pg_wait_sampling_profile p using ( queryid ) order by p.count desc limit 7;
 calls | time | rows | hit | read | dirty | ev_t | event |count | query
------------+-------+--------+--------+-------+-------+--------+----------------+------+----------------
-
 415384|188566 | 415384 | 2572231 | 10815 | 27047 | IO | DataFileRead |15657 | UPDATE pgbench_accou
```

- example of joining with `pg_locks view` :

```
 select p.pid, left( l.relation :: regclass ::text,19) relation, l.locktype , replace( l.mode,'Lock ','')
mode, l.granted g, l.fastpath f, p.event_type type, p.event , p.count from pg_wait_sampling_profile p
join pg_locks l on p.pid = l.pid where event_type <>'Activity' and locktype <>' virtualxid ' order by
p.count desc , p.pid, l.relation desc limit 30;
   pid | relation | locktype | mode | g | f | type | event | count
-------+---------------------+--------------+--------------+---+---+--------+------------+---------
 11739 | pg_locks | relation | AccessShare | t | t | Client | ClientRead | 6622131
 11731 | | transactionid | Exclusive | t | f | IO | WALSync | 1906966
```


Profile Requests `pg_wait_sampling` (continuation)

An example of grouping to obtain data on what events the instance processes were most waiting for:

```
 select event_type , event, sum(count) count from pg_wait_sampling_profile where
event_type <>'Activity' group by event_type , event order by count desc ;
  event_type | event | count
------------+-----------------------+--------
 Client | ClientRead | 224012
 Timeout | CheckpointWriteDelay | 137624
 IO | WALSync | 84388
 IO | DataFileRead | 19406
  LWLock | WALWrite | 4549
```

The extension does not collect the duration of wait events. Collecting the duration would lead to large overhead costs. It is only possible to estimate the duration based on the fact that if the event was caught during polling with an interval of once in `pg_wait_sampling.profile_period` milliseconds, then it is most likely that the duration of the event is not much less than the interval. If the duration is longer, then the probability of catching the event several times increases.

Example of connection with `pg_stat_statements` view :

```
 select calls, round( total_exec_time+total_plan_time ) time, rows, shared_blks_hit hit, shared_blks_read read,
shared_blks_dirtied dirty, p.event_type ev_t , p.event , p.count , left(query,20) query from pg_stat_statements
join pg_wait_sampling_profile p using ( queryid ) order by p.count desc limit 7;
 calls | time | rows | hit | read | dirty | ev_t | event |count | query
------------+-------+--------+--------+-------+-------+--------+----------------+------+----------------
 415384|188566 | 415384 | 2572231 | 10815 | 27047 | IO | DataFileRead |15657 | UPDATE pgbench_accou
 1| 812 | 1 | 49 | 11 | 2 | IO | DataFileRead | 75 | select count(*) from
 415384|  9239 | 415384 | 420880 | 0 | 2667 | IO | DataFileExtend | 4 | INSERT INTO pgbench_
 415384| 21376 | 415384 | 1404998 | 1 | 9 | LWLock | BufferContent | 2 | UPDATE pgbench_branc
```

Background process wait events are not output because they do not execute commands (`queryid =0`) .

Example of joining to the `pg_locks view` :

```
 select p.pid, left( l.relation :: regclass ::text,19) relation, l.locktype , replace( l.mode,'Lock ','') mode,
l.granted g, l.fastpath f, p.event_type type, p.event , p.count from pg_wait_sampling_profile p join pg_locks l
on p.pid = l.pid where event_type <>'Activity' and locktype <>' virtualxid ' order by p.count desc , p.pid,
l.relation desc limit 30;
   pid | relation | locktype | mode | g | f | type | event | count
-------+---------------------+--------------+--------------+---+---+--------+------------+---------
 11739 | pg_wait_sampling_pr | relation | AccessShare | t | t | Client | ClientRead | 6622131
 11739 | pg_locks | relation | AccessShare | t | t | Client | ClientRead | 6622131
 11731 | | transactionid | Exclusive | t | f | IO | WALSync | 1906966
 11731 | pgbench_accounts_pk | relation | RowExclusive | t | t | IO | WALSync | 1906966
```

Reset statistics

- 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:

```
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 ();
```

- To clear the buffer cache, stop the instance: `pg_ctl stop -m fast`
- pg_prewarm cleanup : `rm -f $PGDATA/ autoprewarm.blocks`
- flush dirty pages to disk: `sync`
- clear page cache: `echo 3 > /proc/sys/ vm / drop_caches`
- Launch instance: `sudo systemctl restart tantor-se-server-16`

Reset statistics

Loading extension libraries:

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

Before running 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:

```
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 ();
```

To clear the buffer cache, stop the instance :

```
pg_ctl stop -m fast
```

Removing the `pg_prewarm extension file` :

```
rm -f $PGDATA/ autoprewarm.blocks
```

Flushing dirty pages from the operating system cache to disk:

```
sync
```

slab memory structures of the operating system:

```
root@tantor :~# echo 3 > /proc/sys/ vm / drop_caches
```

or

```
postgres@tantor :~# echo 3 | sudo tee /proc/sys/ vm / drop_caches
```

Launching an instance:

```
sudo systemctl restart tantor-se-server-16
```

Practice

- Using the pg_wait_sampling extension

Practice

In practice, the extension is installed and examples of its use are considered.