

DBA 1-17 Tantor : PostgreSQL 17 Administration

Student Guide



Table of contents

chapter	Tantor Postgres: PostgreSQL Administration 17	page
	Introduction	10
	Preliminary preparation	
	Course materials	
	Course Sections	
	About the course	
	About Tantor Labs	
	Tantor Postgres DBMS	
	Tantor xData	
	Tantor Platform	
	Tantor PipelineDB Extension	
	PostgreSQL Extensions Rework	
	PGBootCamp Conferences	
1 a	Installation	22
	Prerequisites	
	Checking the possibility of installation	
	Installer	
	Local installation	
	Installation process	
	After installation	
	Configurators	
	Creating a cluster using the initdb utility	
1b	Control	31
	pg_ctl instance management utility	
	postgres process	
	Managing an instance via systemctl	
	Working in a docker container	
	Three modes of stopping an instance	
	Stopping an instance	
	Stopping an instance	
	Management utilities (SQL command wrappers)	
	Backup Management Utilities	
	Management utilities (other)	
	Management Utilities (continued)	
1c	psql	43
	Terminal client psql	
	psql : connect to database	
	psql : connection parameters	
	psql commands	
	Formatting psql output	
	Output the result in HTML format	
	Output the result in extended format	
	Prompt for entering commands (prompt) psql	
	Autocommit transactions and running psql commands	
	psql variables	
	Executing batch files in psql	
	Graphical applications: DBeaver	
	Graphical applications: pgAdmin	
	Graphic Applications: Tantor Platform	
	Demonstration	
	Practice	
2a	Architecture	60
	PostgreSQL instance	
	PostgreSQL instance	
	Starting an instance, postgres process	
	Starting the server process	
	Shared memory of instance processes	
	System Catalog Table Cache	
	View pg_stat_slru	
	Local process memory	

	Performance pg_backend_memory_contexts	
	Function pg_log_backend_memory_contexts (PID)	
	Memory structures that support the buffer cache	
	Memory structures that support the buffer cache	
	Search for a free buffer	
	Dirty Buffer Eviction Algorithm	
	Buffer Replacement Strategies	
	Search for a block in the buffer cache	
	Pinning the buffer (pin) and locking content_lock	
	Freeing buffers when deleting files	
	bgwriter background writing process	
	Clearing the buffer cache by the bgwriter process	
	Checkpoint	
	Steps to perform a checkpoint	
	Steps to perform a checkpoint	
	Interaction of instance processes with disk	
	Practice	
2b	Multiversion	86
	Row multiversioning	
	Tables	
	Service columns	
	Data block structure	
	String version header	
	Insert row	
	Update line	
	Delete line	
	Smallest data types: boolean , " char " , char , smallint	
	Variable Length Data Types	
	Integer data types	
	Storing dates, times, and their intervals	
	Data types for real numbers	
	Snapshot	
	Transaction	
	Transaction Properties	
	Transaction Isolation Levels	
	Transaction Isolation Phenomena	
	Example of serialization error	
	Transaction Statuses (CLOG)	
	Committing a transaction	
	Subtransactions	
	Types of locks	
	Object locks	
	Compatibility of locks	
	Object locks	
	Row locks	
	Multitransactions	
	Queue when row is locked	
	Practice	
2c	Routine maintenance	117
	Autovacuum	
	Performance pg_stat_progress_vacuum	
	Parameters VACUUM commands	
	Parameters VACUUM commands	
	Parameter default_statistics_target	
	Bloat tables and indexes	
	Heap Optimization Only Tuple	
	update monitoring	
	Impact of FILLFACTOR on HOT cleanup	
	In-page clearing in tables	
	In-page cleaning in indexes	
	Evolution of indexes: creation, deletion, rebuilding	
	Partial indexes	

	REINDEX Team	
	REINDEX CONCURRENTLY	
	HypoPG expansion	
	Transaction counter	
	Practice	
2d	Executing queries	136
	SQL is a declarative language	
	Syntax parsing	
	Semantic analysis	
	Transformation (rewriting) of a query	
	Execution planning (optimization)	
	Executing a request	
	EXPLAIN command	
	EXPLAIN command parameters	
	Tables	
	Indexes for integrity constraints	
	Methods of accessing data in a query plan	
	String Access Methods	
	Methods for joining sets of rows	
	Cardinality and selectivity	
	Cost of query plan	
	Statistics	
	pg_statistic table	
	Cumulative statistics	
	pg_stat_statements extension	
	pg_stat_statements parameters	
	Practice	
2e	Extensions	158
	PostgreSQL Extensibility	
	Extension and library file directories	
	Installing extensions	
	Extension files	
	Foreign Data Wrapper	
	file_fdw extension	
	dblink extension	
	Practice	
3	Configuring PostgreSQL	167
	Review	
	Configuration parameters	
	View parameters	
	View parameters	
	View parameters	
	postgresql.conf parameter file	
	Parameter file postgresql.auto.conf	
	Applying parameter changes	
	Privileges to change parameters	
	Parameter Classification: Context	
	Context parameters internal	
	Classification of parameters: Levels	
	Classification of parameters: Levels	
	Table-level storage parameters	
	Classification of parameters: Categories	
	Category: "For developers"	
	Category: "Custom Settings"	
	Configuration parameter names and values	
	Configuration parameter transaction_timeout	
	Autonomous transactions	
	View pg_stat_slru	
	Configuration parameter transaction_buffers	
	Parameters multixact_members_buffers And multixact_offsets_buffers	
	Parameter configurations subtransaction_buffers	
	Parameter configurations notify_buffers	

	Setting parameters when creating a cluster	
	Permissions for the PGDATA directory	
	PostgreSQL data block size	
	PostgreSQL Limitations	
	enable_large_allocations parameter	
	Limitations on the length of identifiers	
	Configuration parameters	
	Demonstration	
	Practice	
4a	Logical structure	201
	Database cluster	
	Instance	
	Database	
	List of databases	
	Creating a database	
	Changing database properties	
	ALTER DATABASE command	
	Deleting a database	
	Schemas in the database	
	Creating and modifying schemes	
	Path for searching objects in schemes	
	Special schemes	
	Determining the current search path	
	In which scheme will the object be created?	
	Search path in SECURITY DEFINER routines	
	Masking schema objects	
	System catalog	
	Common Cluster Objects	
	Using the system catalog	
	Accessing the system directory	
	reg-types	
	psql commands	
	Database Inspector in Tanor Platform	
	Demonstration	
	Practice	
4b	Physical structure	227
	PGDATA Cluster File Directory	
	PGDATA Cluster File Directory	
	Temporary objects	
	Tablespaces	
	Tablespaces: Characteristics	
	Tablespaces: Characteristics	
	Tablespace Management Commands	
	Changing tablespace directory	
	Tablespace Parameters	
	Working with log files	
	The main data storage layer	
	Additional layers	
	Location of object files	
	Tablespace and Database Sizes	
	Sizing functions	
	Moving objects	
	Change of scheme and owner	
	Reorganizing and moving tables with pg_repack utility	
	Reducing the size of table files with the pgcompacttable utility	
	TOAST (The Oversized-Attribute Storage Technique)	
	TOAST (The Oversized-Attribute Storage Technique)	
	Fields variable lengths	
	Displacement fields in TOAST	
	Field displacement algorithm in TOAST	
	Toast chunk	

	TOAST Limitations	
	Parameters toast_tuple_target And default_toast_compression	
	Columnar storage : general information	
	Columnar Storage : Features use	
	Columnar storage : parameters	
	Demonstration	
	Practice	
5	Logging	261
	Diagnostic log	
	Levels importance messages	
	Location magazine	
	Broadcast messages syslog	
	Rotation files diagnostic magazine	
	Diagnostic magazine	
	Parameters diagnostics	
	Tracking use temporary files	
	Monitoring the operation of autovacuum and autoanalysis	
	Monitoring checkpoints	
	Description of log_checkpoints entries	
	Description of log_checkpoints entries	
	pg_waldump utility and log_checkpoints entries	
	pg_waldump utility and log_checkpoints entries	
	Connection logging	
	log_connections parameter	
	log_disconnections parameter	
	pgaudit and pgaudittofile extensions	
	pgaudit and pgaudittofile extensions	
	Diagnostics of database connection frequency	
	Diagnostics of blocking situations	
	Practice	
6	Security	284
	Users (roles) in a database cluster	
	Users (roles)	
	Attributes (parameters, properties) of roles	
	INHERIT and GRANT WITH INHERIT attribute	
	Switching a session to another role and changing roles	
	Predefined (service) roles	
	Rights to objects	
	Viewing object permissions in psql	
	DEFAULT PRIVILEGES	
	Row-level protection security (RLS)	
	Connecting to an instance	
	pg_hba.conf file	
	Contents of pg_hba.conf	
	Contents of pg_hba.conf (continued)	
	pg_ident.conf name mapping file	
	Practice	
7 a	Physical backup	301
	Types of backups	
	Incremental backups	
	What is reserved?	
	Recovery procedure	
	Pre-Write Log Files	
	LSN (Log Sequence Number)	
	Log file names and LSNs	
	Startup recovery process	
	Functions for working with logs	
	Cold backup	
	Configuration parameter full_page_writes	
	Backup utility pg_basebackup	
	pg_verifybackup utility	
	Magazine archive	

	No loss (Durability)	
	pg_receivewal utility	
	Replication slot	
	Create a basic backup	
	wal-g backup utility	
	Demonstration	
	Practice	
7b	Logical backup	323
	Logical backup	
	Examples of use	
	Comparison of logical and physical redundancy	
	COPY .. TO command	
	COPY .. FROM command	
	psql \ copy command	
	pg_dump utility	
	Parallel unloading	
	pg_restore utility	
	pg_restore capabilities	
	pg_dumpall utility	
	pg_dumpall capabilities	
	Large size lines	
	enable_large_allocations parameter	
	Demonstration	
8a	Physical replication	339
	Physical replication	
	Master and replica	
	Replicas and archive of the magazine	
	Setting up the wizard	
	Creating a replica	
	Replication slots	
	Configuration parameters on replicas	
	Hot replica	
	Feedback to the master	
	Horizon Monitoring	
	Horizon Monitoring	
	Parameters max_slot_wal_keep_size And transaction_timeout	
	Master settings that should be synchronized with replicas	
	Master-replica role reversal	
	Promoting a replica to master	
	Timeline History Files	
	pg_rewind utility	
	Replica instance processes	
	Delayed replication	
	Recovering damaged data blocks from a replica	
	Demonstration	
	Practice	
	Practice	
8b	Logical replication	363
	Logical replication	
	Using Logical Replication	
	Physical and logical replication	
	Identifying Strings	
	Methods of string identification	
	Steps to create logical replication	
	Creating a publication	
	Create a subscription	
	Create a subscription	
	Load per instance	
	Getting log data from a replica	
	Conflicts	
	Bidirectional replication	
	Demonstration	

	Practice	
9	Tantor Platform Review	379
	Use cases	
	Monitoring tools	
	Tantor Platform	
	User settings	
	Workspaces	
	Instance review	
	Patroni cluster	
	Query Profiler	
	Replication	
	Tablespaces	
	Notifications	
	Monitoring configuration	
	Analytics	
	Background process activities	
	Settings	
	Data schema analysis	
	Routine maintenance	
	Task Scheduler	
	Tantor Platform Course	
10	Tantor Postgres 17 New Features	399
	Tantor Postgres - PostgreSQL branch	
	Improvements in Tantor Postgres	
	Additional configuration options	
	Tantor Postgres SE and SE 1C Extensions	
	Parameters optimizer requests	
	Library pg_stat_advisor	
	Parameters enable_temp_memory_catalog and enable_delayed_temp_file	
	Parameter enable_large_allocations	
	Algorithm compression pglz	
	Parameter libpq_compression	
	Parameter wal_sender_stop_when_crc_failed	
	Parameter backtrace_on_internal_error	
	uuid_v7 extension	
	Extension pg_tde (Transparent Data Encryption)	
	Validator oauth_base_validator	
	Library credcheck	
	Extensions fasttrun and online_analyze	
	Extension mchar	
	Extension full	
	Extension orafce	
	http extension	
	Extension pg_store_plans	
	Extension pg_variables	
	Performance when using pg_variables	
	Benefits of the pg_variables extension	
	pg_stat_kcache extension	
	Statistics collected by pg_stat_kcache	
	pg_wait_sampling extension	
	Waiting Event History	
	pg_background extension	
	pgaudit and pgaudittofile extensions	
	pgaudit and pgaudittofile extensions	
	pgcopydb utility	
	pg_anon utility	
	pg_configurator utility	
	pg_diag_setup.py utility	
	Utility pg_sec_check	
	WAL-G (Write-Ahead Log Guard) utility	
	Other extensions	
	Practice	

Copyright

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

The documents are protected by copyright and intellectual property laws.

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

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

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

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

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

Disclaimer for content, products and services of third parties:

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

Copyright © 2025, Tantor Labs LLC

Author : Oleg Ivanov



Created: **25 June 2025**

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



Introduction

PostgreSQL 17 Administration



Preliminary preparation

- Introduction to SQL
- Skills in working in the Linux operating system
- Experience with any relational database is recommended.

Target audience

- PostgreSQL Database Administrators
- Database Application Developers
- Technical support staff



Preliminary preparation

To successfully complete the course, basic skills in working with Linux operating systems and basic knowledge of the SQL language are sufficient: understanding of the SELECT, UPDATE, INSERT and DELETE commands. Operating system skills include: the ability to launch a terminal, view the contents of directories and files in the terminal, copy and edit text files using the ls, cp, mv, cat, vi commands; understand and change file permissions (ls -al, chmod, chown commands).

The course will cover the main tasks of administering PostgreSQL family databases using the example of the Tantor Postgres DBMS and the specifics of operating the Tantor Postgres DBMS.

To successfully complete the course, it is recommended to listen to the instructor, ask questions if they arise, read the text of practical assignments and complete them independently. When completing practical assignments, it is recommended to type commands on the keyboard, rather than copy them into the terminal from the text of the assignments. Manually entering commands, correcting typos that occur when typing commands, studying errors issued for incorrect commands allows you to better remember the commands and the meaning of their use. The feeling of "understanding" the text of the assignments is not enough; in real work, it is important to remember the main keywords and capabilities of the commands in order to quickly find the full syntax. Copying commands can be used if you are familiar with them.

Course materials

- A textbook in the form of a book in pdf format, which contains the theoretical part of the course
- Practical tasks
- virtual machine for performing practical tasks



Course materials

The course materials include:

- 1) A textbook in the form of a book in pdf format, which contains the theoretical part of the course.
- 2) Practical tasks in the form of a book in pdf format.
- 3) A virtual machine with the Astra Linux 1.8 operating system and Tantor Postgres DBMS version **17.5 installed** . Access to the virtual machine for the duration of the course or an image in ova format may be provided. The virtual machine image can be used with Oracle VirtualBox version 6.1 and higher.

Course Sections

0. Introduction
1. Installation and management of DBMS
2. Architecture (5 parts)
3. Configuration
4. Databases (2 parts)
5. Journaling
6. Security
7. Backup (2 parts)
8. Replication (2 parts)
9. Tantor Platform, Overview of Features
10. New Features of Tantor Postgres 17.5



Course Sections

Installation and management of DBMS
Installation
Managing a Database Cluster Instance
Database Cluster Management Utilities
psql utility
Architecture
General information and memory structures
Multiversion
Routine maintenance work
Executing queries
Extensibility
Configuration
Databases
Logical and Physical implementation
Diagnostic log
Safety
Connection and authentication
Backup
Physical and logical redundancy
Replication
Physical and logical
Tantor Platform, Overview of Features
What's New in Tantor Postgres Version 17.5

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 be shifted 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) practical tasks in pdf format
- 3) virtual machine image in ova format

About Tantor Labs

- since 2016 on the international market
- since 2021 on the Russian market
- development of Tantor Postgres DBMS
- development of the Tantor Platform for monitoring and managing PostgreSQL family DBMS, as well as Patroni clusters
- many years of experience in operating high-load systems
- is part of the Astra Group



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 Postgres DBMS, based on the program code of the freely distributed PostgreSQL DBMS.

In 2021, the company completely reoriented itself to the Russian market, where it concentrated its main activities on the design and development of the Tantor Postgres 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.

At the end of 2022, the company joined the Astra Group.

Tantor Postgres

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



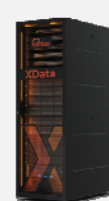
DBMS for high loads, optimized and approved for working with 1C applications

Tantor PipelineDB



An extension that allows continuous data processing

As part of Tantor xData



Maximum version of DBMS, optimized for working with 1C



Training Course "Tantor : PostgreSQL 17 Administration" Astra Group, "Tantor Labs" © 2025 www.tantorlabs.ru

16

Tantor Postgres RDBMS

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

Technical support, assistance in building architectural solutions, and migration from DBMSs of other manufacturers are available for all editions. Tantor Labs software is included in the "Unified Register of Russian Programs for Electronic Computers and Databases". When purchasing Tantor Postgres, a license for the Tantor Platform is provided for managing the acquired DBMSs.

Tantor xData

- Hardware and software complex with high performance, fault tolerance, security
- version 2A - AMD EPYC and x86-64 processors
- version 2B - Baikal-S processors
- high performance and scalability
- reducing infrastructure and administration costs
- includes Tantor Postgres DBMS and Tantor Platform



Tantor xData

The Tantor XData appliance delivers high-scale, mission-critical workloads with high performance and availability. Consolidating multiple Tantor Postgres SE workloads on an XData database engine in enterprise data centers helps organizations improve operational efficiency, reduce administration, and lower costs.

The hardware and software complex (HSC) Tantor XData is designed for migration from foreign manufacturers' systems and provides similar load capacity. It is a replacement for high-load 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 consolidation of 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.

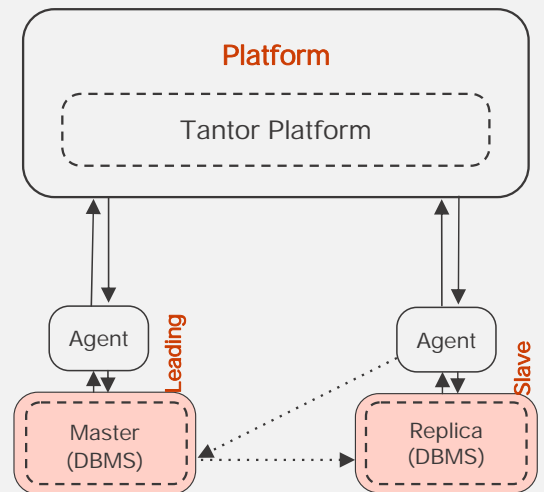
Since 2025, **the second version of** the PAK has been produced:

xData 2 A - on AMD EPYC and x86-64 processors based on Yadro servers.

xData 2 B - on Baikal-S processors manufactured by Baikal Electronics (Astra Group).

Tantor Platform

- software for managing a large number of DBMS and Patroni clusters
- manages Tantor Postgres DBMS and PostgreSQL forks
- performance tuning recommendations
- integration with mail systems, directory services, messengers



Tantor Platform

The Tantor platform is software for managing the Tantor Postgres DBMS, PostgreSQL forks, and Patroni clusters. It allows you to conveniently manage a large number of DBMS. It belongs 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 Postgres DBMS, but also other DBMSs of the PostgreSQL family
5. Integration with mail systems, directory services, messengers
6. Simple implementation: deployment and introduction of the DBMS under maintenance by the Tantor Platform using Ansible.

Tantor DLH Platform

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

Tantor PipelineDB Extension

- extension for open source Tantor Postgres and PostgreSQL DBMS for continuous execution of SQL queries on data streams with incremental saving 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 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
```



Training Course "Tantor : PostgreSQL 17 Administration" Astra Group, "Tantor Labs" © 2025 www.tantorlabs.ru

19

Tantor PipelineDB Extension

Tantor PipelineDB is an extension for the Tantor Postgres and PostgreSQL DBMS ([free](#) Apache 2.0 license), unlike the [limited](#) license of the timescaleDB extension. Allows continuous processing of streaming data with incremental saving of results in tables. Data is processed in real time using only SQL queries. It has a large set of analytical functions that work with constantly updated data. Allows you to connect streaming data with historical data for comparison in real time. 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.

Querying continuous views **instantly** returns **up-to-date** results, making TantorPipelineDB suitable for 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;
```

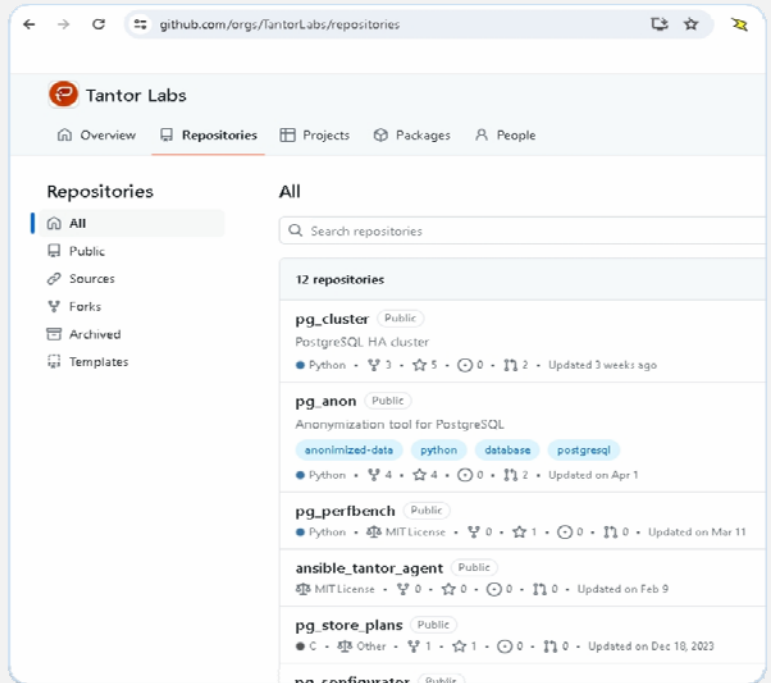
Continuous view [to display daily traffic](#) used by [top ten IP addresses](#) :

```
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>

PostgreSQL Extensions Rework

- 1. pg_cluster
- 2. pg_anon
- 3. pg_perfbench
- 4. ansible_tantor_agent
- 5. pg_configurator
- 6. pg_store_plans
- 7. ldap2pg
- 8. citus
- 9. wal-g
- 10. odyssey
- 11. plant
- 12. pg_orchestrator
- 13. pgtools
- 14. pipelinedb



PostgreSQL Extensions Rework

Tanitor Labs employees develop and create extensions for the PostgreSQL DBMS.

Extension repositories: <https://github.com/orgs/TanitorLabs>

List of extensions:

1. pg_cluster
2. pg_anon
3. pg_perfbench MIT License
4. ansible_tantor_agent MIT License
5. pg_configurator MIT License
6. pg_store_plans
7. ldap2pg PostgreSQL License
8. citus GNU Affero General Public License v3.0
9. wal-g Apache License, Version 2.0 (Izo - GPL 3.0+)
10. odyssey BSD 3-Clause "New" or "Revised" License
11. plantuner
12. pg_orchestrator MIT License
13. pgtools
14. pipelinedb Apache License 2.0

PGBootCamp Conferences

- Tantor Labs takes part in organizing conferences
- The PGBootCamp conference was held:
 - › Yekaterinburg April 10, 2025
 - › Kazan September 17, 2024
 - › Minsk April 16, 2024
 - › Moscow October 5, 2023



PGBootCamp Conferences

Tantor Labs is an active participant in organizing PostgreSQL community conferences as part of the global PG BootCamp initiative.

Participation in the conference is free and possible online and offline: <https://pgbootcamp.ru/>

You can become a speaker at a conference.

Conference papers are openly available: <https://github.com/PGBootCamp>

Performances <https://www.youtube.com/@PGBootCampRussia>

The PGBootCamp conference was held:

Yekaterinburg April 10, 2025

Kazan September 17, 2024

Minsk April 16, 2024

Moscow October 5, 2023



1

1a

Installation



Prerequisites

For installation you will need:

- supported operating system
 - › Astra Linux operating system is recommended
 - › 10 operating systems supported
- Minimum hardware requirements:
 - › 4 CPU cores, 4GB RAM, 40GB SSD



Prerequisites

Tantor Postgres is supplied in compiled form as packages for the operating system package manager. Before installation, you need to check the list of operating systems and their versions for which the Tantor Postgres DBMS is released. The list of supported operating systems includes:

Operating systems with RedHat Packet Manager (rpm)

Redos 7.3

AltLinux c9f2 (P8), p10

Centos 7

MSVSpheer

Oracle Linux 8

Rocky 8, 9

Operating systems with Debian package manager (deb)

Astra Linux Common Edition 2.12

Astra Linux Special Edition 4.7, 1.7, 1.8

Ubuntu 18, 20, 22

Debian 10, 11, 12.

Installation on other operating systems is not supported.

Equipment:

Number of CPU cores: at least 4;

RAM: at least 4GB;

Free disk space: at least 40GB (plus space for user data that you plan to store). Using SSD is recommended.

https://docs.tantorlabs.ru/tdb/en/17_5/se/install-binaries.html

Checking the possibility of installation

- Distributions are distributed as a file for Red Hat (rpm) and Debian (deb) package managers.
- The distributions indicate the packages whose functionality can be accessed by utilities and processes of the cluster instance.
- The documentation does not list dependencies, as it differs for different versions and builds.
- Before installation you can get a list of packages that require installation



Checking the possibility of installation

Programs use shared libraries that provide useful functionality and were used when building them. If the libraries are not installed in the operating system, errors may occur during operation, the cause of which will be difficult to determine. Distributions list the libraries whose functionality utilities and processes can access. Such packages are called required and are related to dependencies. Dependencies may include not only packages, but also the needs of command files called during installation and other tools.

Since the list of dependencies may differ between different versions and builds of PostgreSQL, the documentation does not list the required libraries or packages.

In practice, getting a list of packages that need to be installed is a pressing task.

To get a complete list of dependencies for a specific distribution, you can use the commands:

For Debian package manager: `dpkg -I tantor*.deb`

For RedHat package manager: `rpm -qp --requires tantor-se-server-*.x86_64.rpm`

The utility's response consists of a list of packages and, possibly, versions of packages and libraries. For example:

```
shadow-utils
```

```
grep
```

```
...
```

```
rpmlib(PayloadIsXz) <= 5.2-1
```

`<=` and `=>` symbols indicate that specific library versions are required. The last line of the example specifies the package manager version restrictions, which set the rpm package manager compatibility check to protect against installation on an incompatible operating system version.

This check can be useful for creating a task for installing an operating system. The command can be run on any Linux operating system where the rpm package manager is installed.

To check that dependencies are met before installation, you can use the command:

```
rpm -i --test tantor*.rpm
```

Example of error when checking dependencies of Rocky 9 distribution on Oracle Linux 7.

```
warning: tantor-be-server-17-17.5.1.el9.x86_64.rpm: Header V4 RSA/SHA512 Signature
```

```
error: Failed dependencies:
```

```
...
```

```
python3-libs is needed by tantor-be-server-17-17.5.1-0.x86_64
```

```
rpmlib (PayloadIsZstd) <= 5.4.18-1 is needed by tantor-be-server-17-17.5.1-0.x86_64
```

Errors related to **rpmlib** indicate an unsuitable distribution.

Installer

- checks for possible library conflicts and suggests a command to resolve them
- adds repository address to apt and yum list
- downloads the appropriate distribution and performs installation, cluster creation, service creation
- is a text script:
 - You can see what actions the installer performs
 - it's easy to figure out what the errors are and fix them



Installer

To simplify the installation, Tantor Postgres can be installed using the installer. The installer is downloaded using the command:

```
wget https:// public.tantorlabs.ru /db_installer.sh
```

Once the download is complete, you can change the file system permissions so that the installer script can run:

```
chmod +x db_installer.sh
```

The distribution can be downloaded from your personal account <https://lk.astra.ru/iso-images> and specify the path to the downloaded file to the installer using the **--from -file parameter** :

```
./db_installer.sh --from-file=./tantor-se-server-17_17.5.0_amd64.deb
```

The installer can download the distribution from the Tantor Labs repository. To do this, you need to set the environment variable `NEXUS_URL` :

```
su -
export NEXUS_URL="nexus-public.tantorlabs.ru"
apt update
./db_installer.sh --edition= be
```

To download commercial versions, you need to set environment variables:

```
su -
export NEXUS_URL="nexus.tantorlabs.ru"
export NEXUS_USER="name"
export NEXUS_USER_PASSWORD="password"
apt update
./db_installer.sh --edition= se
```

Possible errors:

1) Conflicts. For example, the client (tantor-se-client-17_17.4.0_amd64.deb) was installed, and the package with tantor-se-server-17 includes the tantor-se-client-17 libraries. In this case, the installer will return an error and a command to fix it by uninstalling the package with which the conflict was detected:

E: Unmet dependencies. Try 'apt --fix-broken install' with no packages (or specify a solution).

After running **apt --fix-broken install** , the utility will ask for confirmation to uninstall the package.

2) The installer creates the file `/etc/apt/sources.list.d/tantorlabs.list` or `/etc/yum.repos.d/tantorlabs.repo` and subsequently it will be possible not to set environment variables. If there is an authentication error or you want not to authenticate, you will need to delete the specified files. Authentication data for downloading the distribution only allows you to download commercial distributions and saving it in the tantorlabs.list file is not considered a security breach.

3) There are files with addresses of non-existent repositories in the `/etc/apt/sources.list.d/` directory . You need to delete such files.

Note: The `apt update` command updates the contents of `/var/lib/apt/lists/` by downloading and expanding archives containing packages contained in the repositories.

Local installation

- Installer parameters `./db_installer.sh --help`
- During installation you can create a cluster
- distributions (rpm and deb packages) have a standard format, they can be unzipped, find out what changes are made to the operating system during the installation process



Local installation

Tantor Postgres Basic Edition (BE) is available for evaluation use. To install Tantor Postgres BE, you only need to set one environment variable:

```
export NEXUS_URL="nexus-public.tantorlabs.ru"
```

Update package lists from repositories:

```
apt update
```

Run the installer, specifying the desired parameters:

```
./db_installer.sh --edition=be --major-version=16 --do-initdb
```

You can specify the main version and whether to create a cluster after installation. You can also create a cluster after installation using the `initdb` utility .

The installer allows you to install any Tantor Postgres DBMS builds from package files. This can be useful if the host does not have Internet access.

Before you begin the installation, make sure you have downloaded the correct binary package that is compatible with your operating system and architecture. The file should have the extension `.deb` for Debian-based systems, `.rpm` for Red Hat-based systems.

To start the installation, go to the directory where the downloaded file is located. Make sure that the installation script `db_installer.sh` is present and has the necessary execution rights. Local installation is performed by the command:

```
./db_installer.sh --do-initdb --edition= se --major-version= 17 --from-file= ./  
tantor- se -server- 17 _5.0_amd64.deb
```

You need to specify the major version with the `--major-version=17` parameter and it must match the version (usually present in the package file name), otherwise the installer may create a directory with an incorrect version number.

You can also install the package without using the installation script, but using the operating system's package manager:

```
rpm -i tantor*.rpm or dpkg -i tantor*.deb
```

In this case, the cluster will not be created and can be created later using the `initdb` utility. In fact, the installer can be useful for local installation because it can perform additional actions. The disadvantage may be that the program code (a wrapper over the package manager) can add errors. For example, it may not provide for all possible features of the operating system configuration.

Installation process

- a user named postgres is created
- Cluster Launcher Service is being created
- directories and files are created
- the required permissions are set on directories, files and their owners



Installation process

During installation:

1) a user is created or modified for the Tantor Postgres DBMS, under which processes will be launched and which will be the owner of directories related to database clusters. Example of a command to create a user and the name postgres:

```
2) useradd -r -g postgres -c "Tantor database server" -d /var/lib/postgresql -s /bin/bash postgres
```

There is no need to change the postgres username to another one for security reasons.

3) The directory `/opt/tantor/db/17` is created, which contains the executable and auxiliary files of Tantor Postgres.

4) The service descriptor file `/usr/lib/systemd/system/tantor-se-server-17.service` is created so that an instance servicing the database cluster can be launched. The database cluster is a directory in the file system of the host (synonyms: computer, node, server) on which the Tantor Postgres software has been installed. DBMS clients do not have direct access to the files. In order for client programs to be able to "work with the DBMS" (send commands in SQL, receive data), a set of processes must be launched on the host that will read and write to the cluster directory and maintain a connection ("socket") with the client program. Such a set of processes and the memory they use in the host operating system are called an "instance" or an instance of the database cluster. The

service status can be checked using the command

```
systemctl status tantor-se-server-17
```

5) the file `/etc/ld.so.conf.d/tantor-se-17.conf` is created. You can view the list of shared libraries loaded into the `/etc/ld.so.cache` cache using the command `ldconfig -p | grep tantor`. You can check that

the LD PRELOAD environment variable does not contain libraries that can overlap the PostgreSQL libraries, since LD PRELOAD prevails.

6) the directory `/var/run/postgresql` and the file `/usr/lib/tmpfiles.d/tantor-db.conf` are created. The file is used by the standard temporary file cleaning service. The directory is the default directory for the Tantor Postgres Unix socket files (configuration parameter (`unix_socket_directories`)). In older versions of PostgreSQL, the directory `/tmp` was used, which, by inertia, may be mentioned in documentation and manuals.

You can check that the directory `/usr/lib/tmpfiles.d` does not contain other files that could remain from previous installations of postgresql, in which the same directory was specified, but with different parameters

```
systemctl status systemd-tmpfiles-*
```

```
systemd-tmpfiles[]: /usr/lib/tmpfiles.d/tantor-db.conf:1: Duplicate line for path  
"/run/postgresql", ignoring
```

7) the directory `/var/lib/postgresql/tantor -se -17/data` is created

, this is the default directory for cluster files. Tantor Postgres utilities learn about the location of the cluster directory either by the `-D` switch (utility parameter), or from the PGDATA environment variable. Therefore, in spoken language this directory is called "PGDATA".

8) `export PATH=$PATH`

```
export PATH=/opt/tantor/db/17/bin:$PATH
```

are added

to the end of the file `/var/lib/postgresql/.bash profile` so that when logged in as the postgres user, you can run the database cluster management utilities without specifying the path to them.

After installation

- Set the PGDATA environment variable in the postgres user profile
- create a cluster if it has not been created yet
- start a cluster instance
- set initial values of cluster parameters using <https://tantorlabs.ru/pgconfigurator>
- install management and monitoring tools (Tantor Platform)



After installation

PostgreSQL has no limit on the number of instances that can be run on a single node. However, production database servers are usually heavily loaded and do not typically run multiple instances of a database cluster on a single node. Multiple instances on a single node may be run temporarily during migration to a new version.

Some PostgreSQL package distributions contain the `pg_controlcluster`, `pg_createcluster` utilities, which are wrappers for the standard `pg_ctl`, `initdb` utilities. The developers of such distributions assume that this simplifies working with several clusters on one node. Tantor Postgres does not use these utilities. Cloud distributions, where the work of a large number of instances is required, can use other assemblies (synonym for forks) of PostgreSQL.

After installation, you can do the following:

1) Set the environment variable in the file

```
/var/lib/postgresql/.bash_profile
```

```
export PGDATA=/var/lib/postgresql/tantor-se-17/data
```

this will simplify the launch of the cluster management utilities that we will look at later `pg_ctl`, `pg_controldata`, `pg_backup` and others - they will not need to specify the launch parameter `-D path_to_PGDATA`.

2) create a cluster if it has not been created yet

3) start cluster `systemctl start tantor-se-server-17`

4) If automatic instance startup was disabled (enabled by default), then enable: `systemctl enable tantor-se-server-17`

5) The Tantor Postgres version can be found using the `tantor_version()` function

6) set initial values of cluster parameters using <https://tantorlabs.ru/pgconfigurator>

7) configure cluster management and monitoring tools Tantor Platform, dBeaver.

8) uninstallation (e.g. of a previous version) is performed by the package manager. For Debian-based systems, `apt-get remove tantor-se-server-17`

https://docs.tantorlabs.ru/tdb/en/17_5/se/binary-download-execute.html

Configurators

- pg_configurator utility
- web version
<http://tantorlabs.ru/pgconfigurator>
- host and planned load characteristics are entered
- gives configuration parameters

The screenshot shows the 'Конфигуратор параметров производительности PostgreSQL' (PostgreSQL Performance Parameters Configurator) web interface. On the left, there are input fields for: 'Доступное количество ядер CPU' (8), 'Доступное количество оперативной памяти RAM, МБ' (1962), 'Тип диска' (SSD), 'Нагрузка базы данных' (ERP/C), 'Платформа' (Linux), 'Версия PostgreSQL' (15), and 'Формат вывода'. On the right, a 'Файл конфигурации' (Configuration File) section displays a list of recommended PostgreSQL parameters and their values, such as 'min_wal_size = 959MB', 'max_wal_size = 2059MB', 'max_replication_slots = 10', etc.



Configurators

The database cluster is created by the initdb utility. The utility creates a postgresql.conf file with default values. These values are designed to support a not very loaded application so that the DBMS can be used on a desktop by an ordinary developer. In Tantor Postgres, the initdb utility does not change the parameter values compared to PostgreSQL's initdb. It is assumed that the parameters for industrial use will be configured separately.

For initial setup, you can use the pg_configurator utility created 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. PGConfig <https://github.com/pgconfig/api>, web version www.pgconfig.org gives recommendations based on 8 parameters
3. PGTune github.com/leopard/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.

Creating a cluster using the initdb utility

- The cluster is created using the `initdb` command line utility.
- `initdb` is run under the operating system user `postgres`
- Before running the utility, you need to create a directory for the cluster files
- select localization settings
- see other utility options



Creating a cluster using the initdb utility

The cluster is created by the command line utility `initdb` . The utility can be called by the installer. `initdb` is run under the `postgres` operating system user.

Before running the utility, you need to create a directory where the files of the created PGDATA cluster will be located, set permissions and ownership rights for this directory and the directories in which it is located for the `postgres` user. When starting the instance, permissions for the PGDATA directory itself are checked :

- 1)permissions should be 0700 (`drwx --- ---`) or 0750 (`drwx rx ---`)
- 2)The owner must be the `postgres` user.

When creating a cluster, you need to select localization settings that cannot be changed after the cluster is created (for databases created when creating a cluster `postgres template0 template1`), but can be selected for databases created after the cluster is created:

- 1)`LC_COLLATE` text sorting rules
 - 2)`LC_CTYPE` character classification (uppercase letters, lowercase letters, digit characters, and other character classes)
- `LOCALE` character encoding scheme - the third part of the value after the dot. This part must be UTF8 or one of the Cyrillic-supporting encodings. Not all combinations are available and can be selected for single-byte encodings. You can select `ru_RU.iso88595` since it is available in operating systems supported by PostgreSQL.

Localization parameters can be set in `initdb` parameters `--locale=en_US.UTF8 --lc-collate=en_US.UTF8 --lc-ctype=en_US.UTF8 --locale-provider={libc|icu} --encoding=UTF8`

If you do not specify parameters, environment variables are used. You can get a list of them with the `locale` command. The list of valid combinations is `locale -a` . Configure `dpkg-reconfigure locales` . `libc` is the standard provider. It makes sense to specify the `--encoding` parameter if the `LOCALE` value does not contain an encoding (after the dot) and there are several valid (compatible) options.

When choosing between UTF8 and `iso88595` (or `cp1251`), you can take into account that in UTF8, Cyrillic characters take up more space - two bytes instead of one. However, applications may need to store, for example, a client's last name in their national language. About the single-byte `koi8` encoding: it should not be used because the binary comparison of characters does not correspond to the linguistic one.

Parameters to pay attention to:

- `-k -g` explicitly enable checksums for data blocks (for WAL records they are enabled) and set less restrictive permissions on created files and directories `0 755` (**zero** means it is an octal number).
- `--auth --auth-host --auth-local`
- `-D` path to PGDATA
- `--wal-segsize` number in megabytes. Default is 16MB. The value must be a power of two. This parameter **can be changed** after cluster creation using `pg_resetwal` utility `--wal-segsize=new_size`

Specifies the size of the write-ahead log file (synonyms WAL file, WAL segment). It is changed either because of a large number of files in one directory, or because the maximum size of the log buffer in shared memory (`wal_buffers`) is limited by the size of the WAL file. The effect of the WAL buffer size on performance is nonlinear.



1

1b

Control



pg_ctl instance management utility

- Main actions: stop, reread configuration files, check instance status, and possibly start instance
- the advantage is the ease of use and getting the result of the command execution
- runs under the postgres user



pg_ctl instance management utility

`pg_ctl` is an instance management utility. The advantage of the utility is its simplicity and ease of use via the command line. Integration with system tools: `pg_ctl` can be easily integrated with system tools and automation scripts, making it a useful tool for automating the management of the PostgreSQL database server. `pg_ctl` provides a powerful and flexible way to manage the PostgreSQL database server, making it one of the essential tools for PostgreSQL administration.

When providing or receiving technical support, it allows you to accurately execute short commands and receive the results of their execution, which can be given via instant messengers in the form of text messages. This is one of the advantages of console utilities compared to graphical ones.

The main commands that can be used with `pg_ctl` are:

start - launching an instance

stop -m smart | **fast** | **immediate** - stop

Before stopping the industrial cluster, it is recommended to perform a checkpoint, i.e. issue the checkpoint command. This reduces the time for stopping.

restart - restart, equivalent to stop and start, so the parameters that are set when stopping can be used.

reload - reloads configuration files without stopping the instance

status - displays the status of the instance

To start an instance, you need to specify the cluster directory - `PGDATA` . This can be done by setting the environment variable before starting `pg_ctl` or by specifying the path to the directory in the `-D` parameter.

postgres process

- `pg_ctl` starts the postgres process, which spawns the rest of the instance processes
- `-o` parameter of the `pg_ctl` utility can be used to pass command line parameters and cluster configurations to the postgres process
- `postgres --single` allows using single-user and single-process mode. This mode is used to fix the cluster contents in complex cases of corruption.
- To exit single-user mode, use the key combination `ctrl+d`



postgres process

`pg_ctl` starts the postgres process, which forks the other processes in the instance and listens for incoming connections. The postgres process has parameters that `pg_ctl` can pass to it. In older versions of PostgreSQL, the postgres process was called postmaster.

To pass configuration parameters from `pg_ctl` to postgres, use the **`-o parameter`**. For example,

```
pg_ctl start -o "-- config_file=./postgresql.conf -- work_mem=8MB "
```

you can also use the syntax

```
pg_ctl start -o "-c config_file=./postgresql.conf -c work_mem=8MB "
```

See the list of parameters that can be passed to postgres:

```
postgres --help
```

The `--single` option starts the postgres process in single-user, single-process mode:

```
postgres --single
```

PostgreSQL stand-alone backend 17.5

```
backend> vacuum full
```

To exit this mode, use the key combination `<ctrl+d>`.

This is not psql, there are no psql commands in this mode, only commands that the server process (synonym backend) can accept.

Parameter `--single` **cannot be passed via `pg_ctl`**, since there is no interprocess communication.

In this mode, there is no interprocess communication and memory locks. Thanks to this, commands are executed faster. This mode is used in rare cases for commands that correct the contents of the cluster, for example `vacuum full`.

Managing an instance via systemctl

- manages only instances launched by it, an instance launched by `pg_ctl` is not managed by the `systemctl` utility
- `systemctl` uses `pg_ctl` to start the instance
- waits for instance to start for 300 seconds (`TimeoutSec=300` parameter in service descriptor file)
- Launching an instance:
 - > `systemctl start tantor-se-server-17`
 - > `systemctl start tantor-se-1c-server-17`
 - > `systemctl start tantor-be-server-17`



Managing an instance via systemctl

In the Linux operating systems supported by Tantor Postgres, `systemd` is used to start services. Tantor Postgres is compiled with the `--with-systemd` option, which provides support for all `systemd` functionality. The distribution comes with a service description file `/usr/lib/systemd/system/tantor-se-server-17.service`, and the administrator does not need to create it. By default, `Type=forking` is used.

By default, a timeout of 5 minutes is set by the **`TimeoutSec=300` parameter** in this file. `systemd` **will kill the instance if it does not start within this time**. On production servers, recovering from a crash using logs can take a significant amount of time. The infinity value is recommended in such cases and disables the timeout logic.

While the server is running, its PID is stored in the `postmaster.pid` file in PGDATA. This is used to prevent multiple instances of the server from running in the same data directory and can also be used to shut down the server.

If the instance processes **are terminated** and the **`postmaster.pid` file** prevents the instance from starting, the `postmaster.pid` file **can be deleted**.

`systemctl` is the main command for working with `systemd`. By default, it runs with root user rights.

Launching an instance:

```
systemctl start tantor-se-server-17.service
```

The suffix `".service"` can be omitted, as it is used by default.

If the `systemctl` utility produces an error when running an instance:

```
Starting Tantor Special Edition database server 17...
```

```
pg_ctl: another server might be running; trying to start server anyway
```

```
lock file "postmaster.pid" already exists
```

```
HINT: Is another postmaster running in data directory
```

```
"/var/lib/postgresql/tantor-se-16/data"?
```

```
pg_ctl: could not start server
```

This may mean that the instance is not started via `systemd`, but by the `pg_ctl` utility, and `systemd` cannot start or stop the instance because it was started by the `pg_ctl` utility. You can check the list of processes in the operating system. `systemd` uses the `pg_ctl` utility for starting/stopping and other actions.

The `systemctl stop tantor-se-server-17` command in this case cannot stop the instance, **it does not produce a result and may create the false impression that the instance is terminated**.

You can check whether the instance has been added to autorun using the command

```
systemctl is-enabled tantor-se-server-17
```

The parent process has PID=1:

```
postgres@tantor:~$ ps -ef | grep init
root 1 0 0 /sbin/init splash
```

Working in a docker container

- Modifiable files, in particular PGDATA, must be located on volumes
- running in a container does not add high availability
- the 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 ps
  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 -v /root/data:/var/lib/postgresql/data --name postgres_container_name
```



Working in a docker container

The process ID (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 is started after the Linux kernel is initialized. The process with PID 1 spawns (starts) all other processes. It is the parent of all other processes that it spawns. All processes must have a parent process. Process 1 has a property: if the parent process of any process dies, the kernel automatically assigns process 1 as the parent of the orphaned process. Process 1 must adopt all orphans.

The postgres process monitors the state of its child processes and receives an exit status when any child process terminates. The normal behavior of the 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 parameter`.

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.

Three modes of stopping an instance

- `smart` - prohibits new connections and waits for existing sessions to voluntarily disconnect. This mode is not practical
- `fast` - new connections are denied, all server processes are sent a signal to abort transactions and exit. This is the best choice. This is the default.
- `immediate` - immediate shutdown mode



Three modes of stopping an instance

The instance can be stopped using the `pg_ctl stop` command .

Command syntax:

```
pg_ctl stop [-D data_dir ]  
[-ms[mart] | f[ast] | i[mmediate] ] [-W] [-t seconds ] [-s]
```

There are three modes to choose from:

`smart` - prohibits new connections and waits for existing sessions to voluntarily disconnect. This can take hours, while new connections are impossible, and this is downtime. In Oracle Database, this mode is called "shutdown normal". Thus, the `smart` mode is not practical. However, unlike Oracle Database, after sending a signal to stop in `smart` mode, you can send a signal to stop in `fast` mode. In Oracle Database, you can only shut down an instance in "abort" mode.

Therefore, if you have launched the `smart` mode, you have the opportunity to extinguish the instance in the `fast` mode.

`fast` - new connections are prohibited, all server processes are sent a signal to abort transactions and exit (linux signal **SIGTERM 15**). Then the remaining background processes of the instance are terminated in the correct order. One of the last actions is a checkpoint. In Oracle Database, this mode is called "shutdown immediate". Unlike Oracle Database, transaction rollback in PostgreSQL is performed immediately, so the delay in stopping is mainly determined by the duration of the checkpoint.

`fast` - **the default stop mode** for stopping via `pg_ctl stop` and via `systemctl stop`

On industrial clusters with a large amount of memory used by the instance, it is possible to minimize the instance stop time, i.e. the downtime. To do this, before stopping the instance, you need to initiate the execution of a checkpoint with the command `checkp o int`. After the checkpoint is executed, send a signal to stop the instance. In this case, the checkpoint, which will still be executed when the instance is stopped (the final checkpoint) in `smart` or `fast` mode, will have to write less data to disk and the final checkpoint will be executed faster.

In `smart` and `fast` modes, all changed data in memory (that needs to be saved, "protected by the write-ahead log") is written to files at the checkpoint, all files are synchronized to one point in time, and information about the successful shutdown of the instance is written to the **pg_control control file** . This is called a "clean shutdown". When the instance is subsequently started, the control file determines that the instance was stopped correctly and no WAL reading is required.

Stopping an instance

- An instance can be stopped using the `pg_ctl stop` command regardless of how the instance was started.
- Using `pg_ctl` is the most convenient and guaranteed way to kill an instance.
- You can send a signal to the postgres process directly:
`kill -INT `head -1 $PGDATA/postmaster.pid``
- a `SIGKILL` (9) signal to any instance process.
- `systemctl stop` does not guarantee that the instance will stop



Stopping an instance

immediate - immediate shutdown mode. The parent postmaster process will send an immediate stop signal (SIGQUIT 3) to all other processes and wait for them to terminate. If any process does not terminate within 5 seconds, it will be sent a SIGKILL (9) signal. Then the postmaster process itself will be terminated. This will "recover" (by replaying the WAL log) the next time the instance is started. It is recommended to use only in extreme cases, such as a hang (no disk activity, no progress) of a fast shutdown. In Oracle Database, this mode is called "shutdown abort".

Using `pg_ctl stop` is the most convenient way to shut down an instance, but you can send a signal to the postgres process directly:

```
kill -INT `head -1 $PGDATA/postmaster.pid`
```

Note that the quotes are backticks, not apostrophes.

It is not worth sending the SIGKILL (9) signal to the postgres process, since the shared memory and semaphores will not be released until the operating system is rebooted or until they are released manually with the `ipcrm` command. Also, server and background processes may remain in memory. You can view the shared memory segments and semaphores with the `ipcs` operating system command, and release them with `ipcrm`.

You should not send a SIGKILL (9) signal to other instance processes, including server processes (as is common when working with Oracle Database), as this may cause the instance to stop immediately.

To disconnect sessions and interrupt a running command (in another session without interrupting it) in PostgreSQL, it is convenient to use the functions `pg_terminate_backend` (sends SIGTERM 15 to the server process) and `pg_cancel_backend` (sends SIGINT 2).

Before performing procedures that require a correct shutdown, **you should make sure that:**

1) all processes of the stopped instance have been unloaded from memory (are not present in the operating system)

2) the status of the correct cluster shutdown was written to the control cluster:

```
pg_controldata | grep state
```

Database cluster state: **shut down**

https://docs.tantorlabs.ru/tdb/en/17_5/se/server-shutdown.html

Stopping an instance

- When performing checkpoints (parameter `log_checkpoints=on`), the cluster log will contain messages like:

MESSAGE: Checkpoint started: shutdown immediate

or

LOG: checkpoint starting: shutdown immediate

- The text in the "shutdown immediate" message refers to the checkpoint properties, not the instance shutdown mode. Stopping an instance in immediate mode (`pg_ctl stop -m immediate` command) does not perform a checkpoint.
- PostgreSQL does not have a shutdown immediate command



Stopping an instance

will contain messages like the following when performing checkpoints (`log_checkpoints=on` parameter):

MESSAGE: Checkpoint started: shutdown immediate

or

LOG: checkpoint starting: shutdown immediate

PostgreSQL does not have a shutdown immediate command.

The text "shutdown immediate" in the log refers to the checkpoint properties, not the instance shutdown mode. When stopping an instance in immediate mode

(command `pg_ctl stop -m immediate`) checkpoint is not executed.

Text in checkpoint messages (after **LOG: checkpoint starting:**) means:

shutdown - checkpoint caused by stopping the instance

immediate - execute the checkpoint at maximum speed, ignoring the value of the `checkpoint_completion_target` parameter

force : perform a checkpoint even if nothing has been written to the WAL since the previous checkpoint (there was no activity in the cluster), this happens if the instance is shut down or at the end-of-recovery

wait : Wait for the checkpoint to complete before returning control to the process that called the checkpoint (without `wait`, the process will run the checkpoint and continue running).

end-of-recovery : checkpoint at the end of log rolling (cluster recovery by startup process)

xlog : checkpoint caused by log files reaching half the size specified by `max_wal_size` ("by size", "on demand")

time : the checkpoint was triggered by reaching the `checkpoint_timeout` parameter value ("by time")

Management utilities (SQL command wrappers)

- are located in the directory `/opt/tantor/db/17/bin`
- directory path is included in `PATH` environment variable of postgres user in Linux
- Part of the command line utilities are wrappers for SQL commands
- Utilities and their corresponding commands:
 - `clusterdb` - `CLUSTER`
 - `createdb` - `CREATE DATABASE`
 - `createuser` - `CREATE ROLE`
 - `dropdb` - `DROP DATABASE`
 - `dropuser` - `DROP ROLE`
 - `reindexdb` - `REINDEX`
 - `vacuumdb` - `VACUUM`



Management utilities (SQL command wrappers)

In the directory `/opt/tantor/db/17/bin` the path to which is added for the user postgres to the environment variable `PATH` during installation there are utilities for working with the database cluster. We have considered the `initdb` utility. Next we will consider the main utility - the terminal client `psql`, which allows you to pass SQL commands for execution.

Some of the cluster administrator's actions are not performed by SQL commands (or are more convenient to perform) and command line utilities are supplied for such actions. We will consider some of them during the course.

Wrappers for some SQL commands (which can be sent to be executed by the `psql` utility). Sometimes it is convenient to perform actions in a database cluster using command line scripts, and in such scripts it is convenient to use wrappers instead of writing a command call via `psql`:

```
psql -c "SQL COMMAND"
```

There is no difference in the result between using shell utilities and SQL commands.

`clusterdb` - wrapper for SQL `CLUSTER` command

`createdb` - a shell for the `CREATE DATABASE` command. There is no difference between creating a database with this utility or with the command

`createuser` - wrapper for the `CREATE ROLE` command

`dropdb` - wrapper for the `DROP DATABASE` command

`dropuser` - wrapper for SQL `DROP ROLE` command

`reindexdb` - wrapper for SQL command `REINDEX`

`vacuum db` - wrapper for `VACUUM` command

`vacuum lo` - has nothing to do with vacuuming (`VACUUM`). `vacuumlo` is a utility that is easy to run periodically to remove (clean) orphaned large objects from cluster databases. There are different ways to automate the removal of orphaned large objects (for example, triggers), this utility is one of them. The "lo" extension contains the `lo_manage()` function for use in triggers that prevent orphaned large objects from appearing.

Description of utilities:

https://docs.tantorlabs.ru/tdb/en/17_5/se/reference-client.html

Backup Management Utilities

- `pg_archivecleanup` - used on replicas (standby clusters)
- `pg_basebackup` - creates physical backups
- `pg_combinebackup` - combines incremental backups with full backups
- `pg_dump`, `pg_dumpall`, `pg_restore` for logical backup (creating dumps)
- `pgcopydb` - Tanor Postgres utility for logical backup
- `pg_receivewal` - for creating streaming WAL archives
- `pg_resetwal` to change the size of WAL segments



Backup Management Utilities

`pg_archivecleanup` is used in the `archive_cleanup_command` parameter value to remove unwanted WAL files on the physical replica (standby cluster)

`pg_basebackup` - a utility for creating cluster backups (backups) for clones, replicas and just for storage. It can copy a directory or pull files over the network using the replication protocol

`pg_combinebackup` - combines incremental backups with full backups

`pg_dump` - creates a logical copy of database objects

`pg_dumpall` - creates a logical copy of the entire cluster or common cluster objects in the form of a text script for creating databases and objects. It is used in the procedures for updating the main version, migrating the cluster to other platforms, assemblies, and PostgreSQL forks. Of interest is the `-g` parameter, which allows you to dump common cluster objects.

`pgcopydb` - Tanor Postgres utility for automating data transfer at the logical level between databases with maximum speed, uses `pg_dump`, `pg_restore` utilities and logical backup techniques

`pg_receivewal` - used to retrieve WAL (stream archive) files via the replication protocol. Typically used to organize WAL log storage on nodes with backups.

`pg_recvlogical` - for logical replication, rarely used.

`pg_resetwal` clears the WAL log. Used with the `--wal-segsize` option to resize WAL segments if you want to change their size after cluster creation. The procedure requires care and knowledge of what will happen to backups and WAL file names. It is also critical for the procedure of resizing WAL segments that the cluster is shut down correctly. They are changed either because of a large number of files in one directory, or because the maximum size of the log buffer in shared memory (`wal_buffers`) is limited by the size of the WAL file. The effect of the WAL buffer size on performance is nonlinear.

`pg_restore` - utility for restoring from logical backups created by the `pg_dump` utility in some modes (in other modes, `psql` is used for restoration)

`pg_waldump` - shows the contents of WAL segments, used for debugging in complex recovery cases

https://docs.tanorlabs.ru/tdb/en/17_5/se/reference-server.html

Management utilities (other)

- `pg_checksums` - enable/disable calculation of checksums of data blocks and verification of cluster data blocks
- `pg_rewind` - for synchronizing clusters, for example, after failover to a physical replica and in upgrade procedures
- `pg_upgrade` - to upgrade to a new major version of the DBMS
- `pg_test_fsync` - measures the speed of writing to WAL segments in different modes
- `pg_config` - information about the parameters of installation and assembly of the DBMS
- `pg_controldata` - outputs the contents of the cluster control file `$PGDATA/global/pg_control` in text form



Management utilities (other)

`pg_amcheck` - refers to the standard extension (PostgreSQL extension) `amcheck` , which has a set of functions for checking for damage in objects in which data is physically stored, called relations. Relations (synonym "class") are called tables, indexes, sequences, views, external (foreign) tables, materialized views, composite types. If the `amcheck` functionality reports damage, then it really is there, false positives are excluded.

`pg_checksums` - enable/disable calculation of checksums of data blocks and verification of cluster data blocks. In Oracle Database, the equivalent is the `dbv` (`dbverify`) utility

`pg_rewind` - for synchronizing clusters, usually to restore a former master (primary cluster) after a failover to a physical replica (standby cluster), as well as in upgrade procedures (transition to a new primary version)

`pg_upgrade` - used when upgrading to a new major version of PostgreSQL, as well as when migrating from vanilla PostgreSQL to Tantor Postgres

`pg_test_fsync` - used when configuring WAL log writing parameters

`pg_test_timing` - measures the speed and stability of time stamp acquisition

Useful utilities

`pg_config` - information about the parameters of installation and assembly of the DBMS

`pg_controldata` - outputs the contents of the cluster control file `$PGDATA/global/pg_control` in text form

`pgbench` - the standard PostgreSQL utility for load testing

`pgcompacttable` - Tantor Postgres utility for reducing table file sizes

https://docs.tantorlabs.ru/tdb/en/17_5/se/reference-client.html

Management Utilities (continued)

- `pg_isready` - check that the cluster is accepting connections
- `oid2name` - a convenient utility for finding what object a file or directory belongs to
- `vacuum_maintenance.py` and other Python scripts are used by the `pg_partman` partitioning extension `pg_repack` - refers to the extension of the same name, which implements an analogue of `VACUUM FULL`, only without exclusive locking
- `pg_ctl` - manages the cluster instance, was discussed earlier
- `initdb` - creates a cluster, was discussed earlier



Management Utilities (continued)

`pg_isready` check that the cluster accepts connections, similar to `psql -c "\q"`. The utility is only more convenient to obtain the result, but in `psql` you can specify additional commands to check the availability of objects from the point of view of a specific client application.

`oid2name` is a convenient utility for finding which object a file belongs to in the cluster directory (PGDATA) and table spaces, as well as other information about the belonging of files and directories to cluster objects. Similar actions can be performed using SQL commands and SQL functions, but this is much more complicated.

`postgresql-check-db-dir` - script for a shallow check of the PGDATA directory structure, called by `systemd` before calling `pg_ctl` to start an instance, to make sure that the PGDATA directory contains something that looks like a cluster directory.

`vacuum_maintenance.py` and other Python scripts are used by the `pg_partman` extension for partitioning tables

`pg_repack` - an extension that allows you to reorganize files that store data without completely blocking the object. Analogous to the `VACUUM FULL` command, only without exclusive blocking.

Discussed earlier in this chapter:

`pg_ctl` - Manages a cluster instance

`initdb` - creates a cluster

https://docs.tantorlabs.ru/tdb/en/17_5/se/reference-server.html



1

1c

psql utility



Terminal client psql

- allows you to enter commands interactively
- It is possible to pass commands non-interactively - commands can be taken from a file or a command line parameter
- there are configuration files
 - global `/opt/tantor/db/17/etc/postgresql/psqlrc`
 - local by default `~/.psqlrc`
- Non-interactive execution of commands:
 - `psql -f script_file.sql`
 - `psql -c "CREATE SCHEMA sh; CREATE TABLE sh.t (n numeric);"`
- Description of psql command line options `--help`



Terminal client psql

PostgreSQL comes with a standard terminal client (command line utility) psql.

The course does not aim to monotonously describe all the capabilities of psql, there are many of them. psql functionality is wider than that of similar utilities in databases of other manufacturers. The following slides discuss features that may seem redundant, but **they are exactly what we encounter in everyday work** and simplify the solution of everyday tasks. In practice, additional examples are given for this chapter.

psql allows you to enter commands interactively, send them to the server process, and view the results of executing commands. You can also pass commands to psql non-interactively - commands can be taken from a file or a command line parameter.

```
psql -f script_file.sql
```

```
psql -c "CREATE SCHEMA sh; CREATE TABLE sh.t (n numeric);"
```

psql has configuration files. The global one is located in the directory pointed to by the `pg_config --sysconfdir` parameter.

for Tantor Postgres this is the file `/opt/tantor/db/17/etc/postgresql/psqlrc`

Local for the operating system user is located in his home directory, the default value is `~/.psqlrc`. The location of the local file can be overridden by the `PGCONFIG` environment variable.

By default, the files are not created, but you can create them. In Oracle Database, the `glogin.sql` file is used for sqlplus.

Both files can be made psql version specific by appending a hyphen and the PostgreSQL major or minor version identifier to the file name. For example, `~/.psqlrc-17` or `~/.psqlrc-16.8`. All files apply, but the more specific file takes precedence.

These files can be used to make working in psql more convenient.

https://docs.tantorlabs.ru/tdb/en/17_5/se/app-psql.html#psql

psql: connect to database

- psql connects to a specific database in a cluster
- To connect to the database, you need to pass authentication
- Authentication methods are discussed in the following chapters.
- Role and user are synonyms and absolutely identical concepts
- It is not possible to connect to several databases at the same time, even those located in the same cluster.



psql: connect to database

psql connects to a specific database in the cluster. To connect to the database, you need to pass authentication, which is usually configured separately for local connections via Unix sockets, network connections from the same host to the localhost address (127.0.0.1), and connections from other hosts. PostgreSQL supports various authentication methods, they will be discussed in the following chapters of the course. Authentication is possible without a password, but the session must be associated with a role (user) of the cluster. Connection without association with a role previously created in the cluster is possible only in single-user mode. In single-user mode, the connection is performed under a user who is implicitly granted superuser rights.

Role (ROLE) and user (USER) are synonyms and absolutely identical concepts. The commands `CREATE ROLE` and `CREATE USER` are absolutely identical.

After presenting the role name, the server process checks the privileges: can the role create a session (has the `LOGIN` attribute) with a specific database. The `SUPERUSER` attribute does not include the right to create a session; roles with both the `SUPERUSER` and `NOLOGIN` attributes can exist at the same time.

It is impossible to connect to several databases simultaneously, even from the same cluster. The databases are isolated from each other in terms of security and privileges. To work with tables in different databases simultaneously, even if they are in the same cluster, you can use the `postgres_fdw` (Foreign Data Wrapper) or `dblink` extensions. To copy data between databases, you can use streaming data transfer ("pipe") and the `pg_dump` utility ... | `psql` ...

psql: connection parameters

- Command line parameters for connection:
- -U **role** , defaults to operating system user name
- -d **database_name** , defaults to **role name**
- -h **host** , default /var/run/postgresql
- -p port, default 5432
- Teams:
- Command to display connection details:
`\conninfo`
- reconnect:
`\ c database_name role_name host by rt`
- parameters of the current connection can be passed by specifying the dash character "-"
`\c - - localhost`



psql: connection parameters

Command line parameters for psql that can be used to specify where and under what role to connect:

-U **role** or --username=**role** default value is the name of the operating system user under which psql is running

-d **dbname** or --dbname=**dbname** defaults to the role name specified by the -U parameter

-h **host** or --host=**host** default value /var/run/postgresql (on the instance side, the same value is set during assembly and is displayed in the `unix_socket_directories` parameter), i.e. a local connection via a Unix socket is used.

If psql or other utilities return an error

```
could not connect to server: No such file or directory
Is the server running locally and accepting
connections on Unix domain socket " /tmp/ .s.PGSQL.5432"?
```

It is possible that an old version of the utility is launched (for example, from the path /usr/bin/psql). The version is checked by `psql -V`

In addition to passing the -h parameter, you can specify the Unix socket directory in the `PGHOST` environment variable , for example, **export PGHOST=/var/run/postgresql**

-p port or --port=port the default value is 5432

for local connections via Unix socket the port is also used, since the directory is the same for all clusters. If this is a directory in the file system, then the main postgres process creates a file in it whose suffix is the port number.

For example, /run/postgresql/.s.PGSQL. **5432**

You can also use the shorthand syntax `psql database_name user_name` . For example, `psql postgres postgres`

A useful psql command to display connection details is: `\conninfo`

```
You are connected to database "postgres" as user "postgres" via socket in
"/var/run/postgresql" at port "5432".
```

The name of the role under which the connection was created (authentication passed) is returned. The `SET ROLE` and `SET SESSION AUTHORIZATION` commands do not change the result of `\conninfo`

To reconnect in psql use the command

```
\c database_name role_name host port
```

If you do not want to specify some parameters, but want to use the values of **the current** connection, then instead of the parameter in its position you need to use the dash symbol. You can omit the dash at the end. For example:

```
\c - user1
```

```
You are now connected to database "postgres" as user " user1 ".
```

```
\c - - localhost
```

```
You are now connected to database "postgres" as user " user1 " on host " localhost "
(address "127.0.0.1") at port "5432".
```

If a new connection cannot be established, the old one is maintained.

Getting help with psql commands

- psql commands start with a backslash \
- psql command line options --help
- psql command help \?
- list of SQL commands \h
after \h you can enter the initial words of a command and get help on it
- by default, if the result does not fit in the terminal window, paginated output is used
- for page-by-page output, the operating system utilities more or less are used, the less utility is more convenient
- the utility is installed by the command `\setenv PAGER 'less -XS'`
- paging mode is disabled by the command `\pset pager off`
- in page mode (after the colon):
q - exit, z - forward, b - back, h - help



Getting help with psql commands

After installing PostgreSQL, you can run psql on the server without parameters and psql connects locally via a Unix socket to the postgres database under the postgres role.

psql commands start with a backslash \

command line options --help

psql command help \?

list of SQL commands \h

after \h you can enter the initial words of a command and get help for that command

You can find out what SQL commands psql generates to execute commands starting with \d (describe - get a description of an object) by setting the parameter

`\ set ECHO_HIDDEN on`

If the text does not fit on the screen, the "pager" functionality is used, you will see a colon.

Pressing the <ENTER> key will display another line.

If you need to highlight the next page, then after the colon you need to type the symbol "z"

If you go back to the previous page - the symbol "b" (back)

if you want to interrupt the output you can type the symbol "q" (quit)

if you want to get help and find out what other key combinations there are, you can type the letter "h" (help) after the colon

You can disable pagination with the command `\pset pager off`

Pagination is implemented by passing the output result to the operating system's less utility.

The command history is accessible by default by pressing the up/down arrows on the keyboard. The history of commands typed interactively in psql is stored in the file `~/.psql_history` Its location can be overridden by the environment variables `HISTFILE` or `PSQL_HISTORY` , but there is no point in this. Next to `~/.psql_history` , for example, there is a file `~/.bash_history` with the history of the operating system terminal commands. File names starting with a dot are considered "hidden" files in Linux. For example, the `ls` command without parameters does not show such files.

psql can be run from a client machine from builds other than Tantor Postgres. psql works better with servers of the same or older major version. When connecting to a newer version of the DBMS, psql commands (those that begin with a backslash) may fail to work.

Formatting psql output

- By default, psql displays the query result with pseudo-graphics:

```
postgres=# select datname, datatemplate from pg_database;
 datname  | datistemplate 
-----+-----
 postgres | f
 template1 | t
 template0 | t
(3 rows)
```

- you can fine-tune the output format with the `\pset` parameters and `\a` `\t` `\x` switches
- view current formatting settings: `\pset`
- display command execution time, convenient for performance tuning `\timing on`



Formatting psql output

You can view the current formatting settings by typing `\pset`

If you need to repeat a command at intervals, and such a need for monitoring may arise for the administrator:

```
\watch seconds (exit CTRL+X)
```

```
\a enable/disable vertical column alignment;
```

```
\t enable/disable display of header and footer.
```

By default, columns are separated by a vertical bar, but you can set a different character, such as a space:

```
\pset fieldsep ' '
```

Disabling alignment and replacing the separator with the desired symbol allows you to output the selection result in a format convenient for transferring to a program working with tables.

When executing long queries and comparing execution speed, it is convenient to enable display of execution time:

```
postgres=# \timing
```

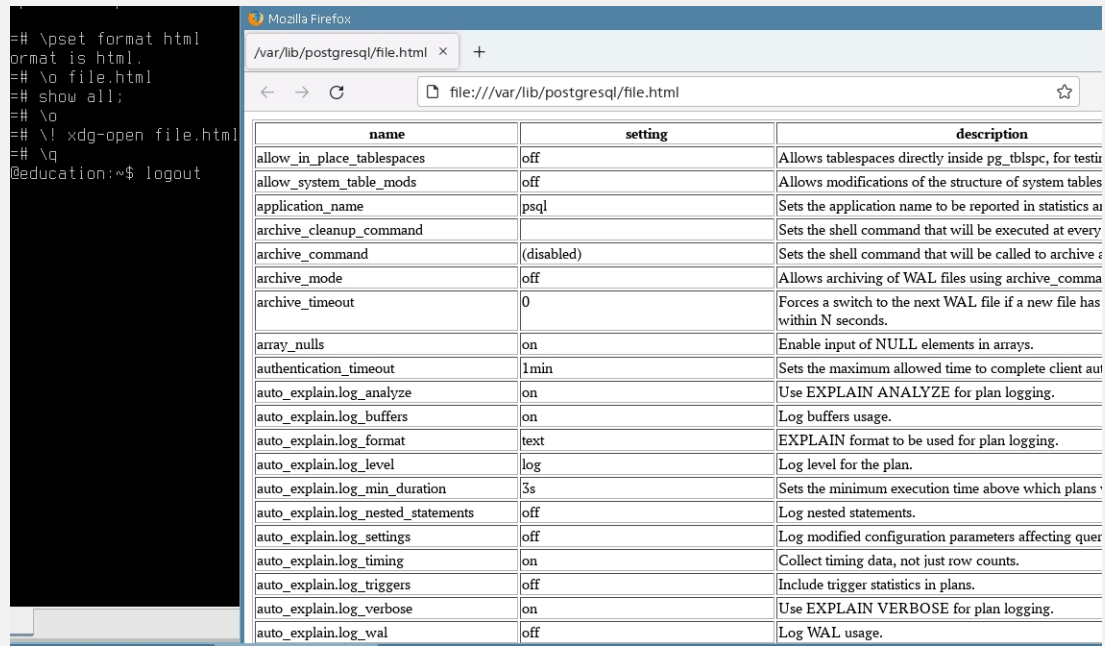
```
Timing is on.
```

```
postgres=# \timing
```

```
Timing is off.
```


Output the result in HTML format

- `psql -c "command;" -H -o f.html | xdg-open f.html`



name	setting	description
allow_in_place_tablespaces	off	Allows tablespaces directly inside pg_tblspc, for testing.
allow_system_table_mods	off	Allows modifications of the structure of system tables.
application_name	psql	Sets the application name to be reported in statistics and logs.
archive_cleanup_command		Sets the shell command that will be executed at every archive change.
archive_command	(disabled)	Sets the shell command that will be called to archive a WAL file.
archive_mode	off	Allows archiving of WAL files using archive_command.
archive_timeout	0	Forces a switch to the next WAL file if a new file has not been archived within N seconds.
array_nulls	on	Enable input of NULL elements in arrays.
authentication_timeout	1min	Sets the maximum allowed time to complete client authentication.
auto_explain.log_analyze	on	Use EXPLAIN ANALYZE for plan logging.
auto_explain.log_buffers	on	Log buffers usage.
auto_explain.log_format	text	EXPLAIN format to be used for plan logging.
auto_explain.log_level	log	Log level for the plan.
auto_explain.log_min_duration	3s	Sets the minimum execution time above which plans are logged.
auto_explain.log_nested_statements	off	Log nested statements.
auto_explain.log_settings	off	Log modified configuration parameters affecting query execution.
auto_explain.log_timing	on	Collect timing data, not just row counts.
auto_explain.log_triggers	off	Include trigger statistics in plans.
auto_explain.log_verbose	on	Use EXPLAIN VERBOSE for plan logging.
auto_explain.log_wal	off	Log WAL usage.

Output the result in HTML format

If the number of columns is large and the terminal client with proportional font is inconvenient for display, `psql` can generate the result not in text format, but in HTML format. This is done by the `-H` parameter or the `\pset format html` parameter.

An example of a command that sends an SQL command for execution and launches a browser with the result in HTML format:

```
psql -c "command;" -H -o f.html | xdg-open f.html
```

In one line you can get the result of large samples in a readable format.

This handy command may be more convenient and faster to execute than using graphical utilities like pgAdmin, and also in cases where graphical utilities are not installed on the operating system.

Output the result in extended format

- switched by the command `\x`

```
postgres=# \x
Expanded display is off.
postgres=# \x
Expanded display is on.
postgres=# select * from pg_settings limit 1;
-[ RECORD 1 ]-----
name          | allow_in_place_tablespaces
setting       | off
unit          |
category      | Developer Options
short_desc    | Allows tablespaces directly inside pg_tblspc, for testing.
extra_desc    |
context       | superuser
vartype       | bool
source        | default
min_val       |
max_val       |
enumvals      |
boot_val      | off
reset_val     | off
sourcefile    |
sourceline    |
pending_restart | f

postgres=# \x
Expanded display is off.
```



Output the result in extended format

If the result string has many columns or long field values, you can display the data row by row.

Switching the output is done with the short command `\x`

`\x` again

The query usually specifies sorting and limiting the number of rows returned: `ORDER BY` and `LIMIT` .

psql command prompt

- Why? To avoid issuing commands "in the wrong window"
- The invitation (prompt) can be changed using the commands

\set PROMPT1 and \set PROMPT2

```
postgres=# \set PROMPT1 '%[%033
\set PROMPT2 '%[%033[0;31m%]%n%
postgres@postgres=# begin;
BEGIN
postgres@postgres *=# select 1;
```

```
postgres=# begin transaction;
BEGIN
```

```
postgres=*# select
postgres-*# tantor_version();
        tantor_version
```

```
-----
Tantor Special Edition 1C 17.2.1
(1 row)
```

```
postgres=*# Selec;
ERROR:  syntax error at or near "Selec"
LINE 1: Selec;
        ^
```

```
postgres=!# commit;
ROLLBACK
postgres=# █
```

- "-" second and subsequent lines (PROMPT2)
- transaction is open and not committed
- "!" transaction is in a failed state and can only be rolled back, even if you type COMMIT;



psql command prompt

It happens that the administrator gave the command "in the wrong window". Changing the psql prompt helps reduce the likelihood of such cases.

The command prompt has default values that distinguish between the first line typed in a command and subsequent ones.

By default, PROMPT2 differs from PROMPT1 by invisible characters: **=** And **-** . It is worth paying attention to them.

PROMPT1, PROMPT2 and PROMPT3 define the appearance of the invitation.

PROMPT1 is issued when psql is waiting for a new command to be entered.

PROMPT2 if there is a string in the buffer, for example because the command was not terminated by a semicolon or the quotes were not closed.

A typical question is: what is the third prompt responsible for?

PROMPT3 is issued when executing the COPY FROM stdin command , when data is entered into the terminal to be inserted into a table. This mode is terminated by \. <ENTER>

This mode is rarely used, so the third prompt is not changed and people forget what it is responsible for.

During industrial operation, it is convenient to change these prompts in the ~/.psqlrc file to see which cluster database is connected.

Example of prompt installation:

```
\set PROMPT1 '%[%033[0;31m%]%n%[%033[0m%]%@%[%033[0;36m%]%/%[%033[0m%]
%[%033[0;33m%]%[%033[5m%]%x%[%033[0m%]%[%033[0m%]%R%# '
\set PROMPT2 '%[%033[0;31m%]%n%[%033[0m%]%@%[%033[0;36m%]%/%[%033[0m%]
%[%033[0;33m%]%[%033[5m%]%x%[%033[0m%]%[%033[0m%]%R%# '
```

```
postgres=# \set PROMPT1 '%[%033
\set PROMPT2 '%[%033[0;31m%]%n%
postgres@postgres=# begin;
BEGIN
postgres@postgres *=# select 1;
```

Autocommit transactions and running psql commands

- By default, psql operates in AUTOCOMMIT mode.
- Change the autocommit mode:
`\set AUTOCOMMIT on`
`\set AUTOCOMMIT off`
- view psql variables: command `\set`
- clearing the buffer of typed commands `\r`
- view buffer or last command if buffer is empty `\p`
- `;<ENTER>` terminates the SQL command and sends it to the server process for execution



Autocommit transactions and running psql commands

A command starting with a backslash " \ " is processed by psql. You can view help on such commands with the command `\?`

`\set` command can be used to view psql variables. Some variables are predefined and control psql operation. You can set your own variables for the time until you exit psql and use them as macros.

It is worth distinguishing the commands `\set` `\pset` `set` . The latter is related to SQL and changes the parameters of the server process at the session level (`set session`) or transaction (`set local`). `\pset` are predefined parameters for formatting the psql output.

The remaining commands are sent as text to the server process. To send a command, enter " ; " and a carriage return (the *<ENTER>* key on the keyboard).

Psql has non-standard commands `\g` `\gx` `\gexec` `\gset` `\g` that can replace the standard ";"

These non-standard commands have a lot of capabilities, but their use in scripts makes the scripts non-portable - the scripts will not be able to run anywhere except psql.

If you don't type " ; " but just type a carriage return, then psql considers the command to be multi-line and previous lines accumulate in the buffer.

If you want to clear the buffer, you can type `\r` (short for `\reset`)

View the contents of the buffer or the last command if the buffer is empty `\p` (short for `\print`)

`AUTOCOMMIT` mode by default . The default autocommit mode is also used in Java programs, in the JDBC specification. Oracle Database does not use autocommit mode in its terminal client `sqlplus` .

Autocommit mode means that psql implicitly sends a `COMMIT` command after each command that runs within a transaction (and together with such a command);

If you want to disable autocommit mode, you can disable this mode in the system `psqlrc` file or in your `~/.psqlrc` file or in your session. This is controlled by the

`\set AUTOCOMMIT on` parameter.

`\set AUTOCOMMIT off`

psql variables

- are set by the command `\set name value`
- Lifetime until psql exits or until `\unset name` is executed
- there are operating system environment variables that psql responds to
- they can be set in psql or in the parameter files (`~/.psqlrc`)
`\setenv PSQL_EDITOR /usr/bin/mcedit`



psql variables

Psql variables are set with the `\set name value` command. They live until psql exits or until `\unset name` is executed.

Variables can be used as macros. Variables can be referenced by prefixing them with a colon.

Example:

```
postgres=# \set TEST1 'select user'
postgres=# :TEST1;
user
```

```
postgres
(1 row)
```

```
postgres=# select * from (:TEST1);
```

By default, `vi` is used for editing commands `\ef` `\ev` `\e` . You can override the editor by setting an environment variable

```
export PSQL_EDITOR=/usr/bin/mcedit
```

Instead of `PSQL_EDITOR` you can use the names `EDITOR` or `VISUAL` .

Or, while in psql, give the command `\setenv PSQL_EDITOR /usr/bin/mcedit`

Or insert the command `\setenv PSQL_EDITOR /usr/bin/mcedit` in the file `~/.psqlrc` or global `/opt/tantor/db/17/etc/postgresql/psqlrc`

In the documentation section "Environment"

https://docs.tantorlabs.ru/tdb/en/17_5/se/app-psql.html#APP-PSQL-ENVIRONMENT

The operating system environment variables to which psql responds are specified.

Popular variables: `PGUSER` `PGDATABASE` `PGHOST` `PGPORT` . They allow you to configure psql connection without specifying parameters to any database.

Operating system environment variables can be set with the `\setenv` command , including in the `~/.psqlrc` file or the global `/opt/tantor/db/17/etc/postgresql/psqlrc` . Other commands like `\set` `\pset` `\!` `export` **environment variables** are not set.

Executing batch files in psql

- `\! linux_command` - execute an operating system command
- `\o file.sql` - redirect output to file
- `\o` - return output to screen
- `\i file.sql` - execute commands from file
- More examples of how to execute commands from a file:
`psql < file.sql`
`psql -f file.sql`
- Execute each line generated by the SELECT query as a command in psql:
`SELECT 'checkpoint;' \gexec`



Executing batch files in psql

In psql, you can execute an operating system command without exiting psql. To do this, use the command `\! linux_command`

To output the results of command execution (POSIX output stream) to an operating system file, you can use the `\o filename` command. The results will not be displayed on the screen.

To execute a batch file, you can use `\i filename`

```
\o checkpoint.sql
```

```
select 'checkpoint;' \g (tuples_only=on format=unaligned)
```

```
\o return output to screen
```

```
\i checkpoint.sql
```

You can also execute commands from a file (script) like this:

```
psql < checkpoint.sql
```

```
psql -f checkpoint.sql
```

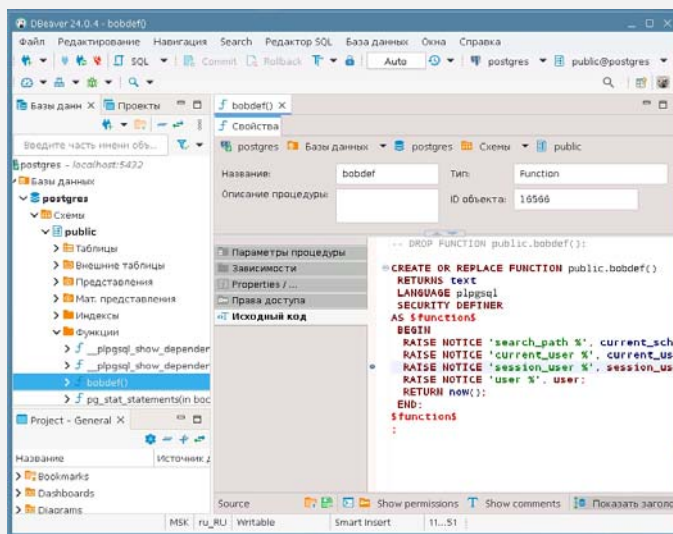
In this case, it is not necessary to put the exit command last in the file; psql will finish working itself when it reaches the end of the file (unlike the Oracle Database sqlplus utility).

Moreover, it is possible to form commands and execute them without creating an intermediate script file. For this, the `\gexec` option is used

```
postgres=# select 'checkpoint;' \gexec
CHECKPOINT
```

Graphical applications: DBeaver

- has a freely distributable version
- written in java and works in Astra Linux 1.8



Graphical applications: pgAdmin

The most popular universal (for development and administration) application is DBeaver, which has a free version.

The application can be downloaded using the command:

```
wget https://dbeaver.io/files/dbeaver-ce_latest_amd64.deb
```

and install with the command:

```
sudo dpkg -i dbeaver-ce_latest_amd64.deb
```

You can launch the application from the Start menu -> Development -> dbeaver-ce or with the command:

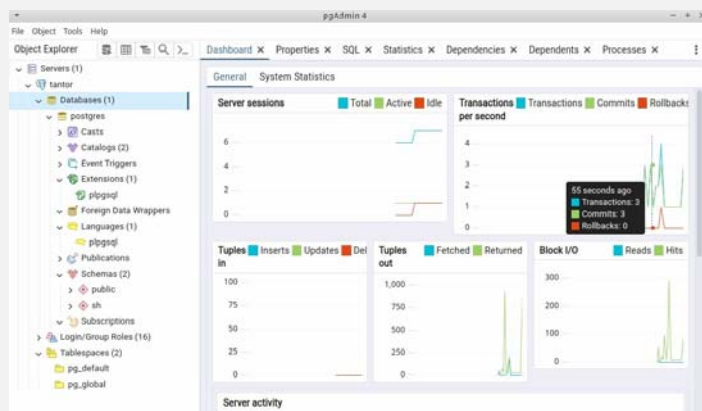
```
/usr/bin/dbeaver-ce
```

DBeaver allows you to debug stored procedures and functions using the pldebugger extension interface.

For application development, you can also use the commercial application DataGrip from JetBrains, which integrates with the company's development environments: IntelliJ IDEA and PyCharm. The integration allows you to check the syntax and auto-completion of SQL commands when writing program code.

Graphical applications: pgAdmin

- a freely distributed graphical program for working with PostgreSQL clusters
- pgAdmin works in Astra Linux 1.8



Graphical applications: pgAdmin

pgAdmin is a freely distributed graphical program for working with PostgreSQL clusters. pgAdmin gained popularity because it appeared before the DBeaver application was created.

The utility version 3 pgAdmin3 had a window interface, the development was completed in 2016. In version 4 pgAdmin4 has a web interface with the ability to create a link on the desktop. The utility allows you to use step-by-step debugging of stored routines - it is a client interface to the functionality of the freely distributed pldebugger library implementing the server part of the debugger functionality. The client interface is also the DBeaver application, written in Java.

pgAdmin3 does not work with PostgreSQL 15 and newer, because when connecting it accesses the datlastsysoid column of the pg_database table of the cluster system catalog, which was removed in version 15.

pgAdmin4 can be installed in Astra Linux 1.8

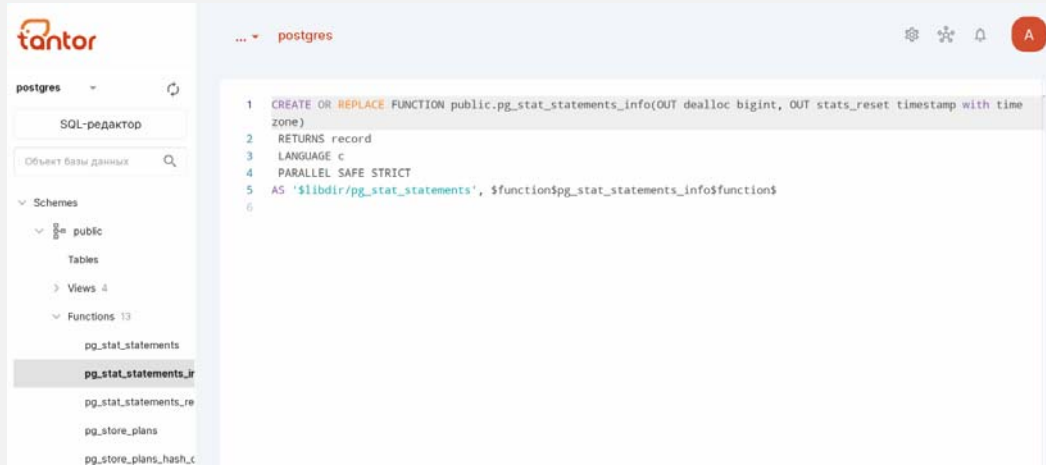
```
postgres@tantor:~$ sudo apt list | grep pgadmin
pgadmin4-desktop/stable 8.13-astra.sel+ci5 amd64 [upgradable from: 7.4-astra.sel+ci5]
pgadmin4-server/stable 8.13-astra.sel+ci5 amd64 [upgradable from: 7.4-astra.sel+ci5]
pgadmin4-web/stable 8.13-astra.sel+ci5 all
pgadmin4/stable 8.13-astra.sel+ci5 all
```

Run from menu Start-> Development ->pgAdmin 4

In the pgAdmin4 menu: in File->Preferences-> Paths -> Binary Paths--> PostgreSQL 17 set the path /opt/tantor/db/17/bin so that you can run the "PSQL Tool" from the Tools menu.

Graphic Applications: Tantor Platform

- application for administration and monitoring of large numbers of PostgreSQL clusters and Patroni clusters
- Enterprise Manager class application



Graphic Applications: Tantor Platform

The Tantor platform is software for managing any PostgreSQL-based DBMS and Patroni clusters. Allows you to conveniently manage a large number of clusters. Belongs to the class of software products that includes Oracle Enterprise Manager Cloud Control.

The Tantor platform is actively evolving to meet the needs of PostgreSQL administration.

The Tantor Platform has a SQL editor where you can view objects, execute commands, create procedures and functions.

https://docs.tantorlabs.ru/tp/5.3/instances/DB_browser.html

Demonstration

- Downloading the installer
- Setting permission to execute the installer
- Setting the location of distributions
- Installation with database creation
- Checking that the cluster is running
- Stopping services
- Uninstallation



Demonstration

Downloading the installer

Setting permission to execute the installer

Setting the location of distributions

Installation with database creation

Checking that the cluster is running

Stopping services

Uninstallation

Practice

1. Creating a cluster
2. Creating a cluster using the initdb utility
3. Single user mode
4. Passing parameters to an instance on the command line
5. Localization
6. Single-byte encodings
7. Using Management Utilities
8. Setting up the psql terminal client
9. Using the psql terminal client
10. Restore saved cluster



Practice

Creating a cluster
Creating a cluster using the initdb utility
Single user mode
Passing parameters to an instance on the command line
Localization
Single-byte encodings
Using Management Utilities
Setting up the psql terminal client
Using the psql terminal client
Restore saved cluster



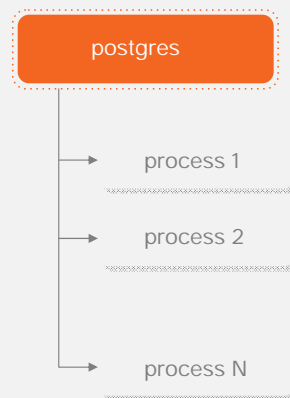
2a

Architecture



PostgreSQL instance

- PostgreSQL instance - the postgres process, the operating system processes it spawns, the memory these processes use
- example of processes:



```
postgres@tantor:~$ ps -eLo ppid,pid,cmd | egrep 'PPID|postgres'
PPID PID CMD
1 743184 /opt/tantor/db/17/bin/ postgres main process
743184 743185 postgres: logger process writing to logging collector
743184 743186 postgres: checkpointer background checkpoint process
743184 743187 postgres: background writer background writer process
743184 743189 postgres: walwriter background log writer process
743184 743190 postgres: autovacuum launcher autovacuum launcher process
743184 743191 postgres: pg_stat_advisor BackgroundTaskManager extension process
743184 743192 postgres: autoprewarm leader pg_prewarm extension process
743184 743193 postgres: logical replication launcher logical replication launcher process
644740 795748 psql -d demo -U alice -h /var/run/postgresql client, psql utility
743184 795749 postgres: alice demo [local] idle process serving psql
```



PostgreSQL instance

The postgres process (the obsolete name postmaster) is a process that services PostgreSQL (database server). This is the first process that starts, listens on network interface ports, and creates a Unix socket file through which it accepts local connections. This process starts (spawns, forks) other processes and is their **parent** process. These are server (traditional name - backend) processes that service client sessions and background processes that perform useful tasks to service the database cluster.

A PostgreSQL database cluster is a set of databases stored in the file system in the PGDATA directory as sets of files. One postgres process instance always serves only one database cluster, and a database cluster can be served by only one postgres process instance. Several postgres instances can run on a single physical or virtual host (in a single operating system), serving different database clusters. Postgres instances must use different ports of both network interfaces and different Unix socket files.

A PostgreSQL instance is the postgres process, the operating system processes it spawns, and the memory these processes use. Each process has local memory, which only that process has access to, and shared memory, which is accessible by multiple processes or even all processes in the instance.

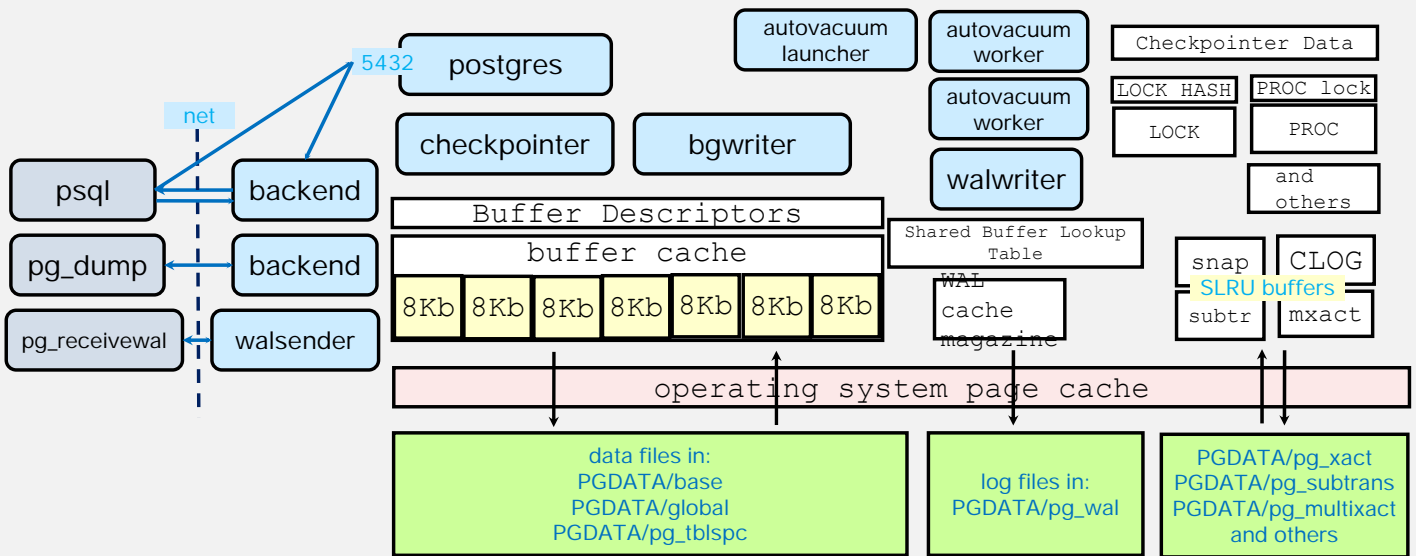
List of PostgreSQL instance processes:

```
postgres@tantor:~$ ps -eLo ppid,pid,cmd | egrep 'PPID|postgres'
PPID PID CMD
1 743184 /opt/tantor/db/17/bin/ postgres main process
743184 743185 postgres: logger process writing to logging collector
743184 743186 postgres: checkpointer background checkpoint process
743184 743187 postgres: background writer background writer process
743184 743189 postgres: walwriter background log writer process
743184 743190 postgres: autovacuum launcher autovacuum launcher process
743184 743191 postgres: pg_stat_advisor BackgroundTaskManager extension process
743184 743192 postgres: autoprewarm leader pg_prewarm extension process
743184 743193 postgres: logical replication launcher logical replication launcher process
644740 795748 psql -d demo -U alice -h /var/run/postgresql client, psql utility
743184 795749 postgres: alice demo [local] idle process serving psql
```

The client connected via a **Unix socket**, to the **demo** database under the user **alice**. The client is served by its server process with process number 795749. The rest of the instance processes are background.

PostgreSQL instance

- processes , memory, cluster



PostgreSQL instance

In PostgreSQL, there is no strict assignment of actions to processes. Server processes can read data files into memory (buffer cache), send blocks to the operating system for writing, send for writing from the log buffer to log files, perform vacuuming using the VACUUM command.

The main resources used by the instance are: disk, memory, processor, network. The most loaded resource is disk. To reduce the load, the data file contents are cached in the buffer cache. The buffer cache is a structure in shared memory, usually having the largest size, so the buffer cache and auxiliary memory structures and the processes that service it are given more attention. These are the checkpointer and background writer (bgwriter) processes. All changes to the data are made through the buffer cache, there are no direct changes to the data files. For temporary tables, an analog of the buffer cache is used, but only in the memory of the server process.

The buffer cache is a read/write cache (changes are held in memory). Fault tolerance is achieved by logging changes that are made to the data files through the buffer cache. The log is called WAL (Write Ahead Log) and consists of files of 16 MB (by default). The log files are written to by server processes and any other processes that make changes to the data, but there is also an auxiliary process called walwriter.

A set of background processes of autovacuum serves a separate task - deleting obsolete data.

The startup process stops after the recovery is complete.

The walsender processes are started when clients (pg_basebackup, pg_receivewal, walreceiver replica processes) connect via the replication protocol.

Starting an instance, postgres process

- the 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
- a `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, creates UNIX socket file
- The file with authentication parameters `pg_hba.conf` is read
- the startup process and background processes are launched



Starting an instance, postgres process

The basic steps to launch an instance are:

1. The postgres process ("postmaster") is started
2. Configuration parameter files are read, the parameters are combined with command line parameters 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` control file is checked, PGDATA is set as the current directory for the process, the `postmastr.pid` file is created in it, TLS is initialized, the shared libraries specified in the `shared_preload_libraries` parameter are loaded , a handler is registered in case the process disappears for the correct termination of child processes, the memory manager is initialized (according to the configuration parameters), and a handler for closing network sockets is registered.
In `postmaster.pid` the first line stores the PID of the running postmaster. **The file 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
6. The authentication settings file `pg_hba.conf` is read
7. The startup process is launched, which determines the cluster state using the `pg_control` control file (if the PGDATA directory was not restored from a backup, i.e. there is no `backup.label` file), and performs cluster recovery if necessary. The instance is opened for reading and writing if the cluster is not a physical replica (there is no `standby.signal` file).
8. While the startup process is figuring out what to do, postgres starts the rest of the background processes.
Server processes are started if there is a request to create a session from clients.
All spawned processes, including server ones, are periodically checked for existence.

Starting the server process

- for each session a server process is created
- 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 "portal" manager TopPortalContext
- the client is authenticated
- shared libraries specified in the `session_preload_libraries` and `local_preload_libraries` parameters are loaded
- MessageContext memory is allocated for the command text and the process is ready to receive commands



Starting the server process

The server process is started by the `postgres` process when a client wants to connect (a request was received on the server socket port or a Unix socket).

The main steps in starting a server process are:

1. When starting, the process gets the structure (part of memory) PGPROC from the list of free ones and sets the fields to the initial values. The structures are in shared memory. PGPROC is also used by background processes.

2. The process registers timeouts according to the values of configuration parameters, which can be viewed with the command:

```
psql \dconfig *_timeout
```

so that the server process can be interrupted when the values of these parameters are exceeded

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 "portal" manager TopPortalContext. A 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 a cursor) or unnamed - SELECT.

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` and `local_preload_libraries` are loaded. During the loading process, the compatibility of the libraries with the PostgreSQL version is checked. If the library was loaded earlier (`shared_preload_libraries`), then the process simply receives a pointer to the loaded library.

10. Memory is allocated for processing messages from the client

11. The ReadyForQuery message is sent to the client - the server process is ready to receive commands from the client.

Shared memory of instance processes

- more than 60 memory structures
- sizes and list - in `pg_shmem_allocations` view
- the size of some structures is specified by configuration parameters

```
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	
Archiver Data	147726208	8	128	128
...				
XLOG Recovery Ctl	4377728	104	128	128
	148145024	2849920	2849920	

(74 rows)



Shared memory of instance processes

Examples of structures in shared memory of an instance:

Proc Array, PROC, PROCLLOCK, Lock Hashes, LOCK, Multi-XACT Buffers, Two-Phase Structs, Subtrans Buffers, CLOG Buffers (transaction), XLOG Buffers, Shared Invalidation, Lightweight Locks, Auto Vacuum, Btree Vacuum, Buffer Descriptors, Shared Buffers, Background Writer Synchronized Scan, Semaphores, Statistics . There are more than 60 structures in total.

These structures are accessible by instance processes. Extensions can 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_size
<anonymous>		4946048	4946048	
Archiver Data	147726208	8	128	128
...				
XLOG Recovery Ctl	4377728	104	128	128
	148145024	2849920	2849920	

(74 rows)

A string with a NULL name reflects unused memory. A string with the name "<anonymous>" reflects the total size of structures for which memory was allocated without assigning a name.

The view does not show structures that are allocated and deallocated "dynamically" - as the instance runs. Dynamic shared memory structures are used by workers. Workers are used, for example, to execute SQL commands in parallel.

Only two instance shared memory structures can use HugePages : the buffer cache (size is set by the `shared_buffers` configuration parameter) and the memory allocated by background processes (memory is reserved for them by the `min_dynamic_shared_memory` configuration parameter).

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 refill it

```
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 is allocated in the local memory of each process in the context of CacheMemoryContext. When accessing 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 (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 restrictions on the size of CacheMemoryContext, it is not a circular buffer or a stack.

When a transaction commits that creates, deletes, or modifies 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 shmInvalBuffer ring buffer in shared memory. The buffer can store up to 4096 messages (the MAXNUMMESSAGES constant). 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.

Cache flush and message count statistics are not collected by standard PostgreSQL extensions.

View 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

PGDATA contains subdirectories that store cluster service data. To speed up read/write access to files in these directories, caches in the instance's shared memory are used. Files are formatted in 8K blocks. Caches operate using a simple least-recently-used (SLRU) algorithm. 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

In PostgreSQL starting with version 17 (in Tantor Postgres starting with version 15), SLRU 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/en/17_5/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 by 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 64MB. Sets the amount of memory allocated by each process (server, parallel) involved in executing the `VACUUM`, `ANALYZE`, `CREATE INDEX`, `ALTER TABLE ADD FOREIGN KEY` commands



Local process memory

Examples of structures in the local memory of the server process:

`RelationCache`, `CatalogCache`, `PlanCache`, `work_mem`, `maintenance_work_mem`, `StringBuffer`, `temp_buffers`

Local memory is accessible only to one process, so locks are not needed to access it. 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 the 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. Index creation and regular (without FULL) vacuum are parallelized. When vacuuming only in the index vacuuming phase (other phases are not parallelized), one index can be processed by one (not several) parallel processes. Whether parallel processes will be used depends on the size of the indexes.

Tantor Postgres has configuration options to customize local memory usage

`enable_temp_memory_catalog` and `enable_large_allocations`.

pg_backend_memory_contexts view

- shows the memory allocated by the server process serving the current session
- memory context - a set of memory chunks that are allocated by a process to perform a 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
- 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



pg_backend_memory_contexts view

The view shows the memory allocated by the server process servicing the **current** session. Memory contexts are a set of memory chunks that are allocated by the process to perform a 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 release some part when releasing memory, otherwise a memory "leak" will occur. When a memory context is released, all child memory contexts are released.

In the `pg_backend_memory_contexts` view, the hierarchy is represented by the columns: name (name of the memory context), parent (name of the parent memory context, level. The ident column contains details of what is stored in the context. An 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	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)

Память, выделенная текущему серверному процессу:

```
select sum(total_bytes), sum(used_bytes), sum(free_bytes) from
pg_backend_memory_contexts;
```

sum	sum	sum
2114816	1380760	734056

In version 18, the `id`, `parent_id`, `path` columns will be added to the view.

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(111);
 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);
```

The following messages will be displayed in 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
```

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 string at the end of the plan:

```
Memory: used=N bytes, allocated=N bytes
```

During the planning phase, when using a large (thousands) number of partitions of a partitioned table, a lot of memory can be used.

The memory under "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](#) (memory for) the transaction. After the transaction is executed, the transaction context memory is freed during the vacuuming process.

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
- the 20 bytes of data that BufferTag occupies is enough (no need to access 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.

Below 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 the cache line, which is usually 64 bytes in modern processors**. These 64 bytes contain:

1) the `BufferTag` structure, which specifies the direct (self-sufficient, i.e. storing everything needed to find a file and a block in it) address of the block on the disk:

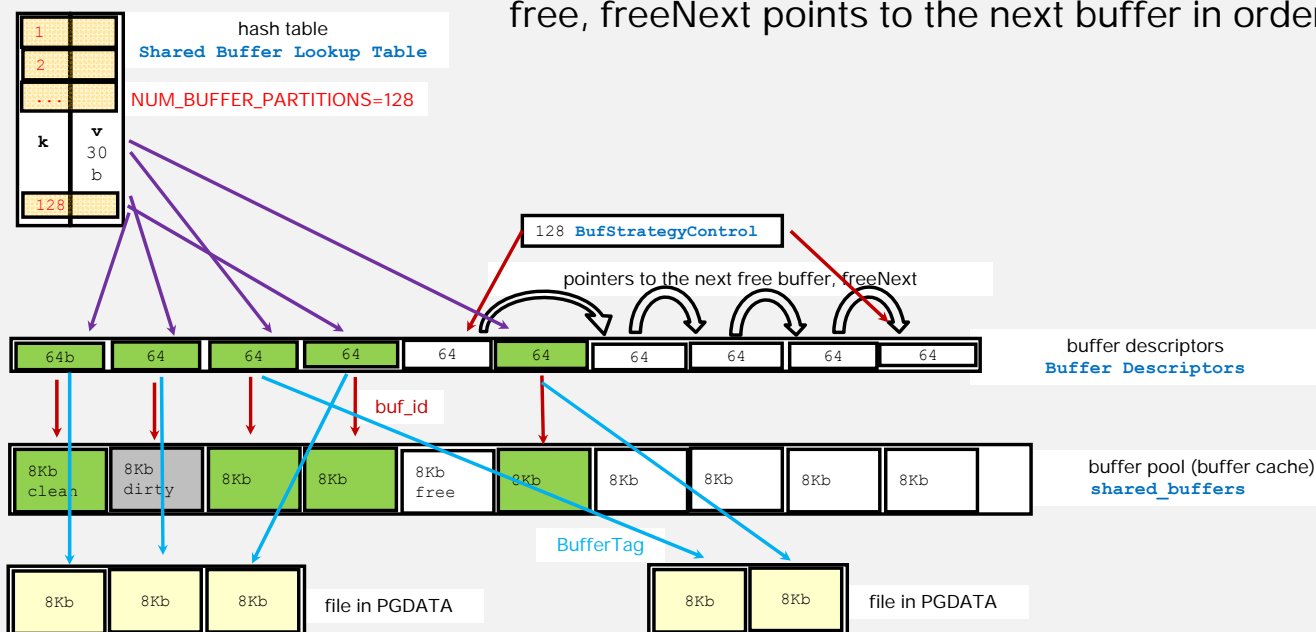
```
typedef struct buftag
{
    Oid spcOid; oid tablespace (symlink name in PGDATA/pg_tblspc)
    Oid dbOid; oid of the database (subdirectory)
    RelFileNumber relNumber; file name, represents a number
    ForkNumber forkNum; fork number (enum with 5 possible values: -1 invalid, 0 main, 1 fsm, 2 vm, 3 init)
    BlockNumber blockNum; block number relative to 0 block 0 of the file, size 4 bytes, maximum is set by the macro MaxBlockNumber
} BufferTag;
```

`BufferTag` size is 17 bytes. Size with alignment is 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 the instance is started, 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 dirty`
 - 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 pins to be released by other processes to lock the buffer for modification
 - 9 - `BM_CHECKPOINT_NEEDED` is 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 a block has changed while it is being written to disk, since shared locks are set during the writing process (an I/O 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 gone beyond the database horizon must wait until no other process is interested in the block in the buffer except itself, that is, the pincount is set to one by itself.

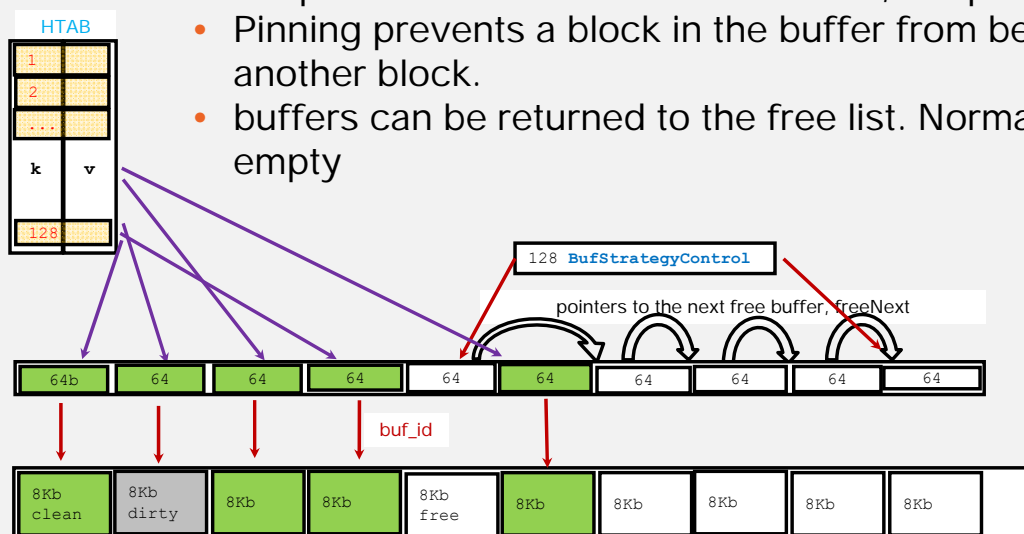
- 5) `int freeNext` - reference to the number of the next free block. **After the instance is launched, while all buffers are free, it points to the next buffer in order** . The logic of the linked list is used ("Linked List").

- 6) `LWLock content_lock` - lightweight lock on the buffer contents

Lightweight buffer content_locks are set by processes for a short time. Two types: Exclusive and Shared.

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 unpins it.
- 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

If a process wants to work with a block, it looks for it in the buffer cache. If it finds it, it pins it. Several processes can simultaneously pin a buffer. If the process does not need the buffer, the process removes the pin. Pinning prevents the block from being replaced in the buffer by another block.

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
take the value -1.
int lastFreeBuffer;
uint32 completePasses; used for statistics
pg_atomic_uint32 numBufferAllocs; used for statistics
int bgwprocno; bgwriter process number to notify
```

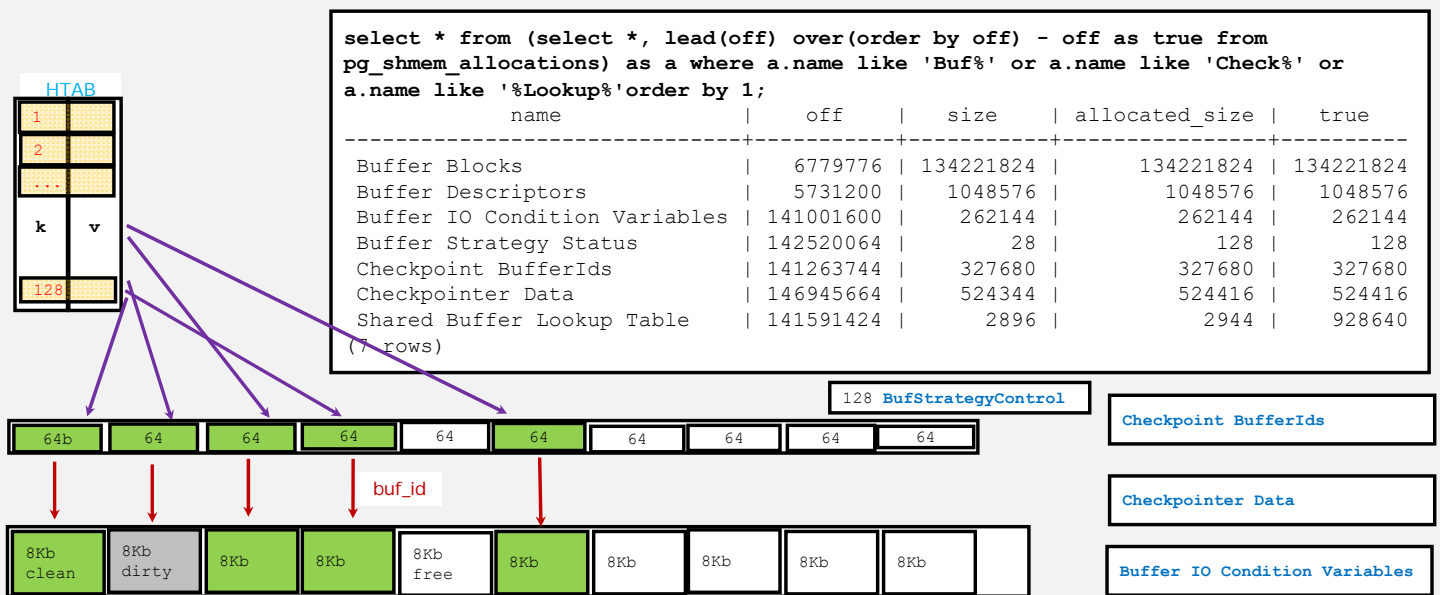
The word "strategy" is used in the sense of "method" from the phrase "buffer cache replacement strategy".

Buffers may be returned to the free list and the free buffer chain may be extended and even recreated after `firstFreeBuffer` has become -1 . This happens after deleting an object, or when deleting an individual file of an object, or truncating the file (including by vacuum), or when deleting a database.

One block cannot be in two or more buffers, it can only be in one buffer .

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

Checkpoint Data size: `CheckpointShmemSize()` = 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 incorrect. You can estimate the size of the structures by the off column (offset) from the neighboring structure:

```

select * from (select *, lead(off) over(order by off) - off as true_size from
pg_shmem_allocations) as a where a.true_size <> a.allocated_size order by 1;
    
```

name	off	size	allocated_size	true_size
...				

Shared Buffer Lookup Table	141706624	2896	2944	928640
----------------------------	-----------	------	------	--------

In the example, the size is 928640 bytes for a buffer cache size of 128 MB.

Buffer Replacement Strategies

- the number of entries in BufTable 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 . By default, 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

The number of records in BufTable is the sum of NBuffers and NUM_BUFFER_PARTITIONS due to the specifics of block handle initialization. For efficiency, the BufferAlloc() function, when loading a block into a buffer that was occupied by another block, first inserts a record with a reference to the new block handle into BufTable and only then frees the record in BufTable with a reference to the old block handle. To avoid a situation where there is no room in BufTable to insert a reference to the new block handle (which is difficult to process), additional space is immediately added to the table. Since parallel work with the BufTable contents is limited by the number of partitions, 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 StrategyType**) for replacing blocks in the buffer ring:

1) **BAS_BULKREAD**. For sequential reading of table blocks (Seq Scan), a set of buffers in the 256K buffer cache is used. The size is chosen so that these buffers fit into the second-level cache (L2) of the processor core. The ring should not be too small to fit all the buffers pinned by the process. Also, in case other processes want to scan the same data, the size should provide a "gap" so that the processes synchronize and simultaneously pin, scan, and unpin the same blocks. This method can also be used by commands that dirty buffers. Also, other processes can dirty 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 of the original database. **Buffer rings are not used for TOAST tables, since TOAST is always accessed 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 SELECT commands. 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:

- creates an instance of the BufferTag structure in its local memory
- computes a 4-byte hash from BufferTag
- **the Shared Buffer Lookup Table** hash table based on the hash value
- requests a lightweight (LWLock) lock of type BufMappingLock on the table partition that contains the hash value
- 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 the 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 the `BufMappingLock` type on the hash table partition into which the hash has fallen

3) calls the `BufTableLookup(BufferTag, uint32 hashcode)` function , 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.

The size of a record (hash bucket slot) in **the Shared Buffer Lookup Table** is 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 get locks on several parts, even on all parts. The lock is not held for long: the buffer header is read by the buffer number in the record without locks (`Buffer Descriptors`), the `refcount` (aka `ref_count`, 18 bits), `usage_count` (4 bits) are increased by an atomic operation (`pg_atomic_read_u32(&buf->state)`), 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.

Pinning the buffer (pin) and locking content_lock

- pinning 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 a buffer, a lightweight content_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 row to a block or change xmin, xmax of existing rows, a process must obtain a content_lock of 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

A pin 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 `Buffer Descriptors block descriptor`. The size of each block descriptor is 64 bytes (aligned). This lock must be held for a short time, unlike the 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 lines, content_lock can be removed, but the pin will not be removed, and in this mode the process will be able to read the block lines that it saw while the process had content_lock.

3. To add a new row to a block or change xmin, xmax of existing rows, a process must obtain a content_lock of the Exclusive type. With Exclusive, no one can have a Shared content_lock and, accordingly, see new rows that are in the process of being changed. Old rows can continue to be read, since they are not changing anyway: they cannot be cleared or frozen due to the event horizon retention.

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. It is not allowed to change the bits and xmin that relate to freezing, for this you need Exclusive content_lock and such changes are logged. And what about checksums? When changing any bits, the checksum will be different, but the checksum changes before writing the block to disk.

5. To remove the space occupied by the string (HOT cleanup or vacuum) after pin and Exclusive, the process waits for other processes to have no 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 the block, showing the intention to work with its contents, because they cannot load the 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 only one vacuum process can clean the table.

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 (unless file truncation is disabled)

```
pgbench --file=CreateAndDrop.sql -j 1 -c 1 -T 10
TPS for HP 128MB 417
TPS for HP 1GB 375
TPS for HP 4GB 227
TPS для HP 8GB 127
TPS для HP 16GB 55
TPS для HP 18GB 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 set on the buffer descriptor, the descriptor is released, and the SpinLock is released.

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 для shared_pool без Huge Pages (HP) размером 128MB - 433
```

```
1GB 367
```

```
4GB 220
```

```
8GB 123
```

```
16GB 43
```

```
18GB 32
```

The time to search for buffer descriptors in the hash table when deleting a small table increases 10 times when the buffer pool increases from 1 GB to 16 GB . Using Huge Pages does not change the speed significantly, since the buffer pool is not scanned.

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) for eviction to replace with another block
- "searching for a free block" is usually a candidate buffer for evicting from a block buffer, since all buffers are usually occupied and the free block list is empty
- when displacing a dirty block 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 checkpoint, bgwriter, server processes, and autovacuum worker processes. The bgwriter process writes dirty buffers and marks them as clean. The bgwriter reduces the likelihood that server processes will encounter dirty blocks when searching for a candidate buffer (victim) for eviction to replace with another block. When evicting a dirty block from a buffer, there is no access to the I/O bus, this is a copy from memory (buffer) to memory (Linux page cache). Delays are not as critical as it may seem. The bgwriter, walwriter, bgworker processes have similar names, but they are different processes. The bgwriter process is configured by the following 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 - how many milliseconds bgwriter sleeps 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 server processes have loaded into the buffer cache ("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 fastest 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, at zero value 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 "slow start", the iteration will scan at least:

$\text{NBuffers} / 120000 * \text{bgwriter_delay} + \text{reusable_buffers_est}$ blocks.

For a buffer cache size of 128 MB and 200 milliseconds of latency, this would be 27 + reusable_buffers_est blocks.

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



Clearing the buffer cache by the bgwriter process

The block is written by the `SyncOneBuffer(..)` function. First, a spin lock is taken on the block descriptor and the `BM_LOCKED` bit is set. The following values are checked: `refcount=0` (the block is not needed by processes), `usage_count=0` (falls into the gradation of long unused), `BM_DIRTY` bit=1 (dirty), `BM_VALID=1` and if the values are not as given, the spin lock is released 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 pin are released.

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 the Shared lock and pin.

The LSN is read from the block in the buffer and the `XLogFlush(XLogRecPtr record)` function is executed, 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 checksum calculation is enabled, the buffer contents are copied to the local memory of the bgwriter process by the `memcpy()` system call. The checksum is calculated on the local copy and this 8Kb copy is passed to the Linux kernel code, which places the block as 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 is calculating the checksum, and the checksum will be incorrect even if one bit is changed. 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 removed from the buffer descriptor and the buffer becomes "clean" and the block descriptor lock is released.

Since long unused (`usage_count=0` and `refcount=0`) buffers are flushed, the probability that the block will be needed by another process; 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.

The buffer is not included in the free list, the buffer becomes clear.

Checkpoint

- executes the checkpoint 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 checkpoint process. Checkpoints are performed: periodically, at the end of the instance stop and start procedure, replica promotion, backup, checkpoint command, database creation. Checkpoints are not initiated on the replica, but restart points are performed. In the event 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) stops 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 shutdown

`WAIT` (wait) return control only after the checkpoint is completed

`CAUSE_XLOG` (wal) by `max_wal_size` parameter 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 immediate force wait properties** .

Steps to perform a checkpoint

- If an instance is stopped, a status about the instance shutdown is written to the `pg_control` file
- 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 is terminated, a status about the instance shutdown is written to the `pg_control` file . The LSN of the next log entry is calculated. This will be the LSN of the checkpoint start, but checkpointer does not create a separate log entry about the start.

Other processes may set the `DELAY_CHKPT_START` flag. A list of virtual transaction identifiers is collected, the processes of which set the flag. If the list is not empty, then checkpointer waits for the flags to be removed in a loop, sleeping for 10 milliseconds between checks for flag removal. 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 creating a journal entry about the commit.

Then checkpointer starts dumping `slru` buffers and other shared memory structures to disk using `CheckPointGuts(..)` function into files that they cache and/or into WAL and synchronization is performed on these files (`fsync`). These log records should be related to the checkpoint and should come after its start LSN.

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 `IS_SHUTDOWN`, `END_OF_RECOVERY`, `FLUSH_ALL` type 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 at a time. Then it checks that the block is dirty and sets the `BM_CHECKPOINT_NEEDED` flag for dirty blocks, saves the block address in the `Checkpoint BufferIds` shared memory structure . Then it removes the `SpinLock`. The block address is the traditional 5 numbers of the `BufferTag` structure .

If some process clears the buffer, 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 the 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 using the standard quick sort algorithm. The comparison is performed by the `ckpt_buforder_comparator(..)` function in the order: tblspc, relation, fork, block. tblspc comes first, and this is significant. Sorting, in particular, is needed to avoid a situation where blocks are sent to table spaces in order, loading one table space at a time. It is assumed that table spaces 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 `SyncOneBuffer()` with periodic delays (according to the `checkpoint_completion_target` configuration parameter and the calculated write speed) 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. 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.

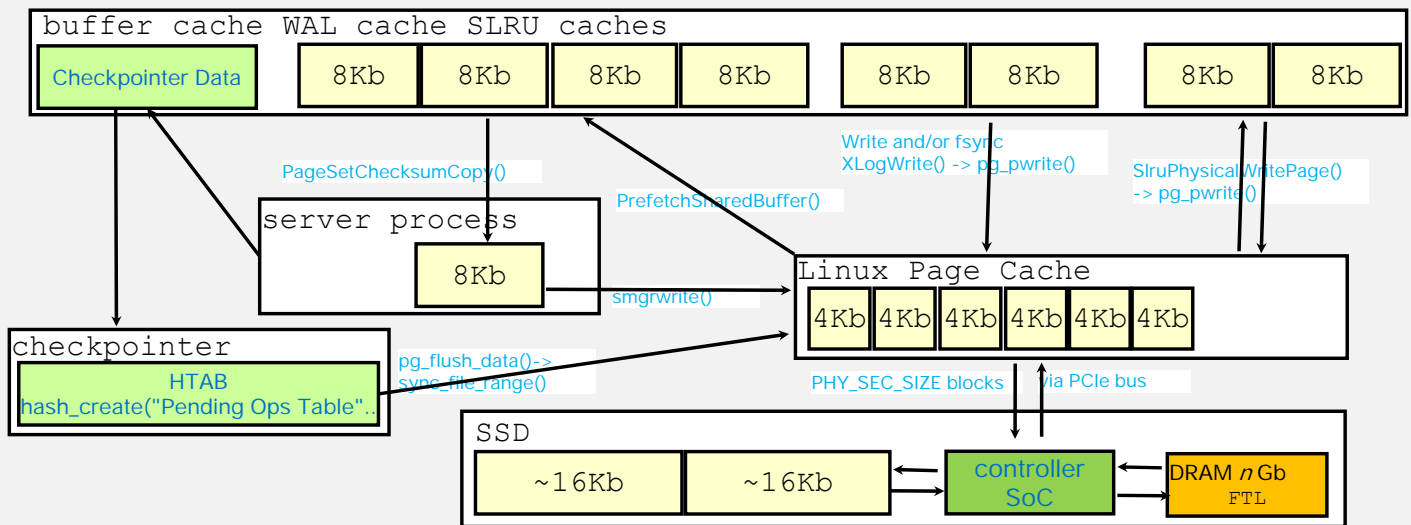
For checkpoints (except for the one performed on instance shutdown), a snapshot with a list of active transactions is saved in the WAL using the `LogStandbySnapshot()` function. This can be useful for replicas when restoring from archive logs.

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 `fdatsync` system call (or other 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 deleted. 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 WAL segments are allocated or old ones are cleared and renamed according to the configuration parameters.

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

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 made on block ranges. References to blocks to be synchronized (in the future) are written to a hash table of 100 blocks, created by `hash_create("Pending Ops Table"` or "`pending sync hash`".) in the local memory of the checkpointer, sorted by `sort_pending_writebacks(..)` to arrange the blocks for transferring the block range. **`fsync()` is executed once for each file (where at least one block has changed) at the end of the checkpoint.**

For synchronization, all files that have been modified since the last checkpoint must be remembered in order to synchronize before the next checkpoint is completed. A hash table (not a linked list) is chosen to eliminate duplication of commands (operations) to write the same block. Hash tables remember blocks that need to be synchronized. For file deletion commands, a linked list is used, since there should be no repeated file deletion commands (operations).

Processes submit operations to the checkpointer process through a shared memory structure `CheckpointShmemStruct` named "Checkpoint Data". The list of shared structures and their sizes are available in the `pg_shmem_allocations` view.

Temporary tables are not synchronized because they do not require fault tolerance.

Practice

1. Transaction in psql
2. List of background processes
3. Buffer cache, EXPLAIN command
4. Pre-Record Log
5. Checkpoint
6. Disaster Recovery



Practice

Transaction in psql
List of background processes
Buffer cache, EXPLAIN command
Pre-Record Log
Checkpoint
Disaster Recovery



2b PostgreSQL Architecture

Multiversion



Row multiversioning

- The table blocks store versions of rows, which are called tuples.
- Queries (SELECT) must return data at one point in time ("consistent"), which is called "read consistency"
- All row versions are physically stored in table files as close to each other as possible (in the same data file blocks)

id	col1	col2	col3
1	a	b	c
2	b	c	a

tuple, string, string version (synonyms)



Row multiversioning

The table blocks store versions of rows, which are called tuples. The latter name comes from relational theory, where tables are called relations, columns are called attributes, and column data types are called domains.

When you create a table in PostgreSQL, a data type with the table name is automatically created, in which the names and data types of the fields correspond to the names of the table columns and their data types. The data type is called composite, since it consists of fields of other data types.

Queries (SELECT) must return data at a single point in time ("consistent"), which is called "read consistency". While queries are running, rows can change and be deleted. To ensure read consistency, old versions of rows must be stored. If a query does not find a row version at the point in time it needs, it will fail with the error "snapshot too old". All row versions are physically stored in table files, if possible, next to each other (in the same data file blocks).

The second reason why old row versions are saved is for transactions. A transaction can update a row, generating a new row version. A transaction can be rolled back or committed. If a transaction is committed, the old row version is not needed by the transaction. If a transaction is rolled back, the old version is needed, but the new one is not needed. Therefore, all row versions generated in transactions must be stored at least until their completion. A feature of PostgreSQL is that if a transaction is rolled back, the row versions that it would have generated if it had been committed physically remain in the blocks and take up space, and are not cleaned up during a rollback. Therefore, transaction rollback in PostgreSQL is performed quickly. A rolled back (ROLLBACK) transaction is called aborted.

Storing row versions is called Multi-Version Concurrency Control (MVCC).

Tables

- an object in which data is stored
- several types: regular tables (heap tables, rows are stored in an unordered manner), unlogged, temporary, partitioned
- Extensions can create new ways to store data and access it
- the number and order of columns are specified 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 them. The Tantor Postgres SE DBMS has the `pg_columnar` extension .

The 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.

The fields for the added column have NULL values by default or are given 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;`

Deleting a column deletes the values in the fields of each row and the integrity constraints that include the deleted column. If the integrity constraint being deleted is referenced by a FOREIGN KEY, you can delete 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 value (if defined) and any integrity constraints that the column is a part of will be converted. It is better to remove integrity constraints before modifying the column type and add the constraints later.

To view the contents of a block, the functions of the standard `pageinspect` extension are used.

https://docs.tantorlabs.ru/tdb/en/17_5/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` - the oid of the table that physically contains the row. 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. `ctid` will change if the row is physically moved to a different block.

`tableoid` - the oid of the table that physically contains the row. Values are meaningful for partitioned and inherited tables. A quick way to find out the oid of a table, as it corresponds to `pg_class.oid`.

`xmin` - the transaction number (xid) that created the row version.

`xmax` - the 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 a row.

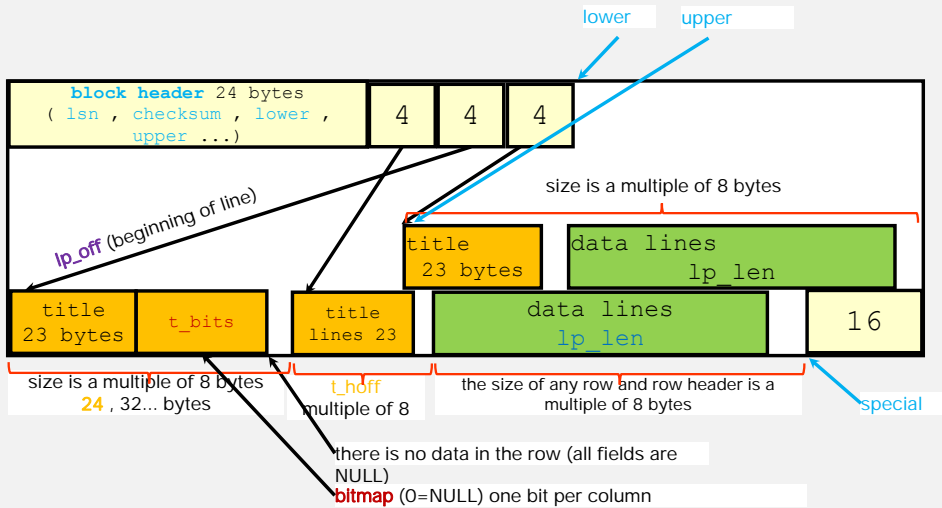
`cmin` is the zero-based sequence number of the command within the transaction that created the row version. Has no application.

`cmax` is the sequence number of the command within the transaction, starting from zero, that deletes or attempted to delete a row. To support "crooked" code, when the same row is updated several times in one transaction.

`xmin`, `cmin`, `xmax`, `cmax` are stored in three physical fields of the row header. `xmin` and `xmax` are stored in separate fields. `cmin`, `cmax`, `xvac` (VACUUM FULL was used before PostgreSQL version 9) in one physical field. `cmin` and `cmax` are interesting only during the life cycle of a transaction for insertion (`cmin`) and deletion (`cmax`). `ctid` is calculated based on the row address. Physically, the row version has `t_ctid`, which stores the address of the next (created as a result of UPDATE) row version. Moreover, this is not a "chain", the connection can be lost, since vacuum can delete a newer row version earlier than the old one (the block was processed earlier) and the old row version will refer to the missing version. If the version is the latest, `t_ctid` stores the address of this version. For partitioned tables, if the UPDATE resulted in the new version moving to another partition (the value of the column included in the partition key changed), a special value is set. Also, during the INSERT process, a "speculative insertion token" may be temporarily set instead of the row version address.

https://docs.tantorlabs.ru/tdb/en/17_5/se/ddl-system-columns.html

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
```

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 log record following the log record of which the block was changed. This LSN is needed so that the block is not sent for writing if the log record has not been written to disk (implementation of the write ahead log rule). It is also used for log recovery.

Tantor Postgres 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** . In PostgreSQL there is no special area for tables, index blocks have one.

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 an **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 of the line **length** (**l p _ len**). Two bits indicate four possible statuses of the pointer: 1 - points to the line, free, and two more statuses that implement HOT (heap only tuple) optimizations: dead and redirect.

If the table has up to 8 columns inclusive, the row header size is 24 bytes. If the table has 9 or more columns, then the size of the row header, **if at least one field contains an empty value** (NULL), becomes 32 bytes, and starting with 73 columns, the row header becomes 40 bytes.

Number of rows in a block, depending on the size of the data area in the row:

```
rows | size
-----+-----
226 | 8
185 | 16
156 | 24
135 | 32
119 | 40
107 | 48
97 | 56
```

String version header

- xmin - xid of the transaction that created the row version
- xmax - xid of the transaction that deleted the row version
- ctid - a link to the next version of this line
- bits - bitmap of empty values
- hoff - offset to the beginning of the line data
- lp_off offset to the beginning of the line

xmin 333	xmax 334	field3 0	ctid (0,2)	infomask2 0100000000000010	infomask 0000000100000010	hoff 24		string data \x0100000009666f6f	pad
xmin 334	xmax 0	field3 0	ctid (0,2)	infomask2 1000000000000010	infomask 0010100000000000 1	hoff 24	bits 10000000	string data \x01000000	pad

```
select * from heap_page_items(get_raw_page('t','main',0));
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
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 8144|    1|   32|  333|  334|    0|(0,2)|    16386|    258|   24|    |
2| 8112|    1|   28|  334|    0|    0|(0,2)|    32770|   10241|   24|10000000|
```



String version header

The row header has a size of 24, 32, ... bytes and is a multiple of 8 bytes. It stores `t_hoff` - the offset to the beginning of the row data. At the end of the header there will be a bitmap `t_bits` (the size is a multiple of a byte), if at least one field of the row is NULL. One bit - one column, 1 - NULL, 0 - 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`. Example of creating the second version of the row:

```
create extension pageinspect;
create table t(n int, c text);
insert into t values (1, 'foo');
update t set c = null;
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|xmin|xmax|ctid|infomask2|infomask|hoff|t_bits|
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 8144|    1|   32|  333|  334|(0,2)|    16386|    258|   24|    |
2| 8112|    1|   28|  334|    0|(0,2)|    32770|   1024|   24|10000000|
```

`lp_off` offset to the beginning of the string, with byte precision.

`lp_len` string length

The size of the row header is always a multiple of 8 bytes (aligned to 8 bytes) and can occupy 24, 32, 40 bytes. The size of the entire row (header + data) is also always a multiple of 8 bytes. For alignment, empty bytes (0x0000) are added to the end.

Insert row

- xmin - xid of the transaction that inserted the row
- xmax - zero
- ctid - self reference
- the transaction that inserted the row does not mark it in infomask as committed. This will be done when the row is subsequently accessed by another transaction or another query

xmin	xmax	field3	ctid	infomask2	infomask	hoff		string data	pad
333	0	0	(0,1)	0000000000000010	0000100 00000010	24		\x0100000009666f6f	

```
select * from heap_page_items(get_raw_page('t','main',0));
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
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 | 8144 | 1 | 32 | 333 | 0 | 0 | (0, 1) | 2 | 2050 | 24 |  |  |
```



Вставка строки

Пример вставки строки:

```
create extension pageinspect;
create table t (n int, c text);
insert into t values (1, 'foo');
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 | 8144 | 1 | 32 | 333 | 0 | (0,1) | 2 | 2050 | 24 |  |
select * from t;
n | c
---+---
1 | foo
(1 row)
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len| xmin| xmax| ctid| infomask2| infomask|t_hoff| t_bits |
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 | 8144 | 1 | 32 | 333 | 0 | (0, 1) | 2 | 2306 | 24 |  |
```

ctid is a system column that indicates the physical location of a row in a table block.

It consists of two numbers: (block_number, line_pointer), where block_number is the block number starting from zero, and line_pointer is the pointer number in the header of this block. btree indexes in leaf blocks store pointers to row versions as ctid. The analogue in Oracle Database is the service column (pseudo column) ROWID, but it is unique within the entire database.

The transaction that inserted the row does not mark in infomask that it was committed. This will be done when the row is subsequently accessed by another transaction or by another query. If the ninth bit is set, it means that the transaction that inserted the row (xmin) was committed.

Update line

- ctid of the previous version points to the address of the new version of the line
- ctid of the current version of the line points to itself
- xmax of the previous version is changed from zero to the transaction number that created the new row version

xmin 333	xmax 334	field3 0	ctid (0,2)	infomask2 0100000000000010	infomask 0000000100000010	hoff 24		данные строки \x0100000009666f6f	pad
xmin 334	xmax 0	field3 0	ctid (0,2)	infomask2 1000000000000010	infomask 0010100000000001	hoff 24	bits 10000000	data lines \x01000000	pad

```
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|xmin|xmax|ctid|infomask2|infomask|t_hoff|t_bits|t_oid
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 8144|      1|   32| 333| 334|(0,2)| 16386|    258|   24|      |
2| 8112|      1|   28| 334|   0|(0,2)| 32770|   10241|   24|10000000|
```



Insert row

Example of creating a second version of a row as a result of an update:

```
update t set c = null;
```

```
select * from heap_page_items(get_raw_page('t','main',0));
```

```
lp|lp_off|lp_flags|lp_len|xmin|xmax|ctid|infomask2|infomask|t_hoff|t_bits|t_oid
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 8144|      1|   32| 333| 334|(0,2)| 16386|    258|   24|      |
2| 8112|      1|   28| 334|   0|(0,2)| 32770|   10241|   24|10000000|
```

When inserting, a second version of the row appears. The data area of the second version includes all fields after the update, i.e. field values may be duplicated.

The ctid of the previous version points to the address of the new version of the row. The ctid of the current version of the row points to itself.

The xmax of the previous version is changed from zero to the transaction number that created the new row version.

If the transaction that performed the UPDATE is not committed, then the server processes of other sessions see all versions of the row, but they check that the second version in infomask does not have bits indicating that the transaction is committed or rolled back, and they access the CLOG structure in shared memory to check the transaction status. There they will see that the transaction is not committed or rolled back, but the process exists. Based on this, they will understand that the second version of the row cannot be issued (otherwise there will be a "dirty read") and return the first version of the row, also checking the transaction status. The status of the transaction (xmin committed) that created the first version of the row is already set in bit 9 of infomask.

infomask bits:

1 bit - there are empty values, 2 bits - there are variable-width fields, 3 - there are fields moved to TOAST, 4 - there are fields of the OID type, 5 - the row is locked in key-share mode, 9 - xmin committed, 10 - xmin aborted, 11 - xmax committed, 12 - xmax aborted, 13 - in xmax multitransaction, 14 - the current version of the row .

infomask2 bits:

from 1 to 11 bits - the number of fields in the row, 14 - key fields changed or the row deleted, 15 - Heap Hot Updated, 16 - Heap Only Tuple .

Delete line

- xmax of the current row version (xmax=0) is set to xid of the deleting transaction

xmin 333	xmax 334	field3 0	ctid (0,2)	infomask2 0100000000000010	infomask 0000000 1 00000010	hoff 24		string data \x0100000009666f6f	pad
xmin 334	xmax 335	field3 0	ctid (0,2)	infomask2 1000000000000010	infomask 0010100000000000 1	hoff 24	bits 10000000	string data \x01000000	pad

```
select * from heap_page_items(get_raw_page('t','main',0));
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
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 8144| 1| 32| 333| 334|(0,2)| 16386| 1282 | 24 | | |
2| 8112| 1| 28| 334| 335|(0,2)| 40962| 8449 | 24 |10000000|
```



Delete line

Example of deletion:

```
delete from t;
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|xmin|xmax|ctid|infomask2|infomask|t_hoff|t_bits|
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 8144| 1| 32| 333| 334|(0,2)| 16386| 1282 | 24 | | |
2| 8112| 1| 28| 334| 335|(0,2)| 40962| 8449 | 24 |10000000|
select * from t;
n | c
---+---
(0 rows)
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|xmin|xmax|ctid|infomask2|infomask|t_hoff|t_bits|
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 8144| 1| 32| 333| 334|(0,2)| 16386| 1282 | 24 | | |
2| 8112| 1| 28| 334| 335|(0,2)| 40962| 9473 | 24 |10000000|
```

If the line deletion had not been committed but rolled back, then after rereading the line, the infomask of the second version of the line would have been set to 10497 instead of 9473 (xmax aborted). After the rollback and vacuum:

```
select * from heap_page_items(get_raw_page('t','main',0));
lp|lp_off|lp_flags|lp_len|xmin|xmax|ctid|infomask2|infomask|t_hoff|t_bits|
--+-+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1| 2| 2| 0| | | | | | | | |
2| 8144| 1| 28| 334| 335|(0,2)| 40962| 10497 | 24 |10000000|
```

The first version of the string is freed, the second version of the string is moved to the end of the block. The second pointer points to the string.

Smallest data types: `boolean`, `"char"`, `char`, `smallint`

- the 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, the efficiency of index scanning using available methods (Bitmap Index Scan, Index Scan, Index Only Scan)
- Data types that take up the least space:
 - › `boolean` takes 1 byte
 - › `"char"` takes 1 byte, stores ASCII characters
 - › `char` takes 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, typplen from pg_type where
typctype='b' and typcategory<>'A' order by typplen,typalign,typname;
```

The `boolean` type takes 1 byte. The `"char"` type also takes 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, which means more characters than in ASCII:

```
drop table if exists t5;
create table t5( c1 "char" default '1');
insert into t5 values(default);
select lp_off, lp_len, t_hoff, t_data from
heap_page_items(get_raw_page('t5','main',0)) order by lp_off;
```

```
lp_off | lp_len | t_hoff | t_data
-----+-----+-----+-----
8144 | 25 | 24 | \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 is `lp_off` (start of line) the same? Because there is alignment of the entire line by 8 bytes and you need to remember about it. `"char"` is intended for use in system catalog tables, but can be used in regular tables. You need to consider how the column will be used. If for searching, then evaluate the efficiency of column indexing, composite indexes, the efficiency of index scanning by 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 'ə' will take 3 bytes: \x 07d18d
 - › if the field length is more than 126 characters, then the field header will become 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, not varchar. varchar is described in the SQL standard. For varchar, you can specify the dimension varchar(1..10485760). The dimension for text is not specified. The dimension works as a "domain" (restriction). Checking the constraint takes processor resources. Of course, if the constraint is important for the correct operation of the application (business rules), then you should not refuse them.

Place occupied:

1) The first byte allows us 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 'ə' will take 2 bytes: d18d.

3) Optional zeros. **Fields up to 127 bytes long are not aligned** . Fields from 127 bytes are aligned by pg_type.typalign (i = 4 bytes).

Example:

```
drop table if exists t5; create table t5(c1 text default '1',c2 text default
'ə', 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 | \x053107d18d03
```

Fields can be compressed and remain in the block. In example 05 07 03 - field length.

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 it is unloaded as text 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` is aligned to 8 bytes
- `int` for primary or unique key will limit the number of rows in the 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)`
- `numeric` (synonym `decimal`) can be used to store numbers, 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` is 8-byte aligned. Using `int` for a primary or unique key will limit the number of rows in a table to 4 billion (2^{32}). The number of fields that can be TOASTed is also limited to 4 billion (2^{32}), but this limit may be reached [earlier](#).

Sequences are used to generate values for the `smallint`, `int`, and `bigint` types, and there are synonyms `smallserial(serial2)`, `serial(serial4)`, `bigserial(serial8)`. These are auto-incrementing columns. Numeric types are signed, and if you use only positive numbers, `serial` uses [the range from 1 to 2 billion](#) (2147483647), not 4 billion.

The variable-length numeric type (synonymous with `decimal`), described in the SQL standard, can be used to store numbers. **The overhead is 4 bytes for storing 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, and 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

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 | \x70580939c1ca020070580939c1ca0200
8136 | 40 | 24 | \x7044c4bcc3ca020070580939c1ca0200
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.

The timestamp and timestamptz data types store time and date with microsecond precision and occupy 8 bytes. **Both types do not store time zones, and the values are physically stored in the same format.**

timestamptz stores data in UTC. The timestamp data type does not display the time zone, does not use the time zone, and stores the value as is (without conversion). 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 precision and also takes up 8 bytes, which is quite a lot.

timetz data type takes up **12 bytes**. The interval data type takes up the most space, at **16 bytes**. Because of their larger size, these two data types are not practical.

Data types for real numbers

- fixed length, floating point, rounded to 6 or 15 digits (significant numbers in decimal format):
 - > float4 , real , float(1..24) - stores at least 6 digits in 4 bytes
 - > float8 , float , double precision , float(25..53) stores at least 15 digits in 8 bytes

```
select 12345 6 78901234567890123456789.1234567890123456789::float4::numeric;  
1234570 000000000000000000000000  
select 12345678901234 5 67890123456789.1234567890123456789::float8::numeric;  
123456789012346 0000000000000000
```

- variable length, without loss of accuracy in calculations:
 - > numeric, decimal
 - > precision can be set by parameters: numeric(precision, scale)

```
select 1234567890123456789.123456789::numeric + 0.000000000000000000 0123456789 ::numeric;  
1234567890123456789.12345678900000000000 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 double precision synonym float(25..53)
- 3) numeric synonym decimal

float4 provides 6 digits of precision (significant numbers in decimal notation), float8 provides 15 digits of precision. The last digit is rounded:

```
select 12345 6 78901234567890123456789.1234567890123456789 ::float4::numeric;  
12345 7 000000000000000000000000  
select 12345678901234 5 67890123456789.1234567890123456789 ::float8::numeric;  
12345678901234 6 0000000000000000
```

The sixth and fifteenth digits 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 0000000000000000
```

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
```

Snapshot

- represents a consistent state of the database at a point in time
- The data snapshot includes:
 - › xmin - the number of the oldest active transaction
 - › xmax - a value that is one greater than the number of the last completed transaction
 - › xip_list - list of active transactions
- A function that returns the contents of a snapshot and a function that exports it for another session:

```
postgres=# BEGIN TRANSACTION;
postgres=# select pg_current_snapshot();
pg_current_snapshot
-----
362:362:
postgres=# select pg_export_snapshot();
pg_export_snapshot
-----
00000024-0000000000000000A-1
```



Snapshot

There may be multiple versions of the same row in data blocks. Each transaction, despite the existence of multiple versions, sees only one of them. A snapshot provides isolation between transactions by providing them with an image of the data at a certain point in time, even if multiple versions of the same row may physically exist in the database.

The picture represents the numbers:

The lower bound of the snapshot , xmin, is the number of the oldest active transaction. All transactions with lower numbers have already been completed (committed), and their changes are reflected in the snapshot, while transactions with higher numbers may have been undone, and their changes are ignored.

The upper bound of a snapshot , xmax, is a value one greater than the number of the last completed transaction. This defines the point in time at which the snapshot was created. Transactions with numbers greater than or equal to xmax are not yet completed or do not exist, and therefore changes associated with such transactions are not reflected in the snapshot.

The list of active transactions , xip_list (list of transactions in progress), includes the transaction numbers of all active transactions, except virtual ones, which do not affect data visibility.

A function that returns the contents of a snapshot and a function that exports it for another session:

```
postgres=# BEGIN TRANSACTION;
postgres=# select pg_current_snapshot();
pg_current_snapshot
-----
362:362:
postgres=# select pg_export_snapshot();
pg_export_snapshot
-----
00000024-0000000000000000A-1
```

Transaction

- Transaction - a set of commands
- Begins explicitly or implicitly
- It is terminated by one of two actions: committing (COMMIT, END commands) or rolling back (ROLLBACK command)
- The result of an aborted transaction is the same as a rolled back command.
- Example of implicit transaction start:

```
postgres=# do $$
begin
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
perform 1;
end $$;
ERROR: SET TRANSACTION ISOLATION LEVEL must be called before any query
CONTEXT: SQL statement "SET TRANSACTION ISOLATION LEVEL REPEATABLE READ"
PL/pgSQL function inline_code_block line 3 at SQL statement
```

```
postgres=# do $$
begin
ROLLBACK;
SET TRANSACTION ISOLATION
LEVEL REPEATABLE READ;
perform 1;
ROLLBACK AND CHAIN;
perform 1;
COMMIT AND NO CHAIN;
end $$;
DO
```



Transaction

Transaction - a set of SQL commands. Starts explicitly or implicitly

It is terminated by one of two actions: committing (COMMIT, END commands) or rolling back (ROLLBACK command)

The result of an aborted transaction is the same as a rolled back ROLLBACK command. A transaction is started explicitly by the BEGIN TRANSACTION command or implicitly - in the **plpgsql block** :

```
postgres=# do $$
begin
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
perform 1;
end $$;
ERROR: SET TRANSACTION ISOLATION LEVEL must be called before any query
CONTEXT: SQL statement "SET TRANSACTION ISOLATION LEVEL REPEATABLE READ"
PL/pgSQL function inline_code_block line 3 at SQL statement
```

The transaction was started in an anonymous plpgsql block.

To change the isolation level in an anonymous plpgsql block, you need to roll back the transaction or commit it:

```
postgres=# do $$
begin
ROLLBACK;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
perform 1;
end $$;
DO
```

In PostgreSQL, you can execute not only select, insert, update, delete, but also almost all commands in transactions. Including create, alter, drop, truncate. You cannot execute commands that generate transactions on their own: vacuum, create/drop database. Example:

```
do $$
begin
begin
drop table if exists a;
create table a ( id int);
end;
rollback and chain;
drop table if exists a;
commit and no chain;
drop table if exists a;
rollback and chain;
end $$;
```

Transaction Properties

- Atomicity - when committing, all commands are executed without exceptions, when rolling back - no commands are executed
 - › Once committed, changes are instantly visible to other sessions
- Integrity - absence of violation of declarative integrity constraints
- Isolation of transactions from each other
 - › It is implemented by one of the isolation levels
- Fault tolerance (Durability) - if the client has received confirmation of the successful transaction commit (COMMIT COMPLETE), then it can be sure that the transaction result will not be lost



Transaction Properties

The value of executing commands in transactions lies in the "ACID" properties of transactions:

Atomicity - when committing, all commands are executed without exception, when rolling back - not a single command is executed. Moreover, changes from the moment of committing are instantly visible to other sessions.

Consistency - absence of violation of declarative integrity constraints.

Isolation of transactions from each other. In SQL, it is implemented by one of the isolation levels and locks (at the row and object level).

Fault tolerance (Durability) - if the client has received confirmation of successful transaction commit (COMMIT COMPLETE), then it can be sure that the transaction result will not be lost. This is guaranteed by the PostgreSQL software and the database cluster administrator. The administrator is required not to restore the cluster to a point in time in the past, not to change the fault tolerance parameters (`fsync`, `full_page_writes`, `synchronous_commit`). To protect against cluster loss, the administrator should ensure proper cluster backup. For example, have a synchronous physical replica or the `pg_receivewal` process confirming transaction commits.

If the client sends a COMMIT command but does not receive a confirmation that the transaction has been committed, the transaction may or may not be committed. Such cases must be resolved by the application; there is no standard way to determine the status of a transaction. Oracle Database uses the Transaction Guard and Application Continuity options for such cases.

Transaction Isolation Levels

- READ UNCOMMITTED - reading uncommitted data. Not used in PostgreSQL
- READ COMMITTED - read committed data. Used by default
- REPEATABLE READ - repeatability by reading. Should be used only for reading, not changing data (READ ONLY)
- SERIALIZABLE - ordered execution
 - › at REPEATABLE READ and REPEATABLE READ levels, if the data has changed, a serialization failure is possible: "can't serialize access", the transaction goes into a failure state and cannot commit



Transaction Isolation Levels

Isolation levels determine the degree to which changes made by one transaction are visible to other transactions.

The SQL standard defines four isolation levels:

READ UNCOMMITTED - Reading uncommitted data: This is the lowest isolation level. It allows transactions to see changes made by other transactions, even if those changes have not yet been committed. Not supported in PostgreSQL, **READ COMMITTED** is used instead.

READ COMMITTED - read committed data. **SELECT** statements see data that was committed at the time the **SELECT** began executing.

REPEATABLE READ repeatability of reading data. **SELECT** commands in one transaction do not see changes committed by other transactions after the start of their transaction. They see changes made only in their transaction. The first command starts the transaction, and it forms a snapshot that is used until the end of the transaction. The snapshot and **SELECT** commands do not lock rows.

SERIALIZABLE (ordered, sequential execution): When transactions of this level are executed simultaneously (with overlapping time), they must produce the same result as if they were committed one by one in **all** permutations of the commit time. This is the highest level of isolation of transactions from each other. To ensure that the result does not change, all transactions that change the data used in the transactions must operate at this level.

At the **REPEATABLE READ** and **REPEATABLE READ** levels, if the data has changed, a serialization failure is possible: "can't serialize access", the transaction goes into a failure state and cannot commit, it must be rolled back.

Transaction Isolation Phenomena

- ISO SQL-92 and subsequent standards define three concurrency phenomena that must be avoided at isolation levels
- integrity constraints cannot be violated at all levels
- dirty reads are not allowed in PostgreSQL at any isolation level, so Read uncommitted is the same as Read committed

Insulation level	Dirty Reading (P1)	Non-repeatable read (P2)	Phantom Reading (P3)	Serialization Violation
Read uncommitted	Allowed, but not in PG	Maybe	Maybe	Maybe
Read committed	No	Maybe	Maybe	Maybe
Repeatable reading	No	No	Allowed, but not in PG	Maybe
Serializable	No	No	No	No



Transaction Isolation Phenomena

ISO SQL-92 and subsequent standards define three concurrency phenomena (isolation of concurrent transactions) that must be avoided at isolation levels.

Serialization violation (not a phenomenon, but a consequence of the description of the Serializable level) is when the result of a successful commit of overlapping transactions turns out to be different for all possible variants of committing these transactions in turn. Also, integrity constraints cannot be violated at all levels. Moreover, integrity constraints do not depend on isolation levels.

A synonym for non-repeatable read (P2) is fuzzy read.

Dirty reads are not allowed at any isolation level in PostgreSQL, so Read uncommitted is the same as Read committed.

Non-repeatable read - When re-reading the same data that was previously read by the same transaction, it is discovered that the data has been modified and committed by another transaction.

There are no other phenomena described in the ISO SQL standards.

At all levels, changes should not be lost (no updates will be lost). In ANSI SQL, similar anomalies were mentioned: lost update (P4) and cursor lost update (P4C), which were allowed at the Read committed level. Lost updates are not allowed in PostgreSQL, since the UPDATE, DELETE commands, when encountering a locked row after releasing the lock, reread the row fields and see the changes of other transactions made after the start of the UPDATE, DELETE command.

https://docs.tantorlabs.ru/tdb/en/17_5/se/transaction-iso.html

Example of serialization errors

- Example of serialization error:

```
postgres=# drop table if exists a;
DROP TABLE
postgres=# create table a (x int);
CREATE TABLE
1 postgres=# begin transaction isolation
level serializable;
BEGIN
3 postgres=# insert into b select count(*)
from a;
INSERT 0 1
5 postgres=# commit;
COMMIT
```

```
postgres=# drop table if exists b;
DROP TABLE
postgres=# create table b (x int);
CREATE TABLE
2 postgres=# begin transaction isolation level
serializable;
BEGIN
4 postgres=# insert into a select count(*) from b;
INSERT 0 1
6 postgres=# commit;
ERROR:  could not serialize access due to read/write
dependencies among transactions
DETAIL:  Reason code: Canceled on identification as
a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

- In Oracle Database at level SERIALIZABLE errors net



Example of serialization error

There are descriptions of concurrent access anomalies, which are given names like read skew (A5A), write skew (A5B), but such anomalies are subjective - for some the result of committing transactions is unexpected (anomalous), for others it is expected.

For example, in Oracle Database, the Serializable level is understood as if there are no parallel sessions, the transaction sees the data at the time of its start, changes made by other transactions are not visible to it (as if they do not exist). Based on this definition, in Oracle Database, SERIALIZABLE level transactions can simultaneously perform INSERT into a SELECT table without receiving errors. In PostgreSQL, a serialization error will be issued. You can subjectively consider this to be a "lost insert anomaly". But Oracle Database is practical and the ability to insert rows with a selection from other tables is considered logical and does not lead to a violation of business logic. In PostgreSQL, the second transaction returns a serialization error, although in the example the result of committing the transactions does not depend on the order in which they are committed: in any order, two rows are created with the values 0 and 1. When checking for errors at the SERIALIZABLE level, PostgreSQL uses "predicate locks" (**SIReadLock** , Serializable Isolation Read Lock), which do not thoroughly check for serialization violations, but return an error if one is potentially possible. If you see a lock with this name, then there are transactions of this level:

```
select locktype, relation::regclass, mode from pg_locks;
locktype | relation | mode
-----+-----+-----
relation | b       | SIReadLock
```

Oracle Database does not have the Read uncommitted level, as does PostgreSQL, and instead of the Repeatable Read level, READ ONLY is used with read repeatability, which reduces the number of errors when developing the logic for using transactions.

CockroachDB and YDB use the Serializable isolation level by default, but the probability that a transaction will not be able to commit is very high, and such DBMSs provide automatic attempts to commit on the server side and the client side. With a large number of parallel transactions, this can lead to a decrease in performance. Because of this, such DBMSs cannot be considered universal, they have their own niche and sequence of operations in transactions, in which there will be no problems with transaction committing, and therefore performance.

Transaction Statuses (CLOG)

- The log is a bitmap with two bits allocated for each transaction.
- Bit values:
 - > 00 - transaction in progress
 - > 01 - fixed
 - > 10 - aborted
 - > 11 - subtransaction committed but parent transaction not completed
- The CLOG is accessed by processes, including vacuum, which performs row version freezing, to find out the status of transactions.
- The maximum size of CLOG files depends on the `autovacuum_freeze_max_age` configuration parameter.

ca	
0	0
1	0
0	1



Transaction Statuses (CLOG)

The transaction status log (CLOG) stores the states of past transactions up to the value of the `autovacuum_freeze_max_age` configuration parameter from the current transaction. The log is a bitmap, with two bits allocated for each transaction. The array is stored in files in the `PGDATA/pg_xact` directory. The files are copied entirely to the WAL at the beginning of each checkpoint. The files are accessed using a shared memory buffer called transaction (formerly called CLOG Buffers):

```
postgres=# SELECT name, allocated_size, pg_size_pretty(allocated_size) from
pg_shmem_allocations where name like '%tran%';
```

name	allocated_size	pg_size_pretty
subtransaction	267520	261 kB
transaction	529664	517 kB

The buffer size is set by the `transaction_buffers` configuration parameter.

Memory usage statistics:

```
postgres=# select name, blks_zeroed, blks_hit, blks_read, blks_written from
pg_stat_slru where name like '%tran%';
```

name	blks_zeroed	blks_hit	blks_read	blks_written
subtransaction	9888	8	0	9889
transaction	308	24935727	24	457

Bit values: 00 - transaction in progress, 01 - committed, 10 - aborted, 11 - subtransaction committed, but is a subtransaction of another transaction that has not yet completed. A subtransaction is created if a savepoint is explicitly (SAVEPOINT) or implicitly (EXCEPTION block in plpgsql) created in a transaction. Subtransactions have their own numbers and select them from the general transaction counter, which exhausts these numbers faster.

The CLOG is accessed by processes including vacuum, which **freezes** row versions, to find out the status of transactions. **The maximum size of CLOG files depends on the `autovacuum_freeze_max_age` configuration parameter**.

<https://eax.me/postgresql-procarray-clog/>

Committing a transaction

- When a transaction is committed, a record is written to the transaction log (WAL) indicating that the transaction has been committed.
- Writing a bit to the CLOG log buffer
- Resources that were used during the transaction are released: locks, cursors (except WITH HOLD cursors), contexts (parts) of the local process memory
- CLOG files are stored in WAL at the beginning of a checkpoint and changes to them are not logged until the next checkpoint. During crash recovery, the CLOG contents are reconstructed from WAL records.



Committing a transaction

When a transaction commits, a record is written to the transaction log (WAL) that the transaction has committed. This is done to ensure fault tolerance. A bit is written to the CLOG log buffer. A bit is set in the CLOG for the transaction that is committing, indicating a successful commit. This allows us to determine which transactions have completed successfully.

Resources that were used during the transaction are released: locks, cursors (except WITH HOLD cursors), contexts (parts) of the local memory of the process.

In case of transaction cancellation (ROLLBACK), instead of committing, information about transaction cancellation is written to CLOG and journal.

CLOG files are stored in WAL at the beginning of a checkpoint, and changes to them are not logged until the next checkpoint. During crash recovery, the CLOG contents are reconstructed from WAL records.

Rolling back and committing a transaction occurs equally quickly.

Subtransactions

- subtransactions are save points
 - › are used to roll back, not to put the transaction into a failed state
- are being created
 - › SAVEPOINT team
 - › EXCEPTION section in pl/pgsql block
 - › in psql when opening a transaction when setting the parameter `\set ON_ERROR_ROLLBACK interactive`
- 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



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.

There is a parameter in psql:

```
postgres=# \set ON_ERROR_ROLLBACK interactive
```

disabled by default. When using the interactive value, when working interactively in psql, psql will set a savepoint before each command in an open transaction. Due to this, in case of any error (for example, a typo in the command), the last command will be rolled back. This makes working in psql more convenient. Setting the value to 'on' is not worth it, since when executing scripts (non-interactively), if transactions are opened in them or the autocommit mode is disabled, savepoints will be set. This will significantly slow down the execution of commands and will unnecessarily consume transaction numbers.

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 modify) and shared (read) modes. There is no deadlock detection, they are automatically released in case of failure. The overhead of acquiring and releasing a lock is small - several dozen processor instructions, if there is no conflict for the lock. Waiting for a lock does not load the processor. Processes acquire a lock in the order of the queue. There are no timeouts for acquiring lightweight locks. Spinlocks are used when accessing LWLock structures. The number of LWLocks is limited by the constant: `MAX_SIMUL_LWLOCKS=200` . There are more than 73 named LWLocks, sets (tranches) of which protect access to structures in shared memory. Their names are present in wait events. Example 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), 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.

Object locks

- When executing commands, a lock is requested on the objects affected by the command.
 - SELECT automatically requests ACCESS SHARE locks on tables, indexes, views
- Until the locks are acquired, the command will not start executing.
- Object-level locks use a "fair" lock queue. This means that locks will be serviced in the order they are requested, regardless of the levels of the locks requested, and there is no priority.
- The lock_timeout configuration parameter can be used to set the maximum time to wait for acquiring a lock on objects or rows.



Object locks

When executing commands, a lock is requested on the objects affected by the command. For example, SELECT automatically requests ACCESS SHARE locks on tables, indexes, and views used in the query to form an execution plan. Until the locks are obtained, the command will not begin to execute.

Object-level locks use a "fair" lock queue, meaning that locks will be serviced in the order they are requested, regardless of the levels of the locks requested, and there is no priority.

The lock_timeout configuration parameter can be used to set the maximum waiting time for obtaining a lock on any object or table row. If the value is specified without units, milliseconds are used. The timeout applies to each attempt to obtain a lock. When working with table rows, there may be many attempts even when executing a single command.

Compatibility of 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



Compatibility of 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 - **installs 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 path.

Strong locks, if present, do not allow weak locks to be installed via 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:

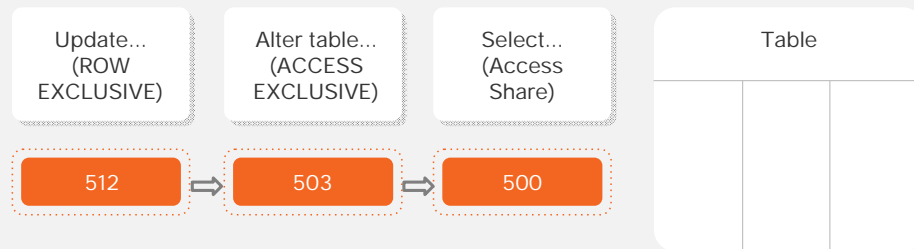
```
ERROR: canceling autovacuum task
```

```
DETAIL: automatic vacuum of table 'name'
```

Autovacuum will try to process the table and its indexes again in the next cycle.

Object locks

- Object-level locks form queues



Object locks

For example, there is a table. It is accessed in transaction 500 to select SELECT data. An Access Share lock is imposed. In parallel, after some time, the Alter table (ACCESS EXCLUSIVE) command comes from transaction 503. The transaction is queued. If another transaction comes that is not compatible in terms of lock level, for example, with number 512 Update (ROW EXCLUSIVE), it will also be queued. To perform their actions, transactions will wait until the previous one is completed.

The level of locks can be compatible. For example, if an update of rows comes with an update of rows of the same table but other, these transactions can do their work in parallel.

Row locks

- Compatibility table

requested mode	line locked in mode			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X



Row locks

Row level locks are set automatically.

A transaction can hold conflicting locks on the same row, but beyond that, two transactions can never hold conflicting locks on the same row. Row-level locks do not affect data queries; they only block writers and blockers for the same row.

Row-level locks are released when the transaction commits or when the savepoint is rolled back, just like table-level locks. Locking modes:

FOR UPDATE: Requests a row lock for update operations, preventing them from being modified or locked by other transactions until the current transaction completes. Used when performing UPDATE, DELETE, SELECT FOR UPDATE, and similar operations.

FOR NO KEY UPDATE: Similar to FOR UPDATE, but weaker locking, does not affect SELECT FOR KEY SHARE commands.

FOR SHARE: Requests a shared row lock for reading, preventing modification or locking by other transactions for UPDATE, DELETE, SELECT FOR UPDATE, and similar operations.

FOR KEY SHARE: Similar to FOR SHARE, but blocks SELECT FOR UPDATE and does not affect SELECT FOR NO KEY UPDATE.

PostgreSQL does not store information about modified rows in memory, and there is no limit on the number of rows that can be locked at one time.

https://docs.tantorlabs.ru/tdb/en/17_5/se/explicit-locking.html

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 key column values
- 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



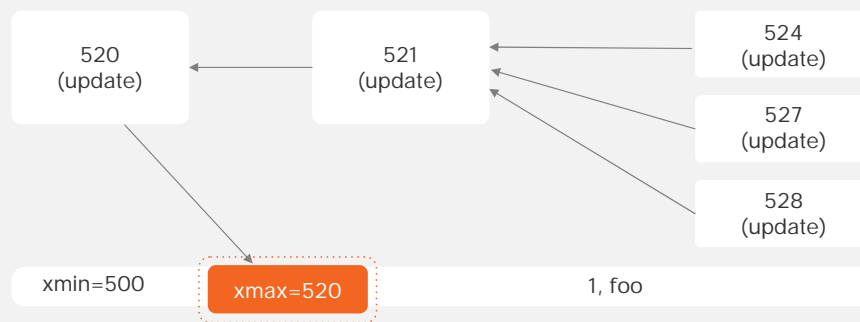
Multitransactions

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 application architect (designer) 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 the multitransaction, that is, there can be two or more transactions. Moreover, a new multitransaction is created, which includes the previous transactions. This is not optimal, but the probability that not two, but three or more transactions will want to change a 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.

Queue when row is locked

- there is the first in line and the rest
- the first in line can be overtaken by a transaction whose row lock level is compatible with the level of the transaction that already locked the row
- if the first in the queue gets a lock, then a transaction is randomly selected to take its place, regardless of the waiting time



Queue when row is locked

A row lock is indicated by the filling of the xmax field in the row version header.

If a transaction arrives that is not compatible with the lock level, it will be queued, trying to grab transaction number 520.

The remaining transactions are queued behind transaction 521.

If transaction 520 is released, transaction 521 acquires a new version of the row, and the next transaction is an arbitrary transaction from the "heap" of waiting transactions. We can say that there is the first in line to acquire the lock and all the rest.

The first in line can be overtaken by a transaction whose row lock level is compatible with the level of the transaction that already locked the row.

Practice

1. Insert, update and delete a row
2. Row version visibility at different isolation levels
3. Transaction status by CLOG
4. Table Lock
5. Line lock



Practice

Insert, update and delete a row

Row version visibility at different isolation levels

Transaction status by CLOG

Table Lock

Line lock



2c PostgreSQL Architecture

Routine maintenance tasks



Autovacuum

- is performed by the working processes of the autovacuum
- tables are selected in which more than 20% of the table size was updated or inserted
- if there has not been a line version freeze for a long time, then a freeze is performed
- after vacuuming the table, it performs an autoanalysis
- Autovacuum requests a SHARE UPDATE EXCLUSIVE level lock and if it cannot obtain a lock on a table, then that table is not vacuumed in this autovacuum cycle.
- autovacuum does not process temporary tables



Autovacuum

Routine Vacuuming is performed by autovacuum workers. Autovacuum selects tables in which `autovacuum_vacuum_scale_factor` has changed from the table size or `autovacuum_vacuum_insert_scale_factor` has been inserted from the table size. The default values are set to 20%. During the vacuuming process:

- 1) versions of table rows that have gone beyond the database horizon are cleared. Blocks that contain only current row versions are skipped
- 2) records in index blocks that point to row versions being cleared are cleared
- 3) a visibility map file is created or updated
- 4) free space map files are created or updated
- 5) TOAST table and TOAST index row versions are cleared

If a table's row version freeze was performed more than `autovacuum_freeze_max_age` transactions ago. The xid of the last freeze is stored for tables in the `relfrozenxid` and `relminmxid` columns of the `pg_class` system catalog table. The default value is 200 million if the transaction counter is 32-bit (PostgreSQL and Tantor Postgres BE) or 10 billion if 64-bit (in Tantor Postgres SE). Periodic freezing is necessary to prevent the issuance of new transaction numbers from stopping and stopping service until the freeze is performed.

Autovacuum requests a SHARE UPDATE EXCLUSIVE level lock and if it cannot obtain a lock on a table, then that table is not vacuumed in this autovacuum cycle.

After vacuuming the table, performs autoanalysis if more than `autovacuum_analyze_scale_factor` (default 10%) of the table rows have changed.

Autovacuum does not process temporary tables. Autovacuum does not work on physical replicas, since the changes made by Autovacuum on the master are transmitted via the log.

https://docs.tantorlabs.ru/tdb/en/17_5/se/routine-vacuuming.html

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
- The phase column reflects the current vacuum phase: initializing (preparatory, happens 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 the 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 is tracked** via `pg_stat_progress_cluster`
- ANALYZE is tracked via `pg_stat_progress_analyze`



Представление `pg_stat_progress_vacuum`

`pg_stat_progress_vacuum` view 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, since it creates data files with rows in sorted order.

The ANALYZE command is tracked through the `pg_stat_progress_analyze` view.

The phase column reflects the current vacuum phase: initializing (preparatory, happens quickly) , scanning heap, vacuuming indexes, vacuuming heap, cleaning up indexes, truncating heap, performing final cleanup .

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 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** .

At the same time, you can use the `pg_stat_activity` view, which also displays the activities 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.

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 quickly remove dead rows from table blocks.
- `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 the `pg_database.datfrozenxid` number - the oldest unfrozen XID in database objects. To get the value, a query is performed `relfrozenxid`, `relminmxid` from `pg_class` using a full scan (there is no index on these columns and it is not needed) . If the size of `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 clean anything, but will only update the value of `pg_database.datfrozenxid` .

`VERBOSE` - displays command execution statistics. Does not add additional load, recommended to 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 the `vacuum_buffer_usage_limit` configuration parameter (range from 128K to 16M, default 256M). 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.

default_statistics_target parameter

- 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`) of a 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;
```



default_statistics_target parameter

To collect statistics, a random sample of rows equal to `300*default_statistics_target` is used. 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 there are many rows in the table, the distribution of values is uneven, then 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 automatic analysis 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. The range of values is: 0..10000; The value -1 returns to the application 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.

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
- extension for row version count estimation:

```
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 a lock incompatible with autovacuum was set on the table at the time of access. 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 comparing the result with the estimate.

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;
```

Оценивать можно по столбцам `dead_tuple_percent` для таблиц и `avg_leaf_density` для

индексов:

relname	t
table_len	8192
scanned_percent	100
approx_tuple_count	1
approx_tuple_len	32
approx_tuple_percent	0.390625
dead_tuple_count	0
dead_tuple_len	0
dead_tuple_percent	0
approx_free_space	8112
approx_free_percent	99.0234375

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 records point to the ctid of the previous version of the row. The 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 executed 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;
```

prevents HOT from being executed if columns c1 and c2 are mentioned in the UPDATE command.

From the index, the server process gets to the old version of the row, sees the `HEAP_HOT_UPDATED` bit, moves along the `t_ctid` field to the new version of the row (taking into account the visibility rules, if it sees 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 row. Such versions of the row 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 row, on which the `HEAP_ONLY_TUPLE` bit is set, and stop on it.

If the new row version is located in a different block than the old row version, HOT does not apply. The `t_ctid` field of the old version will point 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 in the chain of HOT versions. New entries will be created in all indexes on the table pointing to the new row version.

HOT update monitoring

- HOT statistics are available in the `pg_stat_all_tables` and `pg_stat_user_tables` views.
- The 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 update monitoring

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. **Resetting the counters resets 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 reset ("reset") by calling the function:

```
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');
```

Starting with version 17, `pg_stat_reset_shared(null)` resets all these caches; in version 16, it does nothing.

How to perform monitoring? For example, if you created an additional index or increased the number of sections of a partitioned table, you should check how the percentage of HOT updates has changed. `n_tup_hot_upd` is the HOT update counter, `n_tup_upd` is 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;
```

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
- 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



Влияние FILLFACTOR на HOT 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 searches for a place in the block and a chain of versions is created. If the inserted new version of the row fits into the block and the fill percentage exceeds the `min(90%, FILLFACTOR)` boundary, then a flag will be set in the block header that the block can be cleaned. The next update of the block row will perform HOT cleanup - it will clean the block from rows in the chain of versions that have gone beyond the horizon of the base 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 clear is not performed, the row version is inserted into another block, the HOT chain is broken, 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 at which the new row version fits in the block and at the same time 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, the server process can remove **dead tuples** (row versions that have passed 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 space that will be used by new row versions resulting from an UPDATE
- is fulfilled if:
 - › the block is more than 90% full or FILLFACTOR (default 100%)
 - › a 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 a block. This is called in-page cleanup.

HOT cleanup/pruning is performed if one of the following conditions is met:

the block is more than 90% full or FILLFACTOR (default 100%).

the `PD_PAGE_FULL` hint in the block header).

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.

In fact, 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 because they may be referenced by indexes, which the server process cannot check. Only a vacuum can **free the pointers (make the pointers unused)** so that the pointer can be used again. In the version data area, the dead tuples are cleared and the remaining rows are shifted.

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 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

If during Index Scan the server process detects that a row (or a chain of rows referenced by the index record) has been deleted and has gone beyond the database horizon, the LP_DEAD (known dead, killed tuple) **hint bit is set in the lp_flags of the leaf page index record. The bit can be viewed in the** `dead` column returned by the `bt_page_items('t_idx', block)` function. It is not set during Bitmap Index Scan and Seq Scan. A row marked with such a flag will be deleted later when executing a command that makes changes to the index block. Why is index space not freed immediately? Index scans are performed by 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. Other changes to the block require an exclusive lock on the block and another lock on the object itself. 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, LP_DEAD ("ignore_killed_tuples") set on the master is ignored on replicas. 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 records 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 being 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 multiple index and table blocks were read

Execution Time: 0.373 ms

Evolution of indexes: creation, deletion, rebuilding

- `create/drop/reindex index index_name` commands place a SHARE lock that is incompatible with 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` or `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 `SELECT`, `WITH`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` commands to be executed and allows using the fastpath for locking 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;
```

sets a SHARE lock that is 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, rebuild several indexes on one table**, since the SHARE lock is compatible with itself. `CONCURRENTLY` is not compatible with SHARE.

"Not compatible" means that either the command will wait, or will return an error immediately, or will return an error after a 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 a SHARE UPDATE EXCLUSIVE lock, which allows `SELECT`, `WITH`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` commands to be executed and enables the use of the fastpath for locking objects by processes.

The SHARE UPDATE EXCLUSIVE lock is also set by the commands `DROP INDEX CONCURRENTLY`, `REINDEX CONCURRENTLY`, as well as `VACUUM (without FULL)`, `ANALYZE`, `CREATE STATISTICS`, `COMMENT ON`, some types of `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, and 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.

Partial indexes

- 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 indexes

Partial indexes are created on a portion of the table rows. The portion of the rows is determined by the WHERE predicate, which is specified when creating the index and makes the index partial.

The index size can be significantly reduced and vacuuming will be faster, since vacuuming scans all index blocks. Partial indexes can be created. This is useful if the application does not work with unindexed rows. When creating an index, a **WHERE condition can be specified**. The index size 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/en/17_5/se/indexes-partial.html

REINDEX Team

- rebuilds indexes
- REINDEX blocks almost all queries except prepared queries whose plan was cached and which do not use a rebuildable index.
- To rebuild one index:
- REINDEX INDEX index_name;
- If you need to rebuild all indexes on a table:
- REINDEX TABLE table_name;
- You can also rebuild indexes within a specific schema or even the entire database:
- REINDEX SCHEMA schema_name;
- REINDEX DATABASE; you can rebuild indexes on tables of the current database only, except for indexes on tables of the system catalog
- REINDEX SYSTEM; rebuilding indexes on system catalog tables



REINDEX Team

The REINDEX command rebuilds indexes. REINDEX is similar to dropping and recreating an index, because the index contents are rebuilt from scratch. However, locking is handled differently. REINDEX blocks writes, but not reads, of the index's parent table. It also takes an ACCESS EXCLUSIVE lock on the index it is processing, which blocks reads that attempt to use that index. In particular, the query planner tries to take an ACCESS SHARE lock on every index on the table, regardless of the query, so REINDEX blocks almost all queries except for some prepared queries whose plan was cached and that do not use that same index.

To rebuild one index:

REINDEX INDEX index_name;

If you need to rebuild all indexes on a table:

REINDEX TABLE table_name;

You can also rebuild indexes within a specific schema or even the entire database:

REINDEX SCHEMA schema_name;

REINDEX DATABASE; you can rebuild indexes on tables of the current database only, except for indexes on tables of the system catalog

REINDEX SYSTEM; rebuilding indexes on system catalog tables

When rebuilding, you can move indexes to another tablespace; to do this, simply specify the option:

REINDEX (TABLESPACE name) ..;

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-reindex.html

REINDEX CONCURRENTLY

- non-blocking index rebuild
- for each index being rebuilt, a first pass is performed, in which the index is built
- a second pass is performed, in which the records added to the table during the first pass are added to the index
- integrity constraints that used rebuildable indexes are switched to define a new index and the index names are changed
- old index structures are deleted
- the locks are removed
- The rebuild may fail, in which case REINDEX CONCURRENTLY aborts but leaves behind a broken new index in addition to the one being rebuilt. This index will be ignored by queries but will be updated when data changes, which will incur overhead.



REINDEX CONCURRENTLY

Rebuilds an index, locking it with a SHARE UPDATE EXCLUSIVE lock on the index, which is compatible with commands that change rows in a table. The command is executed as follows:

- 1) An index definition is added to pg_index that will replace the index being rebuilt. To prevent any schema changes during the operation, the indexes being rebuilt, as well as their associated tables, are protected by a session-level SHARE UPDATE EXCLUSIVE lock.
- 2) For each index being rebuilt, the first pass is performed, in which the index is built. When the index is built, its pg_index.indisready flag is set to true so that the index is ready for additions, and thus becomes visible to other transactions that started after the index rebuild. This action is performed in a separate transaction for each index. Transactions that started before the index rebuild is complete do not see or use the new indexes.
- 3) A second pass is performed, in which the records added to the table during the first pass are added to the index. This action is also performed in a separate transaction for each index.
- 4) Integrity constraints that used the indexes being rebuilt are switched to defining a new index and the index names are changed. At this point, the pg_index.indisvalid flag of the new index is set to true, and the old index is set to false, and the system catalog caches are flushed, and all sessions that accessed the old index will work with the new index structure. The pg_index.indisready flag of the old index is reset to false to prevent new records from being added to it as soon as the current queries that could access this index are completed.
- 5) Old index structures are dropped. Session-level SHARE UPDATE EXCLUSIVE locks on indexes and tables are released.

The rebuild may fail, in which case REINDEX CONCURRENTLY aborts but leaves behind a non-working new index in addition to the one being rebuilt. This index will be ignored by queries but will be updated when data changes, which will increase overhead. The `psql \d` command marks such indexes as INVALID.

HypoPG expansion

- installed by the `CREATE EXTENSION hypopg` command;
- used when configuring the execution of SQL commands
- allows you to create a definition of indexes that exist only in the current session and do not affect the operation of other sessions
- allows you to find out whether the scheduler will use the index when executing specific commands without creating the index
- The extension allows you to hide any existing indexes in the current session so that they do not affect the scheduler
- hypothetical indexes are created by the function `hypopg_create_index('CREATE INDEX...')` , which is passed the text of the index creation command



HypoPG expansion

When configuring query execution, a question may arise: if you create an index with the desired parameters, will this index be used by the planner to execute the queries that are optimized? You don't want to create a real index, because it can affect the operation of application sessions - slow down commands that change data; creating an index takes a lot of time. The extension allows you to create a definition of indexes that exist only in the current session and do not affect the operation of other sessions. This definition (hypothetical index) is taken into account when creating an execution plan in the session where it is created as existing. When executing a command and during EXPLAIN (analyze), such an index is not used. Also, in the current session, you can hide any indexes from the planner, including existing ones, and see how this affects the generated command execution plans.

The extension has two views where you can see which indexes are hidden in the current session and which hypothetical indexes exist: `hypopg_hidden_indexes`, `hypopg_list_indexes` .

Working with indexes is performed using eleven functions included in the extension. Hypothetical indexes are created by the `hypopg_create_index('CREATE INDEX...')` function , which is passed the text of the index creation command. Hiding any index, including a regular index, from the scheduler in the current session is performed by calling the function:

```
hypopg_hide_index('index_name'::regclass);
```

The execution plan is viewed using the EXPLAIN command.

https://docs.tantorlabs.ru/tdb/en/17_5/se/hypopg.html

Transaction counter

- For a 32-bit transaction counter (xid), the maximum value is 4 billion.
- when this limit is reached, the transaction counter rolls over "zero" and transaction numbering starts from 3
- To prevent the row version counter from overflowing, row versions are "frozen", which means that the row version is the only one that is current and visible in all snapshots.



Transaction counter

Transaction (xid) and multitransaction (mxid) counters are used to track the order of transactions and determine which row versions can be seen by each transaction. In PostgreSQL, the transaction counter is implemented as a 32-bit value. To prevent the counter from overflowing, row versions are "frozen", meaning that a single, current row version is visible across all snapshots.

For a 32-bit transaction counter (XID) in PostgreSQL, the maximum value is 4 billion. When this limit is reached, the transaction counter rolls over "zero" and transaction numbering starts from 3. Values 0, 1, 2 are not used for normal transactions. For example, xid=2 is a sign of a frozen row. xid=0 in the xmax field means that the row version was not deleted.

The numbers of the oldest unfrozen transactions are stored in the pg_database in the datfrozenxid and datminmxid columns. If the current transaction number is slightly less than 2 billion away from the values, new transaction numbers will no longer be issued to server processes. The values can be updated by vacuuming with freezing the tables. The values are determined by one of the tables that has not been frozen for the longest time. By vacuuming this table, the values will be set to the next oldest table that has not been frozen for a long time.

Tantor Postgres SE uses a 64-bit transaction counter. There are no problems with counter overflow on 64-bit transaction counters. The only thing is that if a query or transaction runs for a long time, and during this time 2 billion transactions pass, then such transactions and queries should be interrupted.

In PostgreSQL, the 64-bit counter is not enabled because the changes required to the PostgreSQL code to switch to a 64-bit transaction counter are too large.

Practice

1. Normal table cleaning
2. Table Analysis
3. Rebuilding the index
4. Complete cleaning
5. HypoPG expansion



Practice

Normal table cleaning
Table Analysis
Rebuilding the index
Complete cleaning
HypoPG expansion



2d PostgreSQL Architecture

Executing queries



SQL is a declarative language

- SQL is a declarative programming language for working with databases, where the user describes the desired results of a query without specifying specific execution steps.
- When a user submits an SQL query to the PostgreSQL database management system (DBMS), the following happens:
 - Parsing (syntactic and semantic analysis) of a request
 - Rewriting
 - Planning
 - Execution



SQL is a declarative language

SQL (Structured Query Language) is a declarative programming language, meaning you need to describe what you want to achieve rather than specifying how to do it step by step. Unlike imperative programming languages, where a program provides a sequence of commands that are executed according to an algorithm, declarative languages like SQL focus on what needs to be achieved, leaving the optimization and implementation details to the DBMS.

In SQL, you formulate queries by specifying what data you want to retrieve or what operations you want to perform, but you do not say how the system should do it. SQL is a more abstract and data-friendly language, allowing the system to optimize query execution and hide the details of data storage.

When a user submits a query to the DBMS, the following happens:

Parsing: analyzes the user's request, checks its syntax, and performs semantic analysis to understand the meaning of the request. Consists of syntactic and semantic analysis.

Rewrite (transformation): the query structure is transformed into an equivalent one, more convenient for the following steps

Planning: The optimizer creates an optimal query execution plan by deciding which indexes to use, how to join tables, and in what order to perform operations.

Execution: The query is executed according to the selected plan. This step involves reading rows from the data blocks, processing the rows, and returning the result.

Note: "query" is a command (statement) like SELECT, INSERT, UPDATE, DELETE, MERGE, VALUES, EXECUTE, DECLARE, CREATE TABLE AS, CREATE MATERIALIZED VIEW AS . Query does not mean "request data" (select, choose data), but "request the execution of actions on data processing" . Commands like create, alter, drop are not called queries, because they are not planned (executed in one way), change object definitions (metadata), and not application data.

Syntax parsing

- Syntactic parsing includes the following steps:
 - › lexical analysis
 - › constructing a syntax tree
 - › grammar check

```
SELECT col1, col2 FROM table1 WHERE col2='test1';
```

the query test is syntactically correct



Syntax parsing

Parsing is the analysis of an input sequence of characters (tokens) to determine the structure of words according to the rules of the language's grammar. In the context of programming languages or SQL queries, parsing is used to check whether the input text complies with the correct syntax of the language.

Steps:

- 1) Lexical analysis (tokenization): The input string is broken down into a set of tokens representing minimal syntactic units such as keywords, operators, identifiers, and numbers.
- 2) Syntax tree construction: Tokens are combined into a data structure called a syntax tree, which reflects the hierarchy and structure of the language. This tree is an abstract syntactic representation of the input expression.
- 3) Check for grammar compliance: The parser checks whether the constructed syntax tree complies with the grammar rules of the language. If not, an error is generated indicating that the syntax is incorrect.

In the case of an SQL query, checks that the query complies with SQL syntax rules, allowing the query to be represented (interpreted) and executed.

Semantic analysis

- definition of meaning (semantics), checking the existence of tables, columns, consistency of data types
- checking access rights: does the user have the right to execute the command, access rights to the objects specified in the request: schemas, tables, functions, views, etc.
 - there is an access to the **system catalog tables** , which store object definitions. For example, `pg_class`, `pg_attribute`, `pg_type`, `pg_depend`, `pg_constraint`, `pg_namespace`, `pg_inherits`, `pg_attrdef`, `pg_sequence`
 - The selected data is cached in the local memory of the process servicing the user session.



Semantic analysis

Determining the meaning (semantics): This stage of SQL parsing includes analyzing the meaning of the query, checking the existence of tables, columns, and consistency of data types.

Checking access rights: does the user have the right to execute the command, access rights to the objects specified in the request: schemas, tables, functions, views, etc.

This step accesses the system catalog tables that store object definitions. For example, `pg_class`, `pg_attribute`, `pg_type`, `pg_depend`, `pg_constraint`, `pg_namespace`, `pg_inherits`, `pg_attrdef`, `pg_sequence` . The retrieved data is cached in the local memory of the process in the memory structure (called "contexts") `CacheMemoryContext` , which services the user session. In the future, if changes are made to the rows of the system catalog tables, the process that makes the changes transfers the changes to a circular (new messages overwrite old ones in a circle) buffer (`shmInvalBuffer`) in shared memory with a size of 4096 messages:

```
postgres=# select name, size from (select name, lead(off) over(order by off) -  
off as size from pg_shmem_allocations) as a where name='shmInvalBuffer';
```

```
name          | size  
-----+-----  
shmInvalBuffer | 291072  
(1 row)
```

If a process has not consumed half of the messages, it is notified to consume the accumulated messages. This reduces the likelihood that a 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 (for example, it is performing an operation and cannot be interrupted), and the buffer is full, the process will have to completely clear its system directory cache.

Locks are set on all objects that are used in the query and can be used to create a plan: tables, indexes, table sections. Locks are needed so that while the query is being planned or executed, the objects used are not deleted or their structure is not changed, which would lead to an error when creating a plan or executing the query.

Transformation (rewriting) of a query

- transformation of the original query structure into a similar one in terms of obtaining the result for the purpose of better optimization at the planning and execution stages for the purpose of better optimization at the planning and execution stages
 - › view names, if any, are replaced by the queries on which the views are based

```
LOG: rewritten parse tree:
DETAIL: (
{QUERY
:commandType 1
:querySource 0
:canSetTag true
:utilityStmt <>
:resultRelation 0
:hasAggs false
:hasWindowFuncs false
...
```



Transformation (rewriting) of a query

Query transformation (rewriting) is the transformation of the original query structure into a similar one in terms of obtaining the result for the purpose of better optimization at the planning and execution stages.

For example, view names, if any were in the query, are replaced with the queries on which the views were created.

`debug_print_rewritten` configuration parameter allows you to see the result of rewriting in the diagnostic log. Example:

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*
```

```
STATEMENT:  select * from t limit 1;
```

```
LOG:  rewritten parse tree:
```

```
DETAIL:  (
  {QUERY
    :commandType 1
    :querySource 0
    :canSetTag true
    :utilityStmt <>
    :resultRelation 0
    :hasAggs false
    :hasWindowFuncs false
    :hasTargetSRFs false
    :hasSubLinks false
    :hasDistinctOn false
    :hasRecursive false
    :hasModifyingCTE false
    :hasForUpdate false
    :hasRowSecurity false
  :isReturn false
  :cteList <>
  :rtable (
    {RANGELEFTENTRY
      :alias <>
      :eref
      {ALIAS
        :aliasname now
        :colnames("now")
      }
      :rtekind 3
    ...
```

Execution planning (optimization)

- The goal of this step is to get the best way (plan) to execute the query.
- Possible ways of executing the query are generated, the complexity of execution is estimated and the execution method (plan) with the lowest cost is selected
- The following is used to estimate the cost:
 - statistics describing objects
 - weighting factors from configuration parameters
 - Configuration options that enable the use of string fetching and processing methods
- The cost calculation includes two parts: computational complexity (processor) and input/output.



Query execution planning (optimization)

This is the process of finding the best way to fulfill a request.

The scheduler (optimizer) is the code (written in C) of the server process that executes the query. The logic of the code is algorithmic. Possible ways to execute the query are generated, the complexity of execution is estimated, and the method (plan) of execution with the lowest cost is selected. Statistics describing objects are used to estimate the cost. For example, the number of rows and blocks in tables, indexes, the number of unique values in columns, the number of several most frequently occurring values, etc. The optimizer code contains weighting factors for calculating the cost. Some of the factors are specified in the configuration parameters so that they can be customized. For example, `seq_page_cost`, `random_page_cost`, `parallel_setup_cost`, `parallel_tuple_cost`, `cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost`. Configuration parameters that can incline the optimizer to select methods for sampling and processing data are also taken into account. The names of most of these parameters begin with `enable` and there are several dozen of them. For example: `enable_seqscan` (ability to scan all table blocks to select rows from them); `enable_nestloop` (ability to join sets of rows using nested loops).

The cost calculation includes two parts: computational complexity (processor) and input/output.

The query execution plan can be viewed using the `EXPLAIN query command`;

Some of the locks on objects that were not used in the created plan are removed.

Executing a request

- Reading data: rows are read from blocks of tables, indexes, functions
- Data processing: filtering, sorting, grouping, calculations
- Joining Rowsets: If the query involves joining tables or other data sources
- Grouping rows: for example, if group functions like COUNT, SUM, AVG are used
- Return Result: Return strings to the client or to the code that sent the execution request
- Resource release: Locks are released from objects and memory used to execute the query is freed



Executing a request

Execution is the final step in request processing, where the actions described in the execution plan are performed.

Reading data: rows are read from blocks of tables, indexes, functions.

Data processing: filtering, sorting, grouping, calculations.

Joining rowsets: If the query involves joining tables or other data sources.

Grouping rows: for example, if group functions like COUNT, SUM, AVG are used, the GROUP BY expression.

Returning a result: The process of returning strings to the client or to the code that sent the request for execution.

Resource release: The process that executed the request releases the resources it used: it releases locks on objects and frees (nominally for reuse or by returning to the operating system) the memory used to execute the request.

EXPLAIN command

- produces a query execution plan
 - > By default, the request is not executed.
- If you specify the **analyze option**, the query will be executed, and after execution, a plan with execution details will be shown.
 - > without the **timing off** option " **Execution Time** " it can give an execution time that exceeds the real one several times
 - > **The buffers** option shows the number of buffers that were read during scheduling and execution.

```
postgres=# explain (analyze, buffers) select * from t limit 1;
          QUERY PLAN
-----
Limit  (cost=0.03..0.04 rows=1 width=8) (actual time=0.048..0.067 rows=1 loops=1)
 Buffers: shared hit=2
-> Seq Scan on t  (cost=0.00..14425.00 rows=1000000 width=8) (actual time=0.015..0.020 rows=1 loops=1)
 Buffers: shared hit=2
Planning Time: 0.040 ms
Execution Time: 0.198 ms
(6 rows)
```



EXPLAIN command

The EXPLAIN command displays the execution plan for the query that is selected as optimal. By default, the query is not executed.

If you specify the **analyze option**, the query will be executed, although the rows will not be issued, and only after the query is executed will a plan with additional details be issued. When using analyze, the actual data will appear in the plan rows after "(actual ". If you do not need the execution time of the plan row "actual time", you can specify the " **timing off**" option, this will allow you to get the actual data in the " **Execution Time** " row, since the calls to the counter can be frequent, and calls also take time.

the buffers option - it will show the number of buffers that were read. The buffers indicator has long been underestimated from an optimization point of view, but its importance has been realized and in PostgreSQL version 18 the buffers parameter is enabled by default.

Example of using the EXPLAIN command:

```
postgres=# explain ( analyze , buffers ) select * from t limit 1;
          QUERY PLAN
-----
Limit  (cost=0.03..0.04 rows=1 width=8) (actual time=0.048..0.067 rows=1 loops=1)
 Buffers: shared hit=2
-> Seq Scan on t  (cost=0.00..14425.00 rows=1000000 width=8) (actual time=0.015..0.020
rows=1 loops=1)
 Buffers: shared hit=2
Planning Time: 0.040 ms
Execution Time: 0.198 ms
```

The query plan allows you to evaluate the methods used to process the data and whether there were any errors in predicting the number of rows (the difference between the planned number of rows and the actual rows - actually read), which are called errors in calculating "cardinality" (a synonym for "power" or even the number of rows, but these terms are less common, since they are less pretentious) and "selectivity" (the proportion of rows) - these terms came to SQL from relational theory.

To obtain data on running queries, the `pg_stat_activity` views and the Tantor Postgres extension **pg_trace** are used . For some commands executed manually or automatically, there are the

`pg_stat_progress_analyze`, `pg_stat_progress_cluster`, `pg_stat_progress_create_index`, `pg_stat_progress_basebackup`, `pg_stat_progress_copy`, `pg_stat_progress_vacuum` views.

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-explain.html

Параметры команды EXPLAIN

- 12 параметров, указываются в круглых скобках

```
postgres=# EXPLAIN (analyze, verbose, buffers, serialize text, settings, memory, wal) SELECT *
FROM t WHERE i = 100 AND j = 10;
               QUERY PLAN
-----
Gather  (cost=1000.00..11676.00 rows=10 width=8) (actual time=0.421..65.852 rows=10 loops=1)
  Output: i, j
  Workers Planned: 2
  Workers Launched: 2
  Buffers: shared hit=4425
  ->Parallel Seq Scan on public.t (cost=0.00..10675.00 rows=4 width=8) (actual
time=37.521..57.814 rows=3 loops=3)
    Output: i, j
    Filter: ((t.i = 100) AND (t.j = 10))
    Rows Removed by Filter: 333330
    Buffers: shared hit=4425
    Worker 0:  actual time=55.948..55.955 rows=0 loops=1
                Buffers: shared hit=1396
    Worker 1:  actual time=56.416..56.422 rows=0 loops=1
                Buffers: shared hit=1399
Query Identifier: 687797574221341570
Planning:
  Memory: used=11kB  allocated=16kB
Planning Time: 0.054 ms
Serialization: time=0.042 ms  output=1kB  format=text
Execution Time: 66.150 ms
```



EXPLAIN command parameters

ANALYZE (default false) execute the query but do not send the result to the client. Allows you to evaluate the actual number of rows, execution time, and use extensions and optimizations, as when executing a query. ANALYZE for INSERT, UPDATE, DELETE commands performs data changes.

VERBOSE (false) prints additional data in the plan. For example, schema names, table aliases, bind variable names, query identifier (Query Identifier) so that its execution statistics collected by extensions (pg_stat_statements) can be found.

COSTS (true) displays the estimated cost of each plan node, rows, and width.

SETTINGS (default false) returns configuration parameters that affect the scheduler, with values different from the default.

GENERIC_PLAN (false) allows you to display a plan for a query that uses bind variables of the form \$1, \$2.. Displays a generic plan that will be used instead of private ones if it is not worse than them. Cannot be used simultaneously with ANALYZE.

BUFFERS (until version 18, false) provides information about buffers read from the cache (hit) plus from the operating system (read) from the shared buffer cache (shared) or the local cache for temporary tables (local). It can return dirty - the number of buffers (already included in read or hit), which were changed by a request for the first time after the checkpoint. written - the number of dirty (including earlier requests) buffers that were sent for writing (evicted) because the server process needed to free the buffer to load another block into the buffer.

SERIALIZE (NONE) includes information about the cost of serializing (allocating memory for a string buffer) the query output (after a SELECT or RETURNING), converting the data to text or binary format for sending to the client. This is relevant if fields are selected from TOAST, since by default, data is not selected from TOAST by the EXPLAIN command. The EXPLAIN command never sends the retrieved data to the client, so the network transmission cost is not considered. Works only with ANALYZE. Values: NONE, SERIALIZE [TEXT], SERIALIZE BINARY.

WAL (false) outputs the number of log records, full page images (fpi), and the size of the generated records in bytes.

TIMING (true) prints the time spent on each node. Can significantly increase the overall query execution time. Used with ANALYZE.

MEMORY (false) memory used during planning stage

SUMMARY (true if ANALYZE) prints Planning Time after the query plan

FORMAT (TEXT) except TEXT can be XML, JSON, YAML

Tables

- объект в котором хранятся данные
- several types: regular tables (heap tables, rows are stored in an unordered manner), unlogged, temporary, partitioned
- Extensions can create new ways to store data and access it
- the number and order of columns are specified 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 them. Tantor Postgres SE has the `pg_columnar` extension (`pgcolumnar`).

The 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.

The fields for the added column have `NULL` values by default or are given 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`;

Deleting a column deletes the values in the fields of each row and the integrity constraints that include the deleted column. If the integrity constraint being deleted is referenced by a `FOREIGN KEY`, you can delete 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` value (if defined) and any integrity constraints that the column is a part of will be converted. It is better to remove integrity constraints before modifying the column type and add the constraints later.

To view the contents of a block, the functions of the standard `pageinspect` extension are used.

https://docs.tantorlabs.ru/tdb/en/17_5/se/ddl-alter.html

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, a btree index is created. btree is the most common index type in relational databases, working with many types of data.

For 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 index, PK and UK constraints 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 btree index 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
```

Methods of accessing data in a query plan

- ways in which a process obtains data (rows, records) from data sources
- sources can be tables, external tables, functions, etc.
- Extensions can create their own access methods
- for the Bitmap method, a bitmap is built by index
 - › the construction of the map is marked by the line **Bitmap Index Scan**
 - › The bitmap is scanned for rows or blocks of the table, which is indicated by **the Bitmap Heap Scan line** in the plan:

```
Bitmap Heap Scan on tab (cost=10..1000.51 rows=998 width=11)
Recheck Cond: (coll < '1000'::numeric)
-> Bitmap Index Scan on t_coll_idx (cost=0.00..9.60 rows=998 width=0)
Index Cond: (coll < '1000'::numeric)
```



Methods of accessing data in a query plan

There are many methods (algorithms) for accessing data: Sequential Scan, Index Scan, Index Only Scan, Bitmap Heap Scan, Bitmap Index Scan, CTE Scan, Custom Scan, Foreign Scan, Function Scan, Subquery Scan, Table Sample Scan, Tid Scan, Values Scan, Work Table Scan and others. When parallelizing, the word Parallel is added before the method name in the plan line. Data sources can be tables, external tables, table functions, etc. Extensions can add their own "methods" (implementation of the algorithm) of access, for example, for a table - in the **Custom Scan method**.

For regular tables, the methods are divided into tabular - Sequential and using indexes - Index, Index Only, Bitmap Heap, Bitmap Index. For the Bitmap method, a bitmap is built. The map is built by the **Bitmap Index Scan line**. Then, using the bitmap, the table rows or blocks are scanned, which is marked by **the Bitmap Heap Scan line** in the plan:

```
Bitmap Heap Scan on tab (cost=10..1000.51 rows=998 width=11)
Recheck Cond: (coll < '1000'::numeric)
-> Bitmap Index Scan on t_coll_idx (cost=0.00..9.60 rows=998 width=0)
Index Cond: (coll < '1000'::numeric)
```

Example of accessing a column-stored table:

```
Custom Scan (ColumnarScan) on public.perf_columnar (cost=0.00..138.24 rows=1
width=8)
```

Possible plan node (row) types are listed in the `src/include/nodes/plannodes.h` file of the PostgreSQL source code.

Only one server process has access to temporary tables, so there is no parallelism when scanning a temporary table.

String Access Methods

- two types of "methods" (ways) of accessing table rows: table 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** :

\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)	

```
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" for accessing table rows: table and index.

List of available access methods: \dA or query:

```
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 add access methods to the pg_am table :

```
2425358 | columnar | columnar.columnar_handler | t
2425512 | bloom | blhandler | i
```

Table access methods define how data is stored in tables. To force 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 (the "composite index") whose values will be indexed. The INCLUDE option allows you to store column values in the index structure; expressions cannot be used. Operator classes are not required for the data types of such columns. 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 is the btree index type.

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createindex.html

Methods for joining sets of rows

- Methods:
 - › Nested Loop Join with and without Memoization
 - › Hash Join with and without temporary files
 - › Merge Join
- Sets of rows are always joined in pairs



Methods for joining sets of rows

Rowsets are always joined in pairs, that is, two sets (selections) are joined. In PostgreSQL, there are three ways to join rowsets (selections) in a query execution plan:

Nested Loop Join : one set of rows is sequentially scanned for each row in the second set. This method is optimal for joining sets with a small number of rows. Its computational complexity is equal to **the product of the number of rows** in the samples. Underestimation of the number of rows during estimation (cardinality estimation errors) leads to a significant increase in the execution time for this join method. The order of the tables does not matter when joining this method. The first row is returned without delay. Can be used with a join condition other than equality.

There is a variation of this method with memoization - caching a set that is scanned many times. When using memoization, this set must be smaller. The Memoize node is embedded in the plan between the node supplying the data and the Nested Loop.

Hash Join - possible only for joining by equality condition. First, the sample with the smallest size is selected, which is determined by the number of rows and the size of the sample row, which consists of the columns mentioned in the query (the table may have more columns). Based on this set, a hash structure (called a hash table) is built in the process memory in one pass, which exists until the query is completed. Then the second sample is viewed and rows are selected from the hash structure if there is a match. The computational complexity is proportional to the sum of the rows in both samples. The first row is returned only after the hash table is built, that is, after reading the first set of rows. If there is not enough memory for the hash table, then temporary files are used and the time to perform the connection increases due to the addition of file operations.

Merge Join : This method requires that both samples be sorted by the columns being joined. In query plans, the sorted rows are a side effect of the lower nodes of the execution plan. For example, when scanning a btree index, the rows arrive at the upper node sorted (by the index key columns). The computational complexity is proportional to the sum of the rows from the two samples. The first row is returned without delay, since a hash table is not built.

All connection methods can be performed in parallel processes.

Cardinality and selectivity

- the cardinal number of a relation (abbreviated cardinality) or the power of a relation is the number of rows (they are also tuples)
 - › in terms of rows and actual rows
 - › Since PostgreSQL version 18, rows are decimal numbers.

```
Gather (actual rows=2.00 loops=1)
-> Parallel Seq Scan on bookings (actual rows=0.67 loops=3)
```

- Selectivity is the proportion (from zero to 1) of rows from the sample

[postgresql.org/docs/18/release-18.html#RELEASE-18-CHANGES](https://www.postgresql.org/docs/18/release-18.html#RELEASE-18-CHANGES)

Modify EXPLAIN to output fractional row counts (Ibrar Ahmed, Ilya Evdokimov, Robert Haas)



Training Course "Tantor: PostgreSQL 17 Administration" Astra Group, "Tantor Labs" © 2025 www.tantorlabs.ru

150

Cardinality and selectivity

Relational theory has only a tangential relationship to practice and a lot of fancy terms that make it difficult to understand. The number of attributes (columns) is called the arity or degree of a relation. Data types are called domains or sets of admissible values. A join of tables is defined as a Cartesian product to which a selection (restriction) operation is applied with a predicate (join condition). The Cartesian product itself has no practical meaning, but it is similar to multiplication, which is why it was defined. There is even division, but it is quite fancy. However, when relational theory appeared, network databases were popular, which are even more confusing. Now relational theory and Codd algebra are of historical interest. Some less fancy terms are still used, such as cardinality and selectivity. Eventually, the SQL language appeared, which is loosely based on relational algebra, and tables in SQL are not exactly relations. For example, you can create a table with identical rows.

In the relational data model, the cardinal number of a relation (abbreviated cardinality) or the power of a relation is the number of rows (aka tuples). In practice, this is the rows indicator in the execution plan nodes. Up to version 18 of PostgreSQL, this is an integer. Starting with version 18 of PostgreSQL, the rows value will be given in decimal form. The authors of the patch are Ibrar Ahmed, Ilya Evdokimov (Tantor Labs), and Robert Haas). Example:

```
Gather (actual rows=2.00 loops=1)
-> Parallel Seq Scan on bookings (actual rows=0.67 loops=3)
```

The reason for introducing decimal values is: $0.67 * 3 = 2.00$, while in previous versions $1 * 3 = 2$ looked like a discrepancy.

```
Gather (actual rows=2 loops=1)
-> Parallel Seq Scan on bookings (actual rows=1 loops=3)
```

is the proportion (from zero to 1) of rows from a sample. For example, if 10% of rows pass through the WHERE condition (called a predicate - a term from relational theory) that filters rows, then the "predicate selectivity" is 0.1. If there is no filtering, then the selectivity of the sample is 1. If zero rows are returned, then the selectivity is zero.

The most common errors of the planner are incorrect assessment of selectivity, which is indicated in the plan by a discrepancy between planned rows and actual rows of more than an order of magnitude.

<https://www.postgresql.org/docs/18/release-18.html#RELEASE-18-CHANGES>

Cost of query plan

- Cost - a numerical estimate of the complexity of executing a plan node or the entire query
- Consists of two numbers with two dots between them
`> cost= 0.00..14425.00`
- The first number (startup cost) is the cost of returning the first row in the sample
- The second number (total cost) is the cost of returning all lines
- The cost value is only meaningful for comparing plans for the same query.
- For all queries except cursors, the plan with the smallest second number is selected
- The cost of the same query correlates with the execution time of that query, but nonlinearly



Cost of query plan

Cost is a numerical estimate of the complexity of executing a plan node or the entire query. It consists of two numbers with two dots between them. The first number (startup cost) is the cost of returning the first row in the selection. The second number (total cost) is the cost of returning all rows. For all queries except cursors, the plan with the smallest second number is selected.

The first number is taken into account when choosing the optimal plan only for cursors, for them the plan with the smallest value is selected: the first number + cursor_tuple_fraction * (the second number - the first number). By default, the value of the configuration parameter is:

```
show cursor_tuple_fraction;
```

```
cursor_tuple_fraction
```

```
-----
```

```
0.1
```

The cost value is meaningful only for comparing plans for the same query. Values for different queries are poorly comparable. The cost of the same query correlates with the execution time of this query, but nonlinearly. When CPU cores or I/O are loaded, the cost does not change, but the execution time of the query increases.

cost calculation :

```
postgres=# EXPLAIN (analyze, buffers) SELECT * FROM t;
```

```
QUERY PLAN
```

```
-----
```

```
Seq Scan on t (cost=0.00..14425.00 rows=1000000 width=8) (actual time=0.016..3924.918  
rows=1000000 loops=1)
```

```
Buffers: shared hit=4425
```

```
Planning Time: 0.033 ms
```

```
Execution Time: 7797.977 ms
```

```
postgres=# select relpages, reltuples::numeric, current_setting('seq_page_cost') seq_page_cost,  
current_setting('cpu_tuple_cost') cpu_tuple_cost, current_setting('seq_page_cost')::float * relpages  
CPU, current_setting('cpu_tuple_cost')::float * reltuples IO,  
current_setting('seq_page_cost')::float * relpages + current_setting('cpu_tuple_cost')::float *  
reltuples total_cost from pg_class c where relname = 't';
```

```
relpages | reltuples | seq_page_cost | cpu_tuple_cost | cpu | io | total_cost
```

```
-----+-----+-----+-----+-----+-----+-----
```

```
4425 | 1000000 | 1 | 0.01 | 4425 | 10000 | 14425
```

In the example, the contribution to the input-output cost is $10000/144.25=70\%$.

Statistics

- used by the planner when choosing a plan
- Basic and advanced statistics are stored in system catalog tables
- collection of extended statistics is set manually by the CREATE STATISTICS command or automatically by the pg_stat_advisor extension
 - › after installation, extended statistics are collected together with the basic autovacuum (autoanalysis) or when executing VACUUM (ANALYZE) commands
- cumulative statistics are stored in shared memory, accessible through pg_stat_all_* views and pg_stat_get* functions
 - › when the instance is stopped, it is saved to a file in the PGDATA/pg_stat/directory
 - › used by autovacuum



Statistics

The scheduler uses statistics. Statistics are collected and stored in system catalog tables for tables and indexes. Basic statistics include information about data distribution, number of unique values, size of tables and indexes, and other metrics. Extended statistics are also collected automatically, but you must define the parameters with the CREATE STATISTICS... command.

Statistics are not updated, they are re-compiled by autovacuum (autoanalysis phase) or the ANALYZE command.

Statistics are stored in the system catalog tables:

`pg_class` and `pg_index`: Contain information about the sizes of tables and indexes, as well as the number of rows in the tables.

`pg_statistic`: Contains statistics about column values such as minimum and maximum values, mean, standard deviation, etc.

Extended statistics are stored in `pg_statistic_ext` and `pg_statistic_ext_data`.

Cumulative statistics are available in the `pg_stat_all_*`, `pg_statio_*` views that output data from instance memory using `pg_stat_get*` functions. When an instance is stopped, cumulative statistics are saved in the `$PGDATA/pg_stat` directory .

pg_statistic table

- The pg_statistic table stores basic statistics.
- Collected by autoanalysis and the ANALYZE command, used for query optimization by the planner
- By default, it is rebuilt by default_statistics_target * 300 = 30000 rows
- contains data for each column of the tables
- Example: the proportion of rows with NULL in the third column of the test table :

```
select stanulfrac from pg_statistic
where starelid = ' test '::regclass and staattnum = 3 ;
stanulfrac
-----
0.9988884
```



pg_statistic table

The pg_statistic table stores basic statistics. It is collected by autoanalysis and the ANALYZE command, and is used to optimize queries by the planner. The statistics are approximate values, even if they are relevant. By default, default_statistics_target * 300 = 30000 rows are collected.

The pg_statistic table contains data for each column of the tables.

For example, the proportion of rows with NULL in the third column of the test table:

```
select stanulfrac from pg_statistic where starelid = 'test '::regclass and
staattnum = 3;
stanulfrac
-----
0.9988884
```

The statistics about the proportion of empty values are used by the scheduler.

More details in the documentation

https://docs.tantorlabs.ru/tdb/en/17_5/se/catalog-pg-statistic.html

Накопительная статистика

- according to 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 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
```

- 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 .



Cumulative statistics

In pg_stat_all_tables - statistics on reading table blocks, all indexes on this table, 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 .

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
```

Statistics n_tup_hot_upd is not updated by vacuum.

The pg_stat_xact_all_tables view 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
```

pg_stat_statements extension

- detailed statistics of the instance operation down to 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
```

- 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 combine into `select * from t where id = $1`



pg_stat_statements extension

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, the queries: `select * from t where id = 'a'` and `select * from t where id = 'b'` are combined into the 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` .

Statistics are reset by calling the `pg_stat_statements_reset()` function .

Tantor Postgres SE 17.5 adds the `pg_stat_statements.sample_rate` configuration parameter , which addresses the performance degradation issue when using the extension on heavily loaded clusters.

Tantor Postgres SE 17.5 also added the `pg_stat_statements.mask_const_arrays`, `pg_stat_statements.mask_temp_tables` configuration parameters , which mask array names and temporary table names, allowing for more accurate grouping of statistics by similar queries.

pg_stat_statements 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.
- `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		



pg_stat_statements configuration parameters

```
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 are tracked. Setting this to `on` can result in noticeable performance degradation, especially when multiple sessions are running commands with the same query structure at the same time, causing those sessions to attempt to modify the same rows in `pg_stat_statements` at the same time . The default is `off` .

`pg_stat_statements.track_utility` determines whether the extension will track utility commands. Utility commands are commands other than `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` . The default value is `on` .

Practice

1. Creating objects for queries
2. Extract data sequentially
3. Returning data by index
4. Low selectivity
5. Using statistics
6. pg_stat_statements view



Practice

Creating objects for queries
Extract data sequentially
Returning data by index
Low selectivity
Using statistics
pg_stat_statements view



2e PostgreSQL Architecture

Extensions



PostgreSQL Extensibility

- adding functionality without making changes to source code files and recompiling
- you can create data types, operators, group functions, type casts
- creating stored routines in a wide range of programming languages
- creating and using extensions
 - › a collection of any database objects that can be installed or removed as a single unit
- extending functionality with shared libraries
- External Data Wrappers (FDW)



PostgreSQL Extensibility

Extensibility of PostgreSQL is the ability to be easily adapted to the needs of applications, administrators, and users. Historically, PostgreSQL was developed with an emphasis on extensibility. In early versions of PostgreSQL, when it was still called Postgres, the creator of the system, Michael Stonebraker, focused on extensibility - adding functionality without having to change the source code files written in C. Non-extensible and closed products usually disappear, leaving only products whose functionality can be easily extended by third-party companies.

You can create data types, operators, group functions, type casts.

Install programming languages for writing stored routines.

Extensions are a set of any database objects that can be installed or removed as a single unit.

It is possible to extend the functionality with shared libraries (.so files)

With the help of extensions (the CREATE EXTENSION command) you can install (Foreign Data Wrapper , FDW, external data wrapper). FDW provides the ability to work with data located in external systems (databases, services, files, etc.) using external tables (foreign table) according to logic. FDW is described in the SQL standard as a way to work with external data.

Extension and library file directories

- Library files are located in the directory:
- `/opt/tantor/db/17/lib/postgresql`
- Extension files (*.control and *.sql) are located in the directory:
- `/opt/tantor/db/17/share/postgresql/extension`
- You can find out the location using the commands:

```
postgres@tantor:/opt/tantor/db/17/lib$ pg_config --libdir
/opt/tantor/db/17/lib
postgres@tantor:/opt/tantor/db/17/lib$ pg_config --sharedir
/opt/tantor/db/17/share/postgresql
```

- Installation using the "PGXS" method:

```
root@tantor:~# export PATH=/opt/tantor/db/17/bin:$PATH
root@tantor:~# export USE_PGXS=1
root@tantor:~# cd extension_directory
root@tantor:~# make
root@tantor:~# make install
```

- usually extensions, libraries, utilities, applications are installed by deb and rpm packages



Extension and library file directories

Library files are located in the directory:

`/opt/tantor/db/17/lib/postgresql`

Extension files (*.control and *.sql) are located in the directory:

`/opt/tantor/db/17/share/postgresql/extension`

You can find out the location using the commands:

```
postgres@tantor:~$ pg_config --libdir
/opt/tantor/db/17/lib
postgres@tantor:~$ pg_config --sharedir
/opt/tantor/db/17/share/postgresql
```

or by request:

```
postgres=# SELECT * FROM pg_config where name in ('LIBDIR','SHAREDIR');
name | setting
```

```
-----+-----
LIBDIR | /opt/tantor/db/17/lib
SHAREDIR | /opt/tantor/db/17/share/postgresql
```

However, you need to add the postgresql and extension subdirectories to these configuration parameters. This is inconvenient to remember. Extensions can be installed by copying files to these directories or by the later "PGXS" method, which appeared in PostgreSQL relatively recently, but is just as inconvenient. For this method, you need to add the directory with the `pg_config` utility to the `PATH` and an environment variable that tells the `make` utility to use the PGXS extension installation logic:

```
root@tantor:~# export PATH=/opt/tantor/db/17/bin:$PATH
root@tantor:~# export USE_PGXS=1
```

then go to the extension directory and run the `make` and `make install` commands .

The method is quite complicated. Therefore, the practice of installing extensions, libraries, utilities, applications using deb and rpm packages is widespread.

Installing extensions

- Extensions may include a shared library and/or text files: an extension control file and one or more script files.
- libraries are loaded using parameters or the command:

```
postgres=# \dconfig *librar*
archive_library          |
dynamic_library_path     | $libdir
local_preload_libraries |
session_preload_libraries |
shared_preload_libraries | pg_stat_statements
postgres=# load 'name';
LOAD
postgres=# \dx
```

- Extension objects are installed by the command:

```
postgres=# create extension name;
CREATE EXTENSION
```

- Extensions available for installation are listed in the `pg_available_extensions` view.



Installing extensions

Extensions may include a shared library and/or text files: an extension control file and one or more script files. If an extension consists of only a library, the library may be loaded in several ways, which must be specified in the library description. The library must be specified in one of the parameters:

```
postgres=# \dconfig *librar*
archive_library          |
dynamic_library_path     | $libdir
local_preload_libraries |
session_preload_libraries |
shared_preload_libraries | pg_stat_statements
```

or the LOAD command:

```
postgres=# load 'library';
LOAD
```

If the extension has objects inside the database, such as functions, procedures, views, tables, etc., then the commands for creating them are specified in the `.sql` script file, and the extension characteristics in the `.control` file. You can see the list of such extensions in the views:

```
postgres=# \dv *exten*
```

List of relations

Schema	Name	Type	Owner
pg_catalog	pg_available_extension_versions	view	postgres
pg_catalog	pg_available_extensions	view	postgres

The list of installed extensions can be viewed using the **\dx** command.

The extension is installed with the command: `CREATE EXTENSION name`, and removed with `DROP EXTENSION`. You can replace the extension with another version or change the extension properties with the command `ALTER EXTENSION`.

If the extension should not be used, a dash is inserted into its name and to install it, the name must be enclosed in double quotes:

```
postgres=# create extension "uuid-oss";
CREATE EXTENSION
```

Particularly unsuccessful extensions also have dashes inserted into the parameter names.

Extension files

- example control file
pg_stat_kcache--2.3.0.control

```
# pg_stat_kcache extension
comment = 'Kernel statistics gathering'
default_version = '2.3.0'
requires = 'pg_stat_statements'
module_pathname = '$libdir/pg_stat_kcache'
relocatable = true
```

- example script file pg_stat_kcache--2.3.0.sql

```
-- complain if script is sourced in psql, rather than via CREATE EXTENSION
\echo Use "CREATE EXTENSION pg_stat_kcache" to load this file. \quit
SET client_encoding = 'UTF8';
CREATE FUNCTION pg_stat_kcache_reset()
RETURNS void
LANGUAGE from COST 1000
AS '$libdir/pg_stat_kcache', 'pg_stat_kcache_reset';
REVOKE ALL ON FUNCTION pg_stat_kcache_reset() FROM public;
...
```



Extension files

Extension files can be viewed to learn how extension objects are created.

The control file has the format name.control

There must also be at least one SQL script file that follows the naming pattern name--version.sql which is located in the same place as the control file - in the SHAREDIR/extension directory, unless the directory parameter is specified in the control file. If an absolute path is not specified, then the path is relative to the SHAREDIR directory, which is equivalent to specifying directory = 'extension'.

Parameters in the control file:

encoding - encoding for script files. Default is the database encoding.

requires - names of extensions separated by commas and spaces, on which this extension depends, without them it will not be installed.

relocatable - whether extension objects can be moved to another schema. By default, it cannot be moved, the value is false.

schema - only for non-movable extensions. Schema in which extension objects are created with the CREATE EXTENSION command. Ignored when updating an extension - objects are not moved.

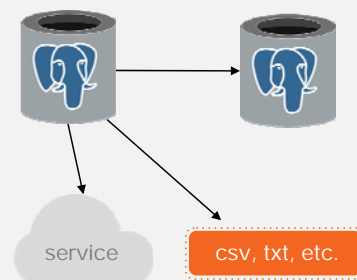
For individual versions of the extension, control files with names like name--version.control may exist in the same place as the control file. The parameters specified in them override the parameters of the main control file.

The format of the script file name is: name--version.sql For version switching scripts: name-version--version.sql The contents of these files are executed in a transaction, so they cannot contain begin, commit, or commands that cannot be executed within a transaction.

https://docs.tantorlabs.ru/tdb/en/17_5/se/extend-extensions.html#EXTEND-EXTENSIONS-FILES

Foreign Data Wrapper

- External Data Wrapper) - functionality for accessing data located outside the database from a session with the PostgreSQL database in a standardized and relatively simple way
- PostgreSQL includes two wrappers (drivers): **postgres_fdw** for working with tables in PostgreSQL databases and **file_fdw** for the contents of text files.
- FDW is installed as an extension
- There are extensions: **mysql_fdw**, **oracle_fdw**, **sqlite_fdw**, **mongo_fdw**, **redis_fdw**
- psql commands to view FDW objects:
\dew, **\des**, **\deu**, **\det**



Foreign Data Wrapper

Foreign Data Wrapper (FDW) is a functionality for accessing data outside the database from a session with a PostgreSQL database in a standardized and relatively simple way. Similar to the functionality of transparent gateways and dblink in Oracle Database. PostgreSQL includes two wrappers (drivers): **postgres_fdw** for working with tables in PostgreSQL databases and **file_fdw** for the contents of text files.

FDW is installed as an extension, and may include a library. The extension implements the logic of the driver (adapter) for accessing an external software system via its protocol. Then the following objects are created:

FOREIGN SERVER - details of connection to the external system are specified:. For example, passwords, database names, network address, port . Example:

```
CREATE SERVER conn1 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'localhost',  
port '5432', dbname 'postgres');
```

USER MAPPING - if the external system has accounts (users, groups, roles), then you can map the cluster roles to the external system accounts

FOREIGN TABLE - always created or imported. For an external data source, a local object is created in the PostgreSQL database that looks like a table or view. When using FDW, work with external data is done as with tables. These tables can be used in queries, connections with regular, temporary and other tables. The insert, update, delete commands can be implemented, but this depends on the external data source. For example, for **file_fdw** , deletion, modification, insertion of lines into text files is not implemented.

Lists of FDW objects can be viewed using psql commands: **\dew**, **\des**, **\deu**, **\det**

There are extensions: **mysql_fdw**, **oracle_fdw**, **sqlite_fdw**, **mongo_fdw**, **redis_fdw** and others.

By default, connections established by **postgres_fdw** to third-party services remain open for reuse in the same session that accessed the external table.

The standard "dblink" extension can be used to access PostgreSQL databases. The functions of this extension can send any commands and receive the result. Its way of working is not similar to Oracle Database dblink. The extension appeared before the FDW specification.

file_fdw extension

- file_fdw allows you to create virtual tables based on data stored in text files
- You can't make changes to the files, you can only read them
- there are no indexes, the entire file is scanned for each SELECT
- Example of creating FDW objects:

```
CREATE EXTENSION file_fdw;
CREATE SERVER csv_server FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE t1 (
column1 text,
column2 numeric,
...
)
SERVER csv_server OPTIONS(filename '/path/to/file.csv', format 'csv');
```



file_fdw extension

file_fdw allows you to create virtual tables based on data stored in files of various formats, such as CSV. It is used to read lines of text files and present them as regular tables. Example:

```
CREATE EXTENSION file_fdw;
CREATE SERVER csv_server FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE t1 (
column1 text,
column2 numeric,
...
)
SERVER csv_server OPTIONS(filename '/path/to/file.csv', format 'csv');
```

https://docs.tantorlabs.ru/tdb/en/17_5/se/file-fdw.html

dblink extension

- allows you to send commands for execution and receive results

```
SELECT * FROM dblink('dbname=postgres user=postgres', $$ select 7; $$ ) as (col1 int);
7
SELECT * FROM dblink_connect('connection1', 'host=/var/run/postgresql port=5432');
OK
SELECT * FROM dblink_send_query('connection1', $$ select 8 from pg_sleep(1); $$ );
1
SELECT dblink_is_busy('connection1');
1
SELECT * FROM dblink_get_result('connection1') as t(col1 int);
8
SELECT dblink_is_busy('connection1');
0
SELECT * FROM dblink_exec('connection1', $$ CHECKPOINT; $$);
CHECKPOINT
SELECT * FROM dblink_disconnect('connection1');
OK
```



file_fdw extension

dblink allows you to send any commands for execution and receive the result. Example:

```
SELECT * FROM dblink('dbname=postgres user=postgres', $$ select 7; $$ ) as (col1
int);
7
SELECT * FROM dblink_connect('connection1', 'host=/var/run/postgresql
port=5432');
OK
SELECT * FROM dblink_send_query('connection1', $$ select 8 from pg_sleep(1); $$
);
1
SELECT dblink_is_busy('connection1');
1
SELECT * FROM dblink_get_result('connection1') as t(col1 int);
8
SELECT dblink_is_busy('connection1');
0
SELECT * FROM dblink_exec('connection1', $$ CHECKPOINT; $$);
CHECKPOINT
SELECT * FROM dblink_disconnect('connection1');
OK
```

https://docs.tantorlabs.ru/tdb/en/17_5/se/dblink.html

Practice

1. Defining the directory with extension files
2. View installed extensions
3. View available extensions
4. Installing and removing custom updates
5. View available extension versions
6. Update to the latest version
7. External data wrappers



Practice

Defining the directory with extension files

View installed extensions

View available extensions

Installing and removing custom updates

View available extension versions

Update to the latest version

External data wrappers



3

Configuring PostgreSQL



Review

- there are about 370 parameters
- affect the operation of the instance
- title is case insensitive
- extensions and applications can use their own configuration parameters, such parameters have a dot in the name
- instance tuning - setting the values of configuration parameters so that the instance operates optimally under the current load



Review

There are approximately 370 configuration parameters that affect the operation of the instance.

Tuning an instance mostly involves setting configuration parameter values at various levels so that the instance operates optimally under the current load.

Parameters have a name (case insensitive) and a value.

Parameter values can be:

logical (the "bool" value in the vartype column of the pg_settings view)

string ("string")

integers ("integer", "int64")

decimal numbers ("real")

numbers ("integer", "int64", "real") with a unit of measurement in bytes or time

values from the list ("enum").

The parameter type names are not related to SQL data types. The maximum and minimum values of numeric types for each parameter are specified in the min_val and max_val columns of the pg_settings view.

It is better to enclose the values of string parameters in apostrophes. If the value itself contains an apostrophe, then duplicate the apostrophe (two apostrophes).

For numeric parameters with units of measurement, the following are acceptable unit designations (case sensitive): B (bytes), kB (kilobytes), MB (megabytes), GB (gigabytes), and TB (terabytes); us (microseconds), ms (milliseconds), s (seconds), min (minutes), h (hours), and d (days). It is better to enclose the values themselves in apostrophes.

For "enum", the list of valid values can be found in the enumvals column of the pg_settings view.

Extensions and applications can define and use their own configuration parameters, such parameters have a period in their name.

https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-custom.html

https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config.html

Configuration parameters

- The main configuration file `postgresql.conf` is viewed first
. Parameters may be duplicated; the latest one is applied.
- Next, the parameters `$PGDATA/postgresql.auto.conf` are applied
- Command line parameters passed to the `postgres -c` process take precedence over those set in the `pg_ctl start -o "-c config_file=/opt/postgresql.conf"` parameter files.



Configuration parameters

When creating a cluster, two files are created:

1) the main file with cluster configuration parameters `postgresql.conf`

If the cluster is running, the location can be seen in the value of the `config_file` parameter

. You can set the value of the `config_file` parameter **only on the command line when starting the cluster** .

You can see the parameters of the main process: `postgres --help`

`postgres` is the PostgreSQL server.

Usage:

`postgres [OPTION]...`

Options:

`-B NBUFFERS` number of shared buffers

`-c NAME = VALUE` set run-time parameter

`-C NAME` print value of run-time parameter, then exit

`-d 1-5` debugging level

`-D DATADIR` database directory

`-c` switch can be used to pass any configuration parameters. Example:

`pg_ctl start -o "-c config_file = /opt/postgresql.conf "`

2) file `postgresql.auto.conf` It is always located in the `PGDATA` directory . If the cluster is running, the location of `PGDATA` can be found in the value of the `data_directory` parameter
`postgresql.conf` file outside the `PGDATA` directory when backing up and restoring. The `pg_basebackup` backup utility copies only the contents of `PGDATA` (and tablespaces) , so parameters specific to the backup node can be placed in `postgresql.conf` outside `PGDATA` and this file will not be overwritten during restoration. The `pg_rewind` utility also synchronizes only the `PGDATA` directory and tablespaces.

View parameters

- psql command `\dconfig`
 - you can search by pattern `\dconfig *name*`
 - you can use tab key
- Command `SHOW parameter_name;`
 - you can use the tab key
 - you need to send it for execution by typing `" ; "`
 - only one parameter
- function `current_setting('parameter name')`



View parameters

The current values of cluster parameters can be conveniently viewed using the psql `\dconfig parameter_mask` command

For example:

```
postgres=# \dconfig *data_d*
```

List of configuration parameters

Parameter	Value
-----------	-------

-----+-----

data_directory	/var/lib/postgresql/tantor-se-17/data
----------------	---------------------------------------

data_directory_mode	0750
---------------------	------

will show the values of the parameters where the string `data_d` occurs

The `SHOW` command will show the current values of the parameter. The tab key in psql will show the allowed values. `SHOW` shows one parameter. The inconvenience of the `SHOW` command is that it must be terminated with `" ; "` otherwise it will remain in the psql buffer .

```
postgres=# show data_directory;
```

```
data_directory
```

```
/var/lib/postgresql/tantor-se-17/data
```

```
(1 row)
```

You can clear the psql buffer with the command `\r`

```
postgres=# \r
```

```
Query buffer reset (cleared).
```

`current_setting(parameter name)` function is an analogue of the `SHOW` command

View parameters

- Views `pg_settings` `pg_file_settings`
- Команда `SHOW ALL;`
 - аналог `SELECT * FROM pg_settings;`
 - нельзя отфильтровать параметры
- Файлы `postgresql.conf` `postgresql.auto.conf`

```
ls -CF /var/lib/postgresql/tantor-se-16/data
```

```
base/          pg_logical/    pg_stat/       pg_wal/
global/        pg_multixact/  pg_stat_tmp/   pg_xact/
pg_commit_ts/  pg_notify/     pg_subtrans/   postgresql.auto.conf
pg_dynshmem/   pg_replslot/   pg_tblspc/     postgresql.conf
pg_hba.conf    pg_serial/     pg_twophase/   postmaster.opts
pg_ident.conf  pg_snapshots/  PG_VERSION     postmaster.pid
```

- view a single parameter on a running or stopped postgres instance `-C parameter_name`



View parameters

The current values of cluster parameters can be conveniently viewed using the `psql \dconfig parameter_mask` command

For example:

```
postgres=# \dconfig *data_d*
```

List of configuration parameters

```
Parameter | Value
```

```
-----+-----
data_directory | /var/lib/postgresql/tantor-se-17/data
data_directory_mode | 0750
```

will show the values of the parameters where the string `data_d` occurs

The `SHOW` command will show the current values of the parameter. The tab key in `psql` will show the allowed values. `SHOW` shows one parameter. The inconvenience of the `SHOW` command is that it must be terminated with `;` ; otherwise it will remain in the `psql` buffer .

```
postgres=# show data_directory;
```

```
data_directory
```

```
-----
/var/lib/postgresql/tantor-se-17/data
```

```
(1 row)
```

You can clear the `psql` buffer with the command `\r`

```
postgres=# \r
```

Query buffer reset (cleared).

Function `current_setting(parameter name)` - analogous to the `SHOW` command .

View parameters

- Views `pg_settings` `pg_file_settings`
- `SHOW ALL`
command ; - similar to `SELECT * FROM pg_settings;`
- parameters cannot be filtered

- PostgreSQL.conf files `postgresql.auto.conf`

```
ls -CF /var/lib/postgresql/tantor-se-16/data
```

```
base/          pg_logical/    pg_stat/       pg_wal/
global/        pg_multixact/  pg_stat_tmp/   pg_xact/
pg_commit_ts/  pg_notify/     pg_subtrans/   postgresql.auto.conf
pg_dynshmem/   pg_replslot/   pg_tblspc/     postgresql.conf
pg_hba.conf    pg_serial/     pg_twophase/   postmaster.opts
pg_ident.conf  pg_snapshots/  PG_VERSION     postmaster.pid
```

- view a single parameter on a running or stopped
postgres instance `-C parameter_name`



View parameters

`pg_file_settings` view contains parameters that **are explicitly specified in the parameter files** . This view can be useful for pre-testing changes to configuration files - to see if an error was made when editing the files. The `pg_file_settings` view does not show the current values that the instance is using. The `applied` column has the value **"f"** if the parameter value is different from the current one **and** a cluster restart is required to apply the value from the file. In other cases (the value has not changed or it is enough to reread the files), the value in the `applied` column will be `"t"`.

`pg_settings` view shows the current effective values of the parameters. The `SHOW ALL;` command is similar to a query on the `pg_settings` view , but you cannot display only some of the parameters, so `SHOW ALL;` is inconvenient.

The contents of any file can be viewed using the function

```
SELECT pg_read_file ('./postgresql.auto.conf') \g (tuples_only=on
format=unaligned)
```

`pg_hba_file_rules` view shows the contents of the `pg_hba.conf` file . The error column of this view provides a description of the error if one was made while editing the file. The `pg_hba.conf` and `pg_ident.conf` files contain security settings.

`postgresql.conf` file is edited manually.

View a single parameter on a running or stopped instance:

```
postgres -C parameter_name
```

https://docs.tantorlabs.ru/tdb/en/17_5/se/config-setting.html

Main postgresql.conf parameter file

- the main file that stores cluster configuration parameters
- can contain more than 370 parameters plus parameters of extensions and shared libraries (*.so)
- Created based on the postgresql.conf.sample file by the initdb utility
- Can be edited
- You can include the contents of other files using the include and include_dir configuration parameters set inside postgresql.conf



Main postgresql.conf parameter file

postgresql.conf file is the main file that stores the cluster configuration parameters. There are about 370 configuration parameters, plus parameters for extensions and shared libraries (*.so) loaded using the shared_preload_libraries configuration parameter.

The file is created by the initdb utility from the example file

```
/opt/tantor/db/17/share/postgresql$ ls -w 1 *.sample
```

```
pg_hba.conf.sample
pg_ident.conf.sample
pg_service.conf.sample
postgresql.conf.sample
psqlrc.sample
```

to the *.sample files . Commented lines start with the # symbol.

initdb utility makes changes to some of the lines depending on the parameters passed to it, environment variables set before its launch, and internal logic. The changes can be viewed by comparing the files

```
diff postgresql.conf postgresql.conf.sample
```

```
65c65
```

```
< max_connections = 100 # (change requires restart)
```

```
---
```

```
> #max_connections = 100 # (change requires restart)
```

The list of parameters that postgres responds to (not extension parameters and arbitrary application parameters) can be output to the file

```
postgres --describe-config > file.txt
```

The columns in the file are separated by tabs.

Using include and include_dir can be useful for companies that provide cloud solutions in the form of a large number of clusters with almost the same configuration for different clients. But you need to remember that the parameter specified "below" overrides the parameter specified "above" (closer to the beginning of the configuration file).

Parameter file postgresql.auto.conf

- Located in the PGDATA directory
- Parameters are added by the command

```
> ALTER SYSTEM SET parameter = value;
```
- The parameters are removed by the command

```
ALTER SYSTEM RESET parameter;  
ALTER SYSTEM RESET ALL;
```
- In order for the new values to take effect after changing with the ALTER SYSTEM command , you need to reread the configuration or reboot the cluster.



Parameter file postgresql.auto.conf

postgresql.auto.conf file is a text file located in the PGDATA directory . It can be edited directly, but it is not recommended because you can make a typo. The purpose of its creation is to be able to make changes to the cluster configuration parameters using the ALTER SYSTEM command , including when connecting over a network, without having to edit files in the server file system.

Syntax

```
ALTER SYSTEM SET parameter { TO | = } { value [, ...] | DEFAULT };  
ALTER SYSTEM RESET parameter;  
ALTER SYSTEM RESET ALL;
```

Changes after this command, as well as after editing any configuration files, are not applied, you need to reread the configuration or reboot the cluster. Rebooting the cluster is only needed to apply parameters that cannot be changed dynamically (without rebooting the cluster). Such parameters can be called "static".

Only users with the SUPERUSER attribute and users who have been granted the ALTER SYSTEM privilege can change cluster parameters using the ALTER SYSTEM command .

The command cannot be executed within a transaction.

Applying parameter changes

- To apply changes, all configuration files are reread and changes are applied that can be made without rebooting the cluster.
- Methods:
 - > `SELECT pg_reload_conf();`
 - > `pg_ctl reload`
 - > `sudo killall -1 postgres`
- Parameters in `postgresql.auto.conf` override values of parameters in `postgresql.conf`
- If a parameter is specified multiple times in `postgresql.conf`, the one specified closer to the end of the file is applied.



Applying Configuration Parameter Changes

To apply changes (reread) in text files of configuration parameters, it is most convenient to use the function

```
SELECT pg_reload_conf();  
pg_reload_conf  
-----
```

```
t
```

```
(1 row)
```

Can be used `pg_ctl`:

```
pg_ctl reload -D /var/lib/postgresql/tantor-se-17/data  
server signaled
```

You can send the SIGHUP signal (number 1) to the main process.

For example, to send a signal to processes named postgres (synonym "postmaster") **of all running**

PostgreSQL instances:

```
killall -1 postgres
```

Parameters set in `postgresql.auto.conf` override the values of `postgresql.conf` parameters

If a parameter is specified multiple times in `postgresql.conf`, the one specified closest to the end of the file is applied.

Privileges to change parameters

- The parameter type is specified in the `context` column of the `pg_settings` view.
- The `ALTER SYSTEM` command can be executed by a role with the `SUPERUSER` attribute and by those roles that have been granted the `GRANT ALTER SYSTEM` on `PARAMETER parameter_name` to role privilege;
- You can view the list of privileges using the `psql` command:

```
\dconfig+ privilege_name
```

Parameter	Value	Type	Context	Access privileges
update_process_title	off	bool	superuser	postgres= sA /postgres+ user1= s /postgres

- or by query `select * from pg_parameter_acl;`
- **A** - right to `ALTER SYSTEM`, **s** - right to `SET`



Privileges to change parameters

Some configuration parameters can only be changed by a role with the `SUPERUSER` attribute.

```
alter user user1 superuser;
```

Starting with version 16, it became possible to grant the privilege to change parameters that can only be changed by a role with the `SUPERUSER` attribute.

Granting privilege to change configuration parameter:

```
create role user1 login;
```

```
grant alter system on parameter update_process_title to user1;
```

There is also a privilege to set parameters at the session level:

```
grant set on parameter update_process_title to user1;
```

You can revoke the granted privilege using the command

```
revoke alter system, set on parameter update_process_title from user1;
```

You can view the list of privileges using the `psql` command:

```
\dconfig+ *
```

The privileges will be listed in the **Access privileges** column.

```
postgres=# \dconfig+ update_process_title
```

List of configuration parameters

Parameter	Value	Type	Context	Access privileges
update_process_title	off	bool	superuser	postgres= s A /postgres+ user1= s /postgres

Where **A** - right to `ALTER SYSTEM`, **s** - right to `SET`.

Disadvantage - you can't filter by the presence of a privilege. It is more convenient to use the query `select * from pg_parameter_acl;` which gives only those parameters for which privileges have been assigned

https://docs.tantorlabs.ru/tdb/en/17_5/se/ddl-priv.html

https://docs.tantorlabs.ru/tdb/en/17_5/se/view-pg-settings.html

Parameter Classification: Context

- internal - read-only parameters
- postmaster - cluster instance restart required
- sighup - just reread the configuration files
- superuser - can be set at session level, but user must have SUPERUSER attribute or privilege to change this parameter
- superuser-backend - cannot be changed after session creation, but can be set for a specific session at connect time if privileges are present
- backend - same as the previous one, but no privileges are needed
- user - can be changed during a session or at the cluster level



Parameter Classification: Context

There are many configuration parameters - more than 370. Next, let's look at how parameters are classified. The first section of classification is by the method (context) of applying the parameter.

The context column of the `pg_context` view has 7 possible values.

```
select context, count(name) from pg_settings  
where name not like '%.%' group by context order by 1;
```

```
context | count  
-----+-----  
backend | 2  
internal | 18  
postmaster | 64  
sighup | 96  
superuser | 44  
superuser-backend | 4  
user | 143  
(7 rows)
```

internal - not set in configuration files and are read-only

postmaster - requires restarting the cluster instance to apply

sighup - to use it, it is enough to reread the files, for example, execute the `pg_reload_conf()` function or the `pg_ctl reload` utility

superuser - can be set at session level, but user must have SUPERUSER attribute or privilege to change this parameter

superuser-backend - cannot be changed after session creation, but can be set for a specific session at connect time if privileges are present

backend - cannot be changed after session creation, but can be set for a specific session at the time of connection by any role

user - can be changed during a session or at the cluster level in the parameter files, in the latter case by rereading the files

https://docs.tantorlabs.ru/tdb/en/17_5/se/view-pg-settings.html

Context parameters internal

- 19 parameters in version 17
- read only

```
postgres=# select name, setting from pg_settings where context='internal' order by 1;
-----
name                | setting
-----
block_size           | 8192
data_checksums       | on
data_directory_mode  | 0700
debug_assertions     | off
huge_pages_status    | off
in_hot_standby       | off
integer_datetimes    | on
max_function_args    | 100
max_identifier_length | 63
max_index_keys       | 32
segment_size         | 131072
server_encoding       | UTF8
server_version       | 17.2
server_version_num    | 170002
shared_memory_size   | 189
shared_memory_size_in_huge_pages | 95
ssl_library           | OpenSSL
wal_block_size       | 8192
wal_segment_size     | 16777216
(19 rows)
```



Context parameters internal

In PostgreSQL version 17, there are 19 parameters whose values cannot be changed. They are not set in configuration files and are read-only.

Some parameters are set during assembly and set PostgreSQL restrictions (limits). Some parameters are descriptive - reflect the current operating mode of the instance or cluster and will change the value when the mode is changed according to the documented procedure.

The list of parameters of this type (**internal**) can be viewed by query:

```
select * from pg_settings where context=' internal ' order by 1;
```

Parameters whose values can change:

`in_hot_standby` - descriptive parameter for the replica

`data_directory_mode` - descriptive, shows the permissions that were set on the `data_directory` (PGDATA) at the time the instance was started

`server_encoding` - set when creating a cluster

`server_version` and `server_version_num` - the procedure for updating the version

`wal_segment_size` - changed by `pg_resetwal` utility

`shared_memory_size*` - descriptive parameters, depend on `huge_page_size`

https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-preset.html

Classification of parameters: Levels

- Cluster team
 - > `ALTER SYSTEM SET parameter = value;`
- Databases
 - > `ALTER DATABASE name SET parameter = value;`
- Roles
 - > `ALTER ROLE role SET parameter = value;`
- Sessions created by a role that connects to a specific database
 - > `ALTER ROLE role IN DATABASE name SET parameter = value;`



Classification of parameters: Levels

If a parameter in the context column of the `pg_settings` view has a value other than `internal`, then this parameter can be changed using the `ALTER SYSTEM` command or by editing the configuration parameter files.

If the parameter in the context column of the `pg_settings` view has the values `user`, `backend`, `superuser`, then the value of the parameter can be changed at other levels:

At the database level, you can set the parameter value using the commands:

```
ALTER DATABASE name SET parameter { TO | = } { value | DEFAULT };
```

```
ALTER DATABASE name SET parameter FROM CURRENT;
```

```
ALTER DATABASE name RESET parameter;
```

```
ALTER DATABASE name RESET ALL;
```

At the role level or the role connected to the database:

```
ALTER ROLE .. [ IN DATABASE name ] SET parameter { TO | = } { value | DEFAULT }
```

```
ALTER ROLE .. [ IN DATABASE name ] SET parameter FROM CURRENT;
```

```
ALTER ROLE .. [ IN DATABASE name ] RESET parameter;
```

```
ALTER ROLE .. [ IN DATABASE name ] RESET ALL;
```

Note: The category column of the `pg_settings` view reflects the name of the subsystem affected by the setting, not the installation level. This column is used to classify settings.

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-alterdatabase.html

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-alterrole.html

Classification of parameters: Levels

- Within a session
 - > SET work_mem to '16MB';
 - > SELECT set_config('work_mem', '16MB', false);
- Transactions
 - > SET LOCAL work_mem to '16MB';
 - > SELECT set_config('work_mem', '16MB', true);
- During the execution of a function or procedure
 - > CREATE {FUNCTION|PROCEDURE} ..
SET parameter {TO value | = value | FROM CURRENT}
 - > ALTER {PROCEDURE | FUNCTION}..
SET parameter {TO | = } {value | DEFAULT};



Classification of parameters: Levels

At the transaction level, the value is changed using the SET LOCAL command .

Example:

SET work_mem to '16MB'; or SELECT set_config('work_mem', '16MB', false); if false , then set at session level

SET work_mem to DEFAULT; resets the parameter to the value that it would have if no SET commands had been executed in the current session

RESET work_mem; same as previous command

SET LOCAL work_mem to '16MB'; or SELECT set_config('work_mem', '16MB', true);

ALTER {PROCEDURE | FUNCTION} and then one of the following:

SET parameter { TO | = } { value | DEFAULT };

SET parameter FROM CURRENT

RESET parameter

RESET ALL

The last two options remove from the subroutine properties the parameter values that were previously set (when it was created or modified).

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-set.html

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-alterprocedure.html

Table-level storage parameters

- Installed by command
 - `ALTER TABLE name SET(storage_parameter = value);`
- Can be set in the CREATE TABLE command
- Override similar configuration parameters when autovacuum works with a specific table and/or its TOAST table
- The command to override the default_statistics_target configuration parameter for a table column:
 - `ALTER TABLE name ALTER COLUMN name SET STATISTICS number;`



Table level parameters

At the table **and index level** , it is possible to set storage parameters. Storage parameters at the table level can override the parameters for autovacuum when working with a table and/or its TOAST table.

```
ALTER TABLE name SET ( storage_parameter = value);
```

`ALTER TABLE name ALTER COLUMN name SET STATISTICS number;` overrides the `default_statistics_target` configuration parameter for the table column. Range is 0 to 10000. -1 reverts to use `default_statistics_target` .

```
ALTER INDEX name ALTER COLUMN index_column_number SET STATISTICS number;
```

overrides the `default_statistics_target` configuration parameter for the index column.

Options with the "toast." prefix affect the operation of the TOAST table. If they are not set, the TOAST table options are in effect.

```
postgres=# alter table name set (toast. <press tab key twice>
```

```
toast.autovacuum_enabled
toast.autovacuum_freeze_max_age
toast.autovacuum_freeze_min_age
toast.autovacuum_freeze_table_age
toast.autovacuum_multixact_freeze_max_age
toast.autovacuum_multixact_freeze_min_age
toast.autovacuum_multixact_freeze_table_age
toast.autovacuum_vacuum_cost_delay
toast.autovacuum_vacuum_cost_limit
toast.autovacuum_vacuum_insert_scale_factor
toast.autovacuum_vacuum_insert_threshold
toast.autovacuum_vacuum_scale_factor
toast.autovacuum_vacuum_threshold
toast.log_autovacuum_min_duration
toast.vacuum_index_cleanup
toast.vacuum_truncate
```

```
postgres=# alter table t set (toast. <press tab key twice>
```

There are quite a few index types in PostgreSQL. Index storage parameters depend on their type. For example, for indexes of the btree, hash, GiST, SP-GiST type, you can set the fillfactor parameter. For btree - `deduplicate_items`. For GiST - `buffering`. For GIN - `fastupdate`. For BRIN - `pages_per_range` and `autosummarize`. In PostgreSQL, you can add both indexes and data storage methods to tables using extensions.

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createtable.html

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-altertable.html

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createindex.html

Classification of parameters: Categories

- Categories describe what the parameters are for.

```
select category, count(name) from pg_settings group by category order by 2 desc;
```

category	count
Client Connection Defaults / Statement Behavior	31
Developer Options	25
Resource Usage / Memory	22
Query Tuning / Planner Method Configuration	22
Reporting and Logging / What to Log	21
Preset Options	18
Write-Ahead Log / Settings	15
Connections and Authentication / SSL	14
Autovacuum	13
Query Tuning / Planner Cost Constants	13
...	
(42 rows)	



Classification of parameters: Categories

The parameters are logically divided into categories. The categories describe what the parameters are intended for. The names of the categories can be viewed by the query:

```
select category, count(name) from pg_settings group by category order by 2 desc;
```

category	count
Client Connection Defaults / Statement Behavior	31
Developer Options	25
Resource Usage / Memory	22
Query Tuning / Planner Method Configuration	22
Reporting and Logging / What to Log	21
Preset Options	18
Write-Ahead Log / Settings	15
Connections and Authentication / SSL	14
Autovacuum	13
Query Tuning / Planner Cost Constants	13
Reporting and Logging / Where to Log	13
Client Connection Defaults / Locale and Formatting	12
Replication / Standby Servers	11
...	
(42 rows)	

Many parameters relate to performance tuning: query execution tuning, autovacuum.

Category: "For developers"

- Category of options `Developer Options`
- Parameters of this category should not be used in industrial operation.
- The parameters in this category can help to retrieve data in complex cases of page corruption.
- `ignore_system_indexes` - ignore system indexes when reading system tables (but still update indexes when tables are modified). This can be useful if corruption in system indexes prevents the creation of a session to repair the corruption.
- `zero_damaged_pages` - damage in the service area of the block prevents reading data from this block. The parameter allows you to consider the block empty, as if there are no lines in it



Category: "For developers"

As an example, let's look at the parameters of the `Developer Options` category .

`Developer Options` category includes options that should not be used in a production database. However, some of these options can be used to restore the contents of tables if a block is damaged in them and recovery by other means has not been successful (the block is damaged in physical replicas and backups). An example of such options is:

`ignore_system_indexes`

Ignore system indexes when reading system tables (but still update indexes when tables are modified). This can be useful if corruption in system indexes prevents the creation of a session to repair the corruption.

`zero_damaged_pages`

Damage in the service area of a block (page) usually prevents reading data on this page and the selection of rows (using the `SELECT` command) is interrupted. This parameter allows you to skip the contents of the page, considering that there are no rows in it, and continue working with other pages. This allows you to extract rows from undamaged pages. However, at the logical level, the integrity of the data may suffer. The parameter does not change the contents of the pages: they remain damaged, and are not filled with zeros.

https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-developer.html

Category: "Custom Settings"

- Category of parameters Customized Options
- The category includes parameters of extensions, libraries or just arbitrary parameters, in the name of which there is a dot
- library and extension developers provide the ability to customize their functionality in a standard way

```
SET myapp.par1 = '20';
```

```
SET
```

```
SHOW myapp.par1;
```

```
myapp.par1
```

```
-----
```

```
20
```

```
(1 row)
```



Category: "Custom Settings"

Extensions and libraries loaded by the `shared_preload_libraries` parameter or the `LOAD` command can have their own configuration parameters. These parameters are processed according to the logic of regular parameters. However, these parameters are unknown to the DBMS until the module is loaded. In particular, the DBMS cannot check the validity of parameter values when they are changed, for example, by the `ALTER SYSTEM` command, so before loading the library, this command cannot set parameters unknown to the DBMS, even if the parameter name contains a dot. They can be set at the session level. By default, if a parameter name contains a dot, the DBMS considers such parameters customized options (can be translated as "user settings", "non-system parameters"). Extension and library developers specify the name of their extension as a prefix and come up with parameter names. You can also save arbitrary parameter names if the name contains a dot in `postgresql.conf`. As soon as the library is loaded (for example, the `LOAD` command) and "registers" its parameters by a program call, the DBMS checks the parameter values and if they are invalid, it sets them to the default value specified by the library. Those parameters that the library did not register when loading by a program call are removed from memory, as if they were not set in the `postgresql.conf` configuration file and at other levels. A warning about this can be written to the cluster log.

Parameter names **without a period** in the name must exist in the DBMS; using a non-existent parameter name (for example, a typo) in the `postgresql.conf` file will prevent the cluster from starting.

```
waiting for server to start....
```

```
LOG: unrecognized configuration parameter "myappparam1" in file
```

```
"/var/lib/postgresql/tantor-se-17/data/postgresql.conf" line 834
```

```
FATAL: configuration file "/var/lib/postgresql/tantor-se-17/data/postgresql.conf" contains errors
```

https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-custom.html

Configuration parameter names and values

- Parameter names are case insensitive.
- Parameter values are one of five types:
- **boolean**: values can be specified as `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0`
- **string**: values are best specified in apostrophes
- **integer or decimal number**: in units from the `unit` column of the `pg_settings` view , or just a number
- **enum** : value from the list in the `enumvals` column of the `pg_settings` view



Configuration parameter names and values

At the beginning of the chapter we considered that parameter values can be of several types. Let's look at them in more detail. Parameter types:

Boolean value: Values can be set as `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0`

String: It is better to use apostrophes. If there is an apostrophe symbol inside the string, then put two apostrophes instead of one. If the string contains an integer, quotation marks are not necessary

Integer or decimal number : If the integer is written in hexadecimal (starts with `0x`) it must be enclosed in quotation marks. If there is a zero at the beginning, then it is an integer value in octal.

Number with unit: Some numeric parameters have an implicit unit, as they describe the amount of memory or time. If you specify a number without a unit, the number can be interpreted as a byte, kilobyte, block, milliseconds, seconds, minutes. The unit can be found in the `unit` column of the `pg_settings` view . It is convenient to use the unit as a suffix. It can be specified immediately after the number or after one space. In any case, be sure to enclose the values in apostrophes. Valid memory units (case is important):

B (bytes), **kB** (kilobytes), **MB** (megabytes), **GB** (gigabytes) and **TB** (terabytes).

Valid icons for time:

us (microseconds), **ms** (milliseconds), **s** (seconds), **min** (minutes), **h** (hours) and **d** (days).

Enum: written in the same way as string parameters, but limited to a set of valid values that are case-insensitive. The list of values is specified in the `enumvals` column of the `pg_settings` view .

Configuration parameter `transaction_timeout`

- allows you to cancel any transaction that lasts longer than a specified period of time
- the effect of the parameter applies to both explicit transactions (started with the `BEGIN` command) and implicitly started transactions corresponding to a separate command
- A value of zero (default) disables the timeout.
- Allows you to set a limit on the duration of transactions
- `statement_timeout` allows you to set the maximum execution time for a single command



Configuration parameter `transaction_timeout`

Let's look at some examples of configuration parameters. This will help you understand how changing parameter values affects the operation of the instance.

`transaction_timeout` allows you to cancel any transaction or single statement that exceeds the specified time period, not just idle ones. This parameter applies to both explicit transactions (started with the `BEGIN` command) and implicitly started transactions corresponding to a single statement. Introduced in Tantor Postgres 15.4. A value of zero (the default) disables the timeout.

`statement_timeout` allows you to set the maximum execution time for a single command. If the time is exceeded, the command is interrupted. The time is counted from the moment the server process receives the command until its execution is complete.

Transactions and single queries (using snapshot) hold the database event horizon. This prevents old row versions from being purged. The `transaction_timeout` and `statement_timeout` parameters allow you to protect the horizon from being held by transactions and queries.

To protect against idle transactions, you can use `idle_in_transaction_session_timeout`. If exceeded, the session is terminated:

```
postgres=# commit;
```

```
IMPORTANT: Connection closed due to idle timeout in transaction
```

```
The server unexpectedly closed the connection
```

```
Most likely the server stopped working due to a failure.
```

```
before or during the execution of a request.
```

```
Connection to the server was lost. Recovery attempt was successful.
```

`transaction_timeout` parameter can be set at the session level, which allows it to be used to implement logic by which, after a certain time, the results of transactions are no longer relevant and are not needed.

https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-resource.html

Autonomous transactions

- Tantor Postgres SE has autonomous transactions
- used in plpgsql routines
- allow you to execute commands and save the results of their execution regardless of whether the transaction in which the autonomous transaction was called is committed, rolled back, or aborted

```
postgres=# CREATE TABLE t (a int);
CREATE TABLE
postgres=# CREATE OR REPLACE PROCEDURE p()
LANGUAGE plpgsql
AS $$
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO t VALUES (1);
END;
$$;
CREATE PROCEDURE
postgres=# BEGIN TRANSACTION;
BEGIN
postgres=# CALL p();
CALL
postgres=# ROLLBACK;
ROLLBACK
postgres=# SELECT * FROM t;
 a
---
 1
(1 row)
```



Autonomous transactions

Autonomous transactions can be implemented, for example, via dblink to your own database, but the problem is in performance. Autonomous transactions in Tantor Postgres SE provide high-speed implementation of autonomous transactions. A pool of autonomous sessions is created, serviced by background workers. The pool is generated when the first autonomous transaction is created. Server processes grab a session from the pool, pass the autonomous transaction operators for execution, and return the connection to the pool. Resources for creating and stopping processes servicing autonomous transactions are not spent. The server and background processes exchange data synchronously via shared memory. Nested autonomous transactions are allowed. Additional (up to a hundred) background processes are launched to service nested autonomous transactions.

An example of how an autonomous transaction works:

```
CREATE TABLE tbl(a int);
CREATE OR REPLACE FUNCTION func() RETURNS void
LANGUAGE plpgsql
AS $$
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO tbl VALUES (1);
END;
$$;
START TRANSACTION;
SELECT func();
ROLLBACK;
SELECT * FROM tbl;
```

The implementation of autonomous transactions was proposed by Tantor Labs to the community:

<https://www.postgresql.org/message-id/f7470d5a-3cf1-4919-8404-5c4d91341a9f@tantorlabs.com>

Configuration parameter `transaction_buffers`

- sets the size of shared memory used to cache the contents of `PGDATA/pg_xact` - the subdirectory with transaction commit status
- The default value is 0, which is equal to the size of the shared buffer pool divided by 512 (`shared_buffers/512`)



Configuration parameter `transaction_buffers`

PostgreSQL has buffers in shared memory of an instance, which are called "SLRU buffers" because they use the Simple Least Recently Used buffer eviction algorithm. Starting with PostgreSQL version 17 (in Tantor Postgres starting with version 15.4), the sizes of SLRU caches can be set by the configuration parameters `commit_timestamp_buffers`, `multixact_member_buffers`, `multixact_offset_buffers`, `notify_buffers`, `serializable_buffers`, `subtransaction_buffers`, `transaction_buffers` .

The default values of the `commit_timestamp_buffers`, `transaction_buffers`, `subtransaction_buffers` parameters are set depending on the size of the buffer cache (the value of the `shared_buffers` parameter).

`transaction_buffers` parameter specifies the size of shared memory used to cache the contents of the `PGDATA/pg_xact` subdirectory containing transaction commit status. The default value is 0, which is equal to the size of the shared buffer pool divided by 512 (`shared_buffers/512`), but not less than 4 blocks. Changing this value requires restarting the instance.

Caching helps to quickly determine the status of a transaction. The need to determine the status of recent transactions and up to the horizon of the cluster databases arises very often for server processes. When processes see versions of changed rows in blocks, they often need to determine the transaction status of each row version processed. Transaction commit status uses vacuum to find the transaction status when cleaning up old row versions. Commit status uses two bits per transaction (committed `COMMIT` or explicitly rolled back `ROLLBACK` or implicitly aborted). If

`autovacuum_freeze_max_age` set to the maximum allowed value for 32-bit transaction counters of 2 billion, `pg_xact` is expected to be about half a gigabyte in size, and `pg_commit_ts` is expected to be about 20 GB.

The downside of increasing the value of `autovacuum_freeze_max_age` (as well as `vacuum_freeze_table_age`) is that the `pg_xact` subdirectories And `pg_commit_ts` database cluster will take up more space. The default value in builds using a 32-bit transaction counter is 200 million transactions, which corresponds to about 50 MB of `pg_xact` storage and about 2 GB of `pg_commit_ts` storage . For 64-bit counters, the default value for `autovacuum_freeze_max_age` is 10 billion.

Subtransaction statuses are also saved. When a top-level transaction is committed or rolled back, the subtransaction statuses (two bits each) are also written to the `pg_xact` subdirectory . When a top-level transaction is aborted, all of its subtransactions are aborted as well.

Parameters `multixact_members_buffers` and `multixact_offsets_buffers`

- sets the size of the shared memory used to cache the contents of the two `PGDATA/pg_multixact` subdirectories
- **verify that the cluster uses 64-bit transaction identifiers** by the parameter values:

```
\dconfig autovacuum_*age
```

List of configuration parameters

Parameter	Value
autovacuum_freeze_max_age	10000000000
autovacuum_multixact_freeze_max_age	20000000000

- If the values are greater than 4 billion, then 64-bit transaction and multitransaction counters are used



Parameters `multixact_members_buffers` and `multixact_offsets_buffers`

Instances may experience performance degradation if there are a large number of concurrent transactions, subtransactions, many multitransactions, or `SERIALIZABLE` transactions. Increasing the buffer sizes ("SLRU caches") helps improve performance.

The PostgreSQL parameters `multixact_offsets_buffers` and `multixact_members_buffers` set the size of shared memory used to cache the contents of two `PGDATA/pg_multixact` subdirectories that store the history of completed and ongoing multitransactions. The history is needed to check the status (not completed, committed, aborted) of transactions. Changing the parameter values requires restarting the instance.

Vacuuming allows you to remove old files from the `pg_multixact/members` and `pg_multixact/offsets` subdirectories.

Since only one transaction ID (the "xmax" field) can be stored in a row header, PostgreSQL uses multitransactions to support row locking by multiple transactions simultaneously. The list of transactions included in a multitransaction ID is stored in the `pg_multixact` subdirectory.

Tantor Postgres SE uses 64-bit transaction IDs, which are unlikely to max out and do not require modulo 32 arithmetic to compare them. At the page level, a wrap-around problem is possible if a session holds a snapshot that has accumulated more than 4 billion transactions.

You can check that the cluster uses 64-bit transaction identifiers by the parameter values:

```
\dconfig autovacuum_*age
```

List of configuration parameters

Parameter	Value
autovacuum_freeze_max_age	10000000000
autovacuum_multixact_freeze_max_age	20000000000

The values shown are 10 billion and 20 billion, which is greater than the 4 billion maximum for 32-bit numbers.

subtransaction_buffers configuration parameter

- sets the size of shared memory used to cache the contents of PGDATA/pg_subtrans
- subtransaction is created:
 - › SAVEPOINT team
 - › if the block in plpgsql language contains the EXCEPTION section
 - › in psql implicitly before each command in a transaction, if the parameter is enabled (interactive or on) by the command:
`\set ON_ERROR_ROLLBACK interactive`
- the presence of the EXCEPTION section implicitly sets a savepoint before the start of the block (before BEGIN)



subtransaction_buffers configuration parameter

subtransaction_buffers specifies the size of shared memory used to cache the contents of PGDATA/pg_subtrans .

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
```

Subtransactions can be explicitly started either by using the SAVEPOINT command or by other means, such as the EXCEPTION clause of the PL/pgSQL language. That is, subtransactions are used quite actively.

The transaction ID of each subtransaction's immediate parent transaction is written to the pg_subtrans catalog . Transaction IDs of top-level transactions are not written because they do not have a parent transaction. Transaction IDs of read-only subtransactions are also not written.

subxid s are cached in shared memory for each backend by default . Once this limit is exceeded, disk I/O overhead increases significantly, as the subxid data has to be looked up in pg_subtrans . The subtrans_buffers parameter avoids this.

The VACUUM, CREATE/DROP DATABASE, CREATE /DROP TABLESPACE commands cannot be executed in a transaction because they implicitly generate transactions:

```
postgres=# begin;
BEGIN
postgres=# vacuum;
ERROR: VACUUM cannot run inside a transaction block
```

notify_buffers configuration parameter

- sets the size of shared memory used to cache the contents of PGDATA/pg_notify
- Default value is 8 blocks (64Kb)
- Used in the NOTIFY/LISTEN architecture for data exchange between processes:

```
postgres=# listen abc;
```

```
LISTEN
```

```
postgres=# notify abc;
```

```
NOTIFY
```

Asynchronous notification "abc" received from server process with PID 1234.



notify_buffers configuration parameter

notify_buffers configuration parameter specifies the size of the shared memory used to cache the contents of PGDATA/pg_notify .

Used in the NOTIFY/LISTEN architecture for data exchange between processes:

```
postgres=# listen abc;
```

```
LISTEN
```

```
postgres=# notifyabc;
```

```
NOTIFY
```

Asynchronous notification "abc" received from server process with PID 1284.

Setting parameters when creating a cluster

- environment variables and parameters of the `initdb` utility
- `initdb` parameters `--lc-collate`, `--lc-ctype`, `--encoding` cannot be changed after cluster creation
- The values of localization parameters can be viewed using the `psql \l` command
- The `initdb -k` or `--data-checksums` option specifies the calculation of checksums in data file pages.
- You can enable and disable checksum calculation after creating a cluster
- check status in `psql \dconfig data_checksums`
- Checksums of data blocks are verified using the `pg_checksums` and `pg_verifybackup` utilities.



Setting parameters when creating a cluster

The `initdb` cluster creation utility has parameters (keys) that set the properties of the cluster being created. `initdb` is also affected by environment variables set before the utility is launched. `initdb` parameters override the values set by environment variables. Some parameters cannot be changed after the cluster is created.

Some of the parameters specified when creating a cluster may change after it has been created.

`initdb` utility parameter `-k` or `--data-checksums` specifies the calculation of checksums in the blocks of data files located in tablespaces. In Tantor Postgres, checksum calculation is enabled by default if the cluster is created by the installation utility. If the `initdb` utility is started manually, it works as in PostgreSQL and the cluster is created without setting the checksum calculation.

You can enable, disable, or verify file checksums using the `pg_checksums` utility. To verify backups, use `pg_verifybackup`. You can find out whether checksums are enabled on a cluster using the `pg_controldata` utility or look at the value of the configuration parameter (read-only) `data_checksum`.

```
pg_controldata -D . | grep checksum
```

Data page checksum version: 0

Zero means disabled. A value other than zero means enabled.

It is not recommended to disable checksum verification. If a data block on the disk is damaged while accessing this block, processes, including cleaning processes, will not be able to continue working. This can lead to the impossibility of cleaning and freezing pages.

Why do I need to know these parameters? If during the transition to new major versions of the DBMS it is necessary to create a cluster, then it must be created with the same parameters as the one being updated.

https://docs.tantorlabs.ru/tdb/en/17_5/se/locale.html

Permissions for the PGDATA directory

- The `initdb -g` or `--allow-group-access` option sets permissions to 0750 (rwx rx ---)
- if you do not specify, then you do not need to recreate the cluster, you can change the permissions on PGDATA (and subdirectories) to acceptable values
- Valid values are 0750 and 0700 :
- **pg_ctl start**

```
waiting for server to start....
```

```
FATAL: data directory "/var/lib/postgresql/tantor/se16/data"
      has invalid permissions
```

```
DETAIL: Permissions should be u=rwx ( 0700 ) or u=rwx,g=rx (
      0750 ).
```

```
stopped waiting
```



Permissions for the PGDATA directory

The `initdb` cluster creation utility has parameters (keys) that set the properties of the cluster being created. `initdb` is also affected by environment variables set before the utility is launched. `initdb` parameters override the values set by environment variables. Some parameters cannot be changed after the cluster is created.

Some of the parameters specified when creating a cluster may change after it has been created.

`initdb` utility parameter `-k` or `--data-checksums` specifies the calculation of checksums in the blocks of data files located in tablespaces. In Tantor Postgres, checksum calculation is enabled by default if the cluster is created by the installation utility. If the `initdb` utility is started manually, it works as in PostgreSQL and the cluster is created without setting the checksum calculation.

You can enable, disable, or verify file checksums using the `pg_checksums` utility. To verify backups, use `pg_verifybackup`. You can find out whether checksums are enabled on a cluster using the `pg_controldata` utility or look at the value of the configuration parameter (read-only) `data_checksum`.

```
pg_controldata -D . | grep checksum
```

Data page checksum version: 0

Zero means disabled. A value other than zero means enabled.

It is not recommended to disable checksum verification. If a data block on the disk is damaged while accessing this block, processes, including cleaning processes, will not be able to continue working. This can lead to the impossibility of cleaning and freezing pages.

Why do I need to know these parameters? If during the transition to new major versions of the DBMS it is necessary to create a cluster, then it must be created with the same parameters as the one being updated.

https://docs.tantorlabs.ru/tdb/en/17_5/se/locale.html

PostgreSQL data block size

- size 8192 bytes = 8Kb set at compile time
- synonym for “page”
- when in memory, it occupies a buffer in the buffer cache



PostgreSQL data block size

By default, the page size (data block) is 8 kilobytes (or 8192 bytes). The data block size is set during compilation and in version 16 cannot be changed without recompiling the software. It is defined by the `BLCKSZ` macro , which is set by default to 8K (8192 bytes) in the file

`/opt/tantor/db/17/include/pg_config.h`

You can find out the block size

```
pg_controldata | grep 'block size'
```

Database block size : 8192

WAL block size: 8192

or the `block_size` configuration parameter

The data block size determines the limits for many PostgreSQL cluster characteristics.

https://docs.tantorlabs.ru/tdb/en/17_5/se/limits.html

PostgreSQL Limitations

database size	not limited
number of databases in the cluster	4,294,950,911
relations in one database	1,431,650,303
ratio size	32TB (8KB block)
blocks in the table	4,294,950,911
columns in the table	1600
columns in the selection (SELECT)	1664
field size (including text, bytea)	1GB
large object size (lo)	4TB
total lo volume in the database	32TB (8KB block)
identifier length	63 bytes
indexes per table	unlimited
string buffer size	1GB-1
columns in a composite index	32



PostgreSQL Limitations

The PostgreSQL data block size can be 16Kb, 32Kb. Currently, 8Kb is chosen empirically (by trial and error). It is determined by the current hardware development (for example, cache sizes). Internal operation algorithms, constants, parameters were chosen based on the 8Kb block size. When changing the block size, bottlenecks may appear under heavy load. Relations (synonym "class") are called : **tables, indexes, sequences, views, foreign tables, materialized views, composite types** . If the volume of data stored in table blocks exceeds 32 TB, it is worth using partitioned tables.

TOAST tables are also limited to 32TB, which **may limit** the number of rows in the main table.

Moreover, **the number of fields that can be extracted from row versions in TOAST is no more than 4 billion (2 to the power 32)** . This **may limit** the number of rows in the table.

The block size affects the maximum relation size. Large field values up to 1GB can be stored in `text`, `varchar`, `bytea` columns . This limitation follows from the fact that the maximum field size in a TOAST table is 1GB.

You can use the legacy `lo` data type . All values of this type in one database are stored in one system catalog table. Since the maximum size of a non-partitioned table is 32 TB, the maximum lo size in one database is also 32 TB. For example, one database can store no more than 8 fields of 4 TB each.

The number of columns on which an index can be created is limited by the `INDEX_MAX_KEYS` macro . The value of the constant is shown by the `max_index_keys` parameter .

There is also a limit on the number of function parameters equal to 100, but it can be increased to 600 (with a block size of 8 KB) by recompilation (macro `FUNC_MAX_ARGS` in `pg_config_manual.h`).

The maximum size of the string buffer (`MaxAllocSize` in `stringinfo.c`) is `0x3fffffff` = 1 Gigabyte minus 1 byte. When processing strings (`SELECT *` and `COPY` commands), memory is allocated for the string buffer. If the size of the processed data is larger and the buffer exceeds this limit when increasing its size again, an error is returned: "Cannot enlarge string buffer" . The Tantor Postgres configuration parameter `enable_large_allocations` and a similar parameter of the `pg_dump` utility can increase the size of the string buffer to 2 GB.

enable_large_allocations parameter

- Tantor Postgres DBMS parameter that increases the StringBuffer size 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



Parameter enable_large_allocations

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

`pg_dump` utility may refuse to dump such rows because it does not use the `WITH BINARY` option of the `COPY` command. For text fields, a non-printable character occupying one byte will be replaced by a sequence of printable characters occupying two bytes (for example, `\n`) and the text field may 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
```

и утилит командной строки:

```
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. The StringBuffer is allocated dynamically during the processing of each line, not when the server process starts. If there are no such lines, the parameter does not affect the operation of the server process.

This problem occurs with the row of the config table of the 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;
```

Limitations on the length of identifiers

- The maximum length of identifiers (e.g. table names, columns, indexes, etc.) is 63 characters.
- For example, you can create a table with a name containing up to 63 characters:
- `CREATE TABLE my_really_long_table_name_with_63_characters(...);`
- Or a column with a name that also contains up to 63 characters:
- `ALTER TABLE my_table_name ADD COLUMN my_really_long_column_name_with_63_characters INTEGER;`
- identifiers exceeding this size are truncated, about which a warning is issued



Limitations on the length of identifiers

The maximum length of identifiers (table names, column names, indexes, etc.) is 63 characters. This means that an identifier can contain up to 63 characters. This is a default limitation and applies to all identifiers in the database.

For example, you can create a table with a name containing up to 63 characters:

```
CREATE TABLE my_really_long_table_name_with_63_characters(...);
```

Or a column with a name that also contains up to 63 characters:

```
ALTER TABLE my_table_name ADD COLUMN  
my_really_long_column_name_with_63_characters INTEGER;
```

This limitation is in place to ensure compatibility with different systems and to make working with databases easier. If you need to use longer identifiers, you should rethink your database design.

=====

identifiers exceeding this size are truncated, about which a warning is issued

```
create table sixty-three characters 456789 (n numeric);
```

```
NOTICE: identifier "sixty-three characters 456789 " will be truncated to "sixty-three  
characters"
```

```
CREATE TABLE
```

```
\d sh*
```

```
Table "public.sixtythreecharacters"
```

```
Column | Type | Collation | Nullable | Default
```

```
-----+-----+-----+-----+-----+-----
```

```
n | numeric | | |
```

Identifiers include names of relations and columns. Identifiers can be enclosed in quotation marks. If the identifier length exceeds 63 bytes, it is truncated. Identifiers without quotation marks must begin with a letter.

The maximum length of an identifier is determined by the macro `NAMEDATALEN-1`, which is set at compile time. The value of the constant is shown by the parameter

```
postgres=# show max_identifier_length;
```

```
max_identifier_length
```

```
-----
```

```
63
```

There are other restrictions, for example, the maximum number of function arguments is 100, and the number of parameters in a query is 65535.

Configuration parameters

- installed during assembly and are not changed
- you can see:
- command line utility `pg_config`
- in the `pg_config` view
- `pg_config()` function
- `SHAREDIR` parameter specifies the directory with extension files.
- `PKGLIBDIR` points to the default shared library directory
- `BINDIR` specifies the directory containing executable files



Configuration parameters

"Configuration parameters" (config) and "configuration parameters" (settings) are similar, but they are different concepts.

Some parameters are set during assembly (compilation, linking). View most of the parameters set during assembly:

1) Command line utility `pg_config`

2) Function `select * from pg_config();`

For example, `SHAREDIR` defines a directory with extension files.

A question that may arise: I installed an extension and want to see what it includes. The easiest way is to find the extension text files and look at the commands and parameters in them. Where can I find these files? Answer: the extension control files are in the `extension` subdirectory of the `SHAREDIR` directory.

List of control extension files :

```
ls $(pg_config --sharedir)/extension/*.control
```

Second question: I loaded a shared library, where is its file? Or - I want to load a library, where should I copy its file? Answer: `PKGLIBDIR` points to the directory of shared libraries by default (files with the `.so` extension). Libraries can be loaded with the `LOAD` command in a session or with the `shared_preload_libraries` parameter.

```
pg_config --pkglibdir
```

```
/opt/tantor/db/17/lib/postgresql
```

`BINDIR` defines a directory with executable files, the path to which is added to the `PATH` environment variable of the postgres user (`.bash_profile` file) during the installation of Tantor Postgres.

```
cat .bash_profile
```

```
export PATH=$PATH
```

```
export PATH=/opt/tantor/db/17/bin:$PATH
```

`PGSYSCONFDIR` specifies the directory where the `pg_service.conf` connection services file is located.

If you create a service description in the services file, you can use it with `psql` "service=service_description"

In Oracle Database, the services file has an analogue in the form of the `tnsnames.ora` file. The `pg_service.conf` file is not in demand, it is not used by JDBC drivers, only by the `libpq` library.

https://docs.tantorlabs.ru/tdb/en/17_5/se/libpq-pgservice.html

Demonstration

- View configuration settings



Demonstration

View configuration settings

Practice

1. Overview of configuration parameters
2. Configuration parameters with unit of measurement
3. Configuration parameters of the logical type
4. Configuration parameters
5. Services file



Practice

Overview of configuration parameters

Configuration parameters with unit of measurement

Configuration parameters of the logical type

Configuration parameters

Services file



4

4a

Logical structure



Database cluster

- A cluster is a set or association of databases.
- The cluster has common objects for all databases.
- The application connects to one database and has access to its objects.
- The cluster is created using the initdb command line utility.
- The cluster is created in a physical storage location - a directory designated "PGDATA" by the name of the environment variable for utilities
- Databases in a cluster can be created and deleted.



Database cluster

The definition of the concept "database cluster" in the documentation is given as follows.

A database cluster (union) or "cluster" for short is a set of databases that have common global SQL objects and their common static and dynamic metadata. A database cluster is created by the initdb command line utility. A SQL object is any object that can be created by the CREATE SQL command. Global SQL objects are roles, tablespaces, replication sources, logical replication subscriptions, and databases. Local SQL objects are those that are not global. Databases are a named set of local SQL objects.

The definition is neat, but seems unclear, if only because "general static" and especially "dynamic metadata" are not given in the list of definitions.

If you have worked with Oracle Database, the PostgreSQL database analog is Pluggable Database (PDB). The cluster analog is a multitenant container database. There is no root database (CDB Root) in PostgreSQL, you can connect to any of the databases to manage the PostgreSQL cluster. The Oracle Seed PDB analog is the template0 or template1 database.

Let's consider the concept of "database" from another angle. An application needs to store data, it stores it in the form of tables and other objects. To store it, the application creates connections to a storage location, which can be called a database. A place where objects can be located (stored, located) together, with which it is possible to work simultaneously (for example, connect tables in one selection), is a logical storage location for the application - a "database".

To create a storage location, you need to run the initdb utility. This utility will create a set of files and directories in the physical storage location - a directory whose path is specified in the initdb parameters. This directory is called PGDATA. PGDATA stores a cluster of databases. In order for applications to be able to connect to any database (there is no concept of connecting to a cluster, one connection, or session, connects to one database), an "instance" must be running on the host (aka server or computer) - a set of server, background (auxiliary) processes and the main postgres process (aka postmaster). Initially, three databases are created, later, after starting the cluster, you can create a database in the cluster using the CREATE DATABASE command. You can connect to any database in the cluster, the database will be created and will be equal among all other databases in the cluster.

Oracle Database has a similar technology called Real Application Cluster (RAC) - a set of one or more instances serving CDB or non-CDB. RAC is not an analogue of the PostgreSQL "database cluster" concept. In PostgreSQL, one cluster serves one instance.

https://docs.tantorlabs.ru/tdb/en/17_5/se/glossary.html

Instance

- An instance is a set of processes and the memory they use that serve a single database cluster.
- The database cluster is served by only one instance.
- The first and main process of an instance is called postgres or postmaster
- A required instance parameter is the port on which the postmaster listens for incoming TCP and local connections.
- By default, the port configuration parameter is set to 5432



Instance

An instance of a database cluster is a set of processes and the memory they use (shared and local for each of these processes) through which applications connect (create sessions) to databases. Cluster databases, since one instance serves exactly one cluster. In a cluster, you can create and delete databases. An instance is the same as a single-instance Oracle Database instance.

Let's go into detail: an instance is one postgres process (postmaster), a set of server (servicing, backend, foreground), auxiliary (background) processes that use shared memory to exchange data with each other and achieve synergy by sharing memory structures that are located in a common memory area. Several DBMS instances can operate on one server, if there is no conflict in the port number, including in the Unix socket file name.

What is "port"? It is a number that is set in the port configuration parameter. By default, it is 5432. The port is used in the Unix socket name (file) and as the TCP port number of network interfaces (IP addresses) that are listed in the local_addresses parameter. The default value is localhost. * - all IPV4 and IPV6 addresses, '0.0.0.0' - all IPV4 addresses, ':::' - all IPV6 addresses. But you can specify a list of names and/or numeric IP addresses of nodes, separated by commas. An empty string means that the instance can only be connected to via a Unix socket.

The postgres instance process listens on this port. In Oracle Database, this is done by non-instance listener processes.

An instance implements all the functional capabilities of the DBMS through its processes: reads and writes files, works with shared memory, provides ACID transaction properties, accepts connections from client processes, checks access rights, performs recovery after a failure, performs replication, and other tasks.

An application connects via a socket to its server process. Background processes are not connected to applications and perform common useful work.

Note: The name postmaster is used to refer to the main process of the instance, since the word PostgreSQL can refer to many things, such as the family of DBMSs to which Tantor Postgres belongs. Tantor Postgres is a fork of the freely distributed PostgreSQL, as are other forks: Enterprise DB, Postgres Pro Enterprise.

Database

- A database is a logical storage location for SQL objects.
- The database is part of a cluster
- The contents of the database can be dumped at the logical level (as SQL commands) and loaded into another database in the same or another cluster.
- The application creates a session with one database
- The session has access to objects of one database
- There is no access to objects of other cluster databases in one session.



Database

The application stores data in the DBMS and accesses it through a connection to the server process of the instance. Within the connection (local via Unix socket or network TCP socket), a session is created. Session, connection, session, connection are often (in the documentation sometimes) used as synonyms, because it is important for the application to issue SQL commands and receive a result. The differences between connections and sessions play a role when configuring load balancers (for example, the pgBouncer application) and network settings. Connection is a physical concept, session is a logical one.

Once a connection is created, the application must have access to all of its objects. For example, it must be able to join selections from multiple tables and use its stored functions. Therefore, all (with a few exceptions) storage objects that the application uses are local to the database, stored in it.

The connection is established only with one cluster database. Data stored in different cluster databases is isolated from each other based on the fact that it is usually intended for use by different applications, and applications should not intersect with each other, including from the point of view of access control.

The idea of isolating applications using databases and combining application objects in one database can be technically bypassed, since the needs of applications are different. For example, using extensions (types `fdw`, `dblink`), an application in its session can work with data in several databases. And several applications using schemas and roles can store tables with the same names in one database without interfering with each other.

List of databases

- To get a list, use the psql command:
`\l` or `\l+`

```
postgres=# \l
                                List of databases
   Name   | Owner   | Encoding | Locale Provider | Collate  | Ctype    |
-----+-----+-----+-----+-----+-----+
 postgres | postgres | UTF8     | libc             | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | libc             | en_US.UTF-8 | en_US.UTF-8 |
 template1 | postgres | UTF8     | libc             | en_US.UTF-8 | en_US.UTF-8 |
 testdb   | postgres | UTF8     | libc             | en_US.UTF-8 | en_US.UTF-8 |
(4 rows)

postgres=# SELECT datname FROM pg_database;
datname
-----
 postgres
 template1
 template0
 testdb
```



List of databases

Initially, after creating a cluster, there are three databases named `postgres`, `template0`, `template1`. You cannot connect to the `template0` database, it is not intended for making changes to it. The list of databases can be obtained:

psql commands `\l` or `\l+`

command **`SELECT datname FROM pg_database;`**

or graphical utilities such as DBeaver, Tantor Platform.

Creating a database

- CREATE DATABASE command or createdb utility
- two modes of database creation: WAL_LOG and FILE_COPY
- creating a database is cloning another database with all its contents
- the database has an owner role (OWNER)
- the owner and name of the database can be changed after its creation
- you can create a database with your own localization parameters (encoding)
- encoding cannot be changed after the database has been created
- the "template" database property (IS_TEMPLATE) affects the ability to clone it by an unprivileged role and delete it



Creating a database

A database can be created by a role with the `SUPERUSER` or `CREATEDB` attribute :

CREATE DATABASE database_name parameter = value parameter = value;

The command has a wrapper utility `createdb`, it is convenient if you want to create databases from the command line.

There are about 15 parameters in the command. You can pay attention to the following parameters:

`OWNER` can be used to specify the name of a role that will have privileges similar to a superuser within this database.

To make a role owner, be a superuser or log into that role directly or indirectly. By default, the creator becomes the owner of the database.

`TEMPLATE` - the name of the database whose contents you will copy. This is any database, not necessarily having the `IS_TEMPLATE` property . By default, `template1` is used. But if you want to create a database with localization parameters different from those specified for `template1`, you need to use `template0` (unmodifiable empty database).

`IS_TEMPLATE` can be changed after the database is created If `IS_TEMPLATE= true` , this database can be cloned by any user with the `CREATEDB` attribute ; otherwise (by default), only superusers and its owner can clone this database. Also, a database with a template property cannot be deleted. To delete it, you must first remove the template property.

The encoding and classification of characters are related to the collation type. To create a database with an encoding different from the encoding with which the cluster was created, you may need to specify four parameters:

```
create database database_name LC_COLLATE = 'ru_RU.iso88595'  
LC_CTYPE='ru_RU.iso88595' ENCODING='ISO_8859_5' TEMPLATE= template0;
```

The collation types available for use can be viewed in the `pg_collation` table. The collation types "C" and "POSIX" are compatible with all encodings. They should not be used, since the sorting order of Cyrillic characters does not comply with linguistic rules.

The localization parameters available for use are determined at the time of cluster creation, are stored in this table, and do not change after the cluster is created.

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createdatabase.html

Changing database properties

- The `ALTER DATABASE` name command can be used to change some properties of an existing database.
- Some properties can be changed only if there is no session to the database
- The `ALTER DATABASE` name `SET` command can be used to change the configuration parameters of sessions created with this database.
- At the database level, you can set about 190 configuration parameters
- The database is a common object of the cluster and its properties can be changed by connecting to any database in the cluster.



Changing database properties

`STRATEGY` - pay attention to this parameter if you are creating a database on an industrial cluster or the database that you are using as a template (which you are cloning) is large. The parameter appeared in version 14 of PostgreSQL and immediately by default began to use the new `WAL_LOG` strategy , which compiles a list of objects and the entire volume of the cloned database passes through the write-ahead logs. The reason for the new strategy is that the previous `FILE_COPY` performed a checkpoint, then copied directories (logging only copy commands), then a second checkpoint. If the template is small, then the first checkpoint on an industrial cluster gives an increased load (the second is insignificant). The reason is not only in the one-time and indirect increase in the recording volume (a dirty block is written by a checkpoint but changes and the second one will be written less often, but it could be written once if there was no checkpoint) the load on the input-output (storage system, disks), but in the fact that after each checkpoint each changeable block is written to the journal in full (8 KB) since by default the parameter `full_page_writes=on` (and it is unsafe to disable it). But if the size of the template database is greater than the value of the `max_wal_size` parameter , then the checkpoint will also be performed and can even be repeated if the size of the database is several times greater than the value of the parameter.

If the template size is small, for example, less than 16MB (the WAL segment size) or several times larger, then you can create a database. If it is larger, then you should choose a time when the cluster is least loaded. If there are replicas, then evaluate the network throughput and, possibly, specify the `FILE_COPY` strategy . If the template size is larger, for example, than half of `max_wal_size` , then `FILE_COPY` is preferable.

You can give a description of the created database. Descriptions for almost any objects can be given with the command:

```
comment on database db1 is 'Database for my purpose';
```

The description can be viewed using the command `\l+`

Descriptions do not affect functionality.

Database-level configuration settings (`ALTER DATABASE`) and database-level permissions (`GRANT`) from the template database are not carried over to the clone.

https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-alterdatabase.html

ALTER DATABASE command

- The command syntax shows which properties can be changed:
- ```
ALTER DATABASE database_name
ALLOW_CONNECTIONS true|false
CONNECTION LIMIT -1/number
IS_TEMPLATE true|false
RENAME TO new
OWNER TO new
SET TABLESPACE new
REFRESH COLLATION VERSION
SET parameter=value
```



## ALTER DATABASE command

You can change the properties of a database using the `ALTER DATABASE` command . Example:

```
alter database name is_template=true;
```

**alter database name SET name=value;** changes one of about 190 session parameters that can be set at the database level.

`CREATEDB` attribute or the superuser can rename the database . You cannot rename the database you are connected to; you must connect to any other database.

You can change the default tablespace, but no one must be connected to the database and all files (except those in other tablespaces) and system catalog object files will be moved at the file system level.

You can change the owner of the database.

You can set configuration parameters to customize the behavior of processes (both background and session serving) that work with objects in this database.

Localization parameters can be selected when creating a database, after the database is created they cannot be changed. The main ones are encoding and collation values ( sorting rules), `ctype` (character classification) which are related to the encoding value, localization provider. Some localization parameters are session-specific and can be changed using the `ALTER DATABASE SET` command .

Database creation and tablespace alteration are non-transactional (cannot be performed within a transaction). Knowing whether commands are transactional is useful, for example, when installing or developing extensions. Non-transactional commands cannot be executed when installing an extension to a database.



# Deleting a database

- Performed by the `DROP DATABASE` command or the `dropdb` wrapper utility
- To execute the command, you need to connect to any cluster database other than the one being deleted.
- The database is deleted with all its contents.
- The command is non-transactional - it cannot be rolled back.
- Files that store database object data are physically deleted



## Deleting a database

If the contents of the database are not needed, the database can be deleted.

When deleting, local objects in other databases are not affected. The delete command is:

```
DROP DATABASE [IF EXISTS] name;
```

Optional keywords in square brackets.

`IF EXISTS` is present in many commands and allows not to generate an error (severity level `ERROR` ) if the object does not exist, but usually reports (severity level `NOTICE` ) that such an object does not exist. Severity levels affect how the message is processed: issued to the client, transferred to the cluster message log.

Next command:

```
DROP DATABASE name (FORCE) ;
```

allows you to detach sessions that are connected to this database, interrupt their transactions and delete the database.

A database with the `IS_TEMPLATE` property can be dropped by removing the template property.

There is no need to delete the `template0` database .

# Schemas in the database

- schema synonym - namespace
- are used to organize the storage of database objects
- Schema is a local database object.
- Schemas allow you to have multiple tables and other types of objects with the same name in the same database.
- An object cannot be in multiple schemas.
- Schemes can be used for logic of combining (packaging) subroutines
- There are no "package" objects in PostgreSQL



## Schemas in the database

A synonym for schema is namespace.

Schemes are used to organize the storage of database objects. Analogy: files in the file system can be located in different directories. Similarly, tables, views, subroutines can be located in different schemes of the same database.

A schema is a **local** database object, i.e. each database in the cluster has its own set of schemas. Schemas with the same names (identifiers) may exist in different databases.

Schemas allow you to have multiple tables and other types of objects with the same name in the same database.

Schemes allow you to combine subroutines (procedures and functions) that are logically related to each other.

Most objects that applications work with must belong to a single schema. Such objects cannot exist without a schema. Before deleting a schema, you will need to reassign objects to another schema. An object cannot be in more than one schema at a time. There are no symbolic or hard links, as in the file system.

When referring to such objects, you can specify a scheme and a dot symbol before the object name. For example:

```
SELECT schema.function() ; or SELECT * FROM schema.table;
```

Oracle Database has package and package body objects. PostgreSQL does not have such objects. Schemas can be used to provide the main functionality of packages - the ability to combine logically similar subroutines into modules (packages). Using extensions that implement packages by adding create package commands leads to the creation of non-portable (to other PostgreSQL family DBMS) code.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/ddl-schemas.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/ddl-schemas.html)

# Creating and modifying schemes

- Schemes are not associated with roles, but have an owner.
- When creating a schema, you can assign an owner role:  
`CREATE SCHEMA schema_name AUTHORIZATION owner;`
- You can change the owner:  
`ALTER SCHEMA schema_name OWNER TO role;`
- You can rename the scheme:  
`ALTER SCHEMA schema_name RENAME TO name;`
- You can grant CREATE and USAGE privileges to roles on schemas.



## Creating and modifying schemes

Schemas are not associated with roles. The owner name of an object and the schema name (in which the object resides) can be different and can be changed after the object is created.

Schemes have an owner. When creating a scheme, you can set it:

```
CREATE SCHEMA schema_name AUTHORIZATION owner;
```

and later change:

```
ALTER SCHEMA schema_name OWNER TO role;
```

You can rename the schema, but you should remember about the search path, the value of which will probably need to reflect the new schema name.

In Oracle Database, schemas and users are linked to each other, which limits flexibility. Oracle Database has "synonym" objects for this reason, PostgreSQL does not have "synonym" objects because they are not needed.

You can grant CREATE and/or USAGE privileges to roles on schemas. This allows you to control the "visibility" of objects in a schema as a whole. Analogy: a file system may have a privilege to access a file, but if there is no privilege to the directory in which the file is located, then there will be no access to the file.

Schemes can be deleted:

```
DROP SCHEMA [IF EXISTS] name [CASCADE] ;
```

If there are objects in the scheme, the scheme will not be deleted by default. If the objects are needed, they should be moved to another scheme. If the objects are not needed, they can be deleted together with the scheme using the CASCADE option .

# Path for searching objects in schemes

- specifies a list of schemas in which to search for an object
- `search_path` parameter , which can be changed at any level
- default value "`$user`", `public`



## Path for searching objects in schemes

Associated with schema objects is the concept of a search path and the corresponding configuration parameter `search_path` . This parameter is set at the cluster level and has a default value of "`$user`", `public`

`$user` - the name of the role in which the session is currently running is substituted.

This parameter can be set at any level and changed by any role.

There is an analogue in file systems - the `PATH` environment variable.

The search path can specify several schemas in which the object will be searched, if the scheme name is not explicitly specified before the object name. The object is searched in the order of the schemes listed in the search path. If the scheme does not exist or there are no rights to it, then the search is performed in the following schemes in order and no errors are returned. The search algorithm is similar to searching for files in the file system.

The template databases have a schema named `public`, so when creating any databases, a schema named `public` will exist. The `public` schema is specified in the search path: "`$user`", `public`

The logic for using the search path is usually chosen in advance and the value of the `search_path` parameter at the cluster or database level is not subsequently changed, because changing the search path may result in objects no longer being found in routines.

The default value allows you to create schemas with the same name as the role names, which is convenient. It should be remembered that a schema is a local object, and a role is common to the entire cluster. If a role has the right to connect to several databases in a cluster, then in each of them you can create a schema with the same name.

# Special schemes

- `pg_catalog` - this schema contains the "system catalog" objects
- `information_schema` - schema described in the SQL standard
- `pg_toast` - schema for TOAST special tables
- `pg_temp` (reference to `pg_tempN` where N is a number) - schema for temporary tables
- `pg_toast_temp` (reference to `pg_toast_tempN` where N is a number) - schema for temporary TOAST tables to temporary tables



## Special schemes

PostgreSQL has utility schemas:

`pg_catalog` - this schema contains "system catalog" objects - service tables, views, functions and other objects

`information_schema` - a schema described in the SQL standard. Contains tables with standardized names and column titles. The developers of the standard believed that DBMS manufacturers would create this schema and tables, which would allow developers to obtain data with the same `SELECT` command when working with DBMS from different manufacturers. This idea was not widely used, since information from standardized tables is not in great demand during development, and also because the specifications of JDBC access interfaces contain methods that allow much more useful information about the DBMS and objects in it to be obtained in a standard way, regardless of the DBMS used.

There are schemas for specific types of tables, which are defined based on the principle that tables must have a schema (tables must be located in some schema):

`pg_toast` is a schema for special TOAST tables, which are used to store large fields. These tables are hidden so that they do not create "information noise". For this purpose, TOAST tables (and their indexes) are created in this service schema. You can know about this schema in case its name is encountered somewhere. Working with TOAST is fully automated and there are no separate commands for working with TOAST objects and the schema. To change the properties related to TOAST, the options of the `CREATE TABLE` and `ALTER TABLE` commands for ordinary tables are used.

`pg_toast_temp` (reference to `pg_toast_tempN` where N is a number) - schema for temporary TOAST tables (and indexes) to temporary tables. Exists no longer than the session lifetime.

`pg_temp` (reference to `pg_tempN` where N is a number) - schema for temporary tables. Temporary tables, indexes, views (their definitions and data) exist either until the end of the transaction or until the end of the session. Implicitly present at the beginning of the search path.

It makes practical sense to know about the `pg_catalog` schema. The name of this schema can be used in `psql` commands to find service tables, views, and functions.

Knowledge of temporary objects is necessary for developers and administrators if they are faced with a large number of temporary object files. Using the Tantor SE1C assembly allows you to reduce problems when working with a large number of temporary objects.

# Determining the current search path

- `psql` command `SHOW search_path;`
  - › returns the search path set for this location
- function `current_schemas(false)`
  - › returns the search path currently in effect at that location as an array
- function `current_schemas(true)`
  - › adds utility schemas, namely `pg_catalog` and `pg_temp_N`
- `current_schema` function or `current_schema()`
  - › returns a single name of the first schema in the search path, or `NULL` if the search path is empty



## Determining the current search path

The current search path can be obtained:

`psql` command `SHOW search_path;` returns the search path set for this location as a string. Comma-separated.

`current_schemas(false)` function - returns the search path currently in effect in this location as an array. However, unlike `search_path`, it does not return non-existent schemes, only specific names of existing schemes. The function is convenient to use in stored routines.

`current_schemas(true)` - adds service schemas, namely `pg_catalog` and `pg_temp_N` (if it was automatically created in the session) if it is implicitly present in the search path. Schemas for TOAST are not given, this is by design. This variant of the function is used to determine whether the object name will be searched first in the system catalog schema. For example, a function or table whose name begins with "`pg_`" (this is how the names of all system catalog objects begin) is searched, user objects according to the convention (code conventions) which application developers usually adhere to should not have names starting with "`pg_`". It is possible to change the search path so that `pg_catalog` is not the first in the list, but this does not make sense and is not practiced.

`current_schema` or `current_schema()` function. In PostgreSQL, parentheses "`()`" are required after the name of a function without arguments. However, for some functions described in the SQL standard to which this function belongs, they are not required, because parentheses are optional in the SQL standard. This function returns a single name of the first schema in the search path (`search_path`) or `NULL` if the search path is empty. User objects will be created in this schema unless the schema name is explicitly specified in the create command. If the function returns `NULL`, the object will not be created without specifying a schema.

# In which scheme will the object be created?

- the name of this schema for regular objects is given by the `current_schema()` function
- temporary objects are created in service schemas
- Service schemas for temporary objects are created automatically



## In which scheme will the object be created?

To determine the schema in which the object will be created, the search path that is valid at the given location of the command execution is used. The name of this schema for ordinary objects is given by the `current_schema()` function. However, if the object is "unusual" (temporary), then the schemas that can contain objects of this specific type are used. This applies to temporary tables, indexes on temporary tables, temporary views, TOAST tables to temporary tables. In this case, if the schema is missing, it will be created (or assigned from previously created ones and not used by other sessions) - it will be assigned a number and it will be used as a suffix in the schema name. In this case, the name of such a service schema will implicitly exist in the search path. Accordingly, such objects will be implicitly searched for and there is no need to prefix their names with the name of the service schema.

Thus, creating a temporary table results in adding rows to the system catalog tables. With massive generation of temporary objects, the system catalog tables and indexes, as well as the file system, can become a bottleneck. After the session ends, the temporary schema objects are deleted, and the schema itself is left for reuse by other sessions, so as not to cause frequent deletion of rows in the `pg_namespace` system catalog table.

If you need to explicitly specify a place in the search path for service schemas, you can specify the names `pg_catalog` and `pg_temp` in the desired order among the regular schemas. This order will be used. However, it is better not to allow overlapping object names and make the names unique, so that you do not have to resort to changing the search path.

# Search path in SECURITY DEFINER routines

- when using `$user` in the search path, the name of the subroutine owner will be substituted in the body of the subroutine
- before calling such a subroutine, the caller can change the value in the search path and remove `$user`, in this case the value set by the caller will be used in the body of the subroutine and the subroutine can start working with other objects
- At any (INVOKER and DEFINER) routine level, you can set values for configuration parameters that can be changed at the session level.



## Search path in SECURITY DEFINER routines

In routines with the `SECURITY DEFINER` property, there is a peculiarity with the search path. For example, with the substitution variable `$user`. In the body of routines (procedures and functions), the owner's rights (`DEFINER`) are used. The user function in such routines returns the owner's name. In the search path with the substitution variable, the owner's name will be. Since `$user` is present in the default value, usually the creator of such a routine tests the routine with this `search_path` value. `search_path` in his session or transaction before calling the subroutine, then this is the value that will be in effect in the body of the subroutine. The visibility of objects may change.

To avoid dependence on such a change of the `search_path` parameter, it can be set forcibly in the subroutine properties:

```
CREATE FUNCTION name(parameters)
RETURNS type
LANGUAGE language
SET search_path TO 'value'
SECURITY {DEFINER | INVOKER}
AS
BEGIN
END;
```

If you put `SET` inside the `BEGIN` and `END` block, there will be no error, but the behavior will be different - the set value will remain after exiting the subroutine, and if a transaction is rolled back (even implicitly if there is an `EXCEPTION` section in the subroutine), the change in the parameter value will be canceled. This creates ambiguity and generates difficult to detect errors.

At the level of any (`INVOKER` and `DEFINER`) subroutine, you can set a value for a configuration parameter that allows changing the value at the session level (user, superuser context).

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-createfunction.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createfunction.html)



# Masking schema objects

- If commands do not use a schema name before the object name, adding a schema to the search path that has an object with the same name masks the original object.
- By default, the temporary schema, the system catalog schema are implicitly present at the beginning of the search path and creating a temporary object masks any table, view, sequence

```
postgres=# \dt pg_authid
 List of relations
 Schema | Name | Type | Owner
-----+-----+-----+-----
 pg_catalog | pg_authid | table | postgres
(1 row)

postgres=# create temporary table pg_authid(t text);
CREATE TABLE
postgres=# select * from pg_authid;
 t

(0 rows)

postgres=# select current_schemas(true);
 current_schemas

 {pg_temp_4,pg_catalog,public}
(1 row)
```



## Masking schema objects

The documentation says: " For security, `search_path` should be configured to exclude any schemas that are writable by untrusted users. This prevents malicious users from creating objects (such as tables, functions, and operators) that could **mask** objects intended for use by the function. Particularly important in this regard is the temporary schema, which is searched first by default and is usually writable by anyone. A safe location can be achieved by forcing the temporary schema to be searched last. To do this, write `pg_temp` as the last element in `search_path` . "

In other words, for a routine with the `DEFINER` tag to be safe, `search_path` must:

- 1) be set at the subroutine definition level
- 2) exclude any schemes that can be created or modified by users with a lower level of privilege than the owner of such a routine
- 3) The `pg_temp` schema must be specified explicitly at the end of the search path.

Also, you should know that by default, after creating a subroutine, the `PUBLIC` role gets the right to execute the subroutine. This behavior can be changed using `default privileges` .

System catalog objects, including functions, can be masked by explicitly specifying the `pg_catalog` schema in the search path after the schema with the masking object. For example: `SET search_path = public, pg_catalog;`

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-createfunction.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createfunction.html)

# System catalog

- The system catalog is tables, views, functions, indexes and other objects that are used to store metadata.
- System catalog objects are located in the `pg_catalog` schema.
- System catalog objects (except global ones) are always located in the default tablespace for the database.
- object names are converted to lower case
- System catalog tables implicitly use cluster processes during SQL command execution



## System catalog

The system catalog contains tables, views, functions, indexes (on a column named `oid` that is in every table of the system catalog), and other objects that are used to store metadata (data about data) and for service purposes. When a table or other object is created, rows are inserted into the system catalog tables and files are created in the file system to store the table rows. The system catalog tables implicitly use cluster processes at the stage of executing SQL commands, for example, to check for the existence of tables, the presence of privileges, and the names of files in which to search for rows.

For example, the command to create a database, in addition to a large number of actions, inserts a row into the `pg_database` table .

System catalog objects are located in the `pg_catalog` schema .

In Oracle Database, the equivalent of a system catalog is called a "data dictionary."

Object names are converted to lower case and stored in lower case (unless double quotes were used when specifying names).

System catalog objects (except global ones) are always located in the default tablespace for the database.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/catalogs.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/catalogs.html)

# Common Cluster Objects

- These are objects (tables and auxiliary objects) of the system catalog that store data about common objects of the cluster
- `pg_global` tablespace
- Common cluster objects:
  - databases
  - roles
  - tablespaces
  - logical replication subscriptions
  - logical replication sources



## Common Cluster Objects

The cluster also has global objects, information about them is stored in several tables (global cluster objects), which are located in the `pg_global` tablespace . These tables are visible in the same way in sessions connected to any database in the cluster. Global objects include 11 tables and 21 indexes on their columns, 9 TOAST tables and 9 indexes on TOAST tables. A total of 50 objects.

Common cluster objects:

1)databases - table `pg_database`

2)roles - `pg_authid`, `pg_auth_members`

3)tablespaces `pg_tablespace`

4)`pg_subscription` logical replication subscriptions

5)logical replication sources - `pg_replication_origin`, `pg_shseclabel`, `pg_shdepend`, `pg_shdescription`

The global catalog also stores privileges to change the values of the `pg_parameter_acl` parameters and the configuration parameters of roles in sessions with specific databases `pg_db_role_setting` .

Roles, tablespaces, replication sources, logical replication subscriptions, and databases themselves are not local SQL objects because they exist outside of any single database; they are called global objects. The names of such objects must be unique across the entire database cluster.

# Using the system catalog

- information can be retrieved from system catalog tables and views using SELECT, WITH, psql commands (start with \), graphical applications, and the Tantor Platform
- \? psql command help
- It is not advisable to make changes to the system catalog tables unless this is described in the documentation.
- Knowing the PostgreSQL architecture, concepts, terms, you can easily get information using psql commands



## Using the system catalog

Changes to system catalog tables are made during the execution of DDL commands. System catalog tables are not locked against changes. It is not advisable to make changes to system catalog tables directly using SQL commands unless this is documented. It is possible to select data from system catalog tables directly using SELECT and WITH commands and it is used in application code and in cluster administration. However, the structure of system catalog tables is not very convenient for human reading. The structure was created many years ago, when the sizes of storage systems were small, as was the memory on computers. For convenience of working with the system catalog, there are views that are convenient to use.

You can get a list of system catalog views using the psql `\dv S` command.

`S` suffix at the end of psql commands allows you to list the contents of the system catalog, which is not normally listed by default.

It is more practical to work with the system catalog using psql commands. The `\? command` gives a list of all psql commands. It will also give help on the `\? command` itself :

Reference

```
\? [commands] help on psql commands (i.e. those starting with \)
```

```
\? options help on command line options of the psql utility
```

```
\? variables help on variables that change psql behavior
```

```
\h [NAME] help on SQL command; * - on all commands
```

Knowing the PostgreSQL architecture, concepts, terms, you can easily obtain information using psql commands.

# Accessing the system directory

- Executed by SELECT and WITH commands
- you can join tables and views
- column named oid is the primary key
- the oid column is used for relationships between tables
- The maximum number of rows in the system catalog tables is 4 billion
- the first three characters in the column names are the abbreviated table name



## Accessing the system directory

You can access tables and views of the system catalog with the SELECT command. The names of tables and views can be obtained with the `psql` command `\dtvs pg_*term*`

By the name of the table or view, understand which table or view contains the necessary information. Then use the `\d` name command to get the column names. The first three characters in the name of the system catalog table columns traditionally contain a letter combination similar to the name of the table where this column was created. For example, in `pg_name space` the prefix is "`nsp`".

**Starting with the fourth character**, there is usually an English word or its understandable abbreviation.

If comments have been created for a table or columns, you can view them by adding "`+`" to the command `\d+ object_name`. Unfortunately, descriptions for system catalog tables are not specified. Descriptions can be found in the documentation.

In the system catalog tables, the first column is called `oid` and its type is `oid`. Let's look at the type description using the `\dT oid` command.

List of data types

| Scheme | Name | Description |
|--------|------|-------------|
|--------|------|-------------|

|            |     |                                                         |
|------------|-----|---------------------------------------------------------|
| pg_catalog | oid | object identifier(oid), maximum 4 billion<br>(1 строка) |
|------------|-----|---------------------------------------------------------|

This type has a description stating that the maximum number of values is 4 billion. This means that the system catalog table can have no more than 4 billion rows. This means that if there is a table for storing types (`pg_class`), then there can be no more than 4 billion types in one database. Also, no more than 4 billion relations in one database. An index has been created on the `oid` column of the system catalog tables, and the column itself is the primary key. If the number of rows in the system catalog table reaches 4 billion, the instance and its processes will continue to work. Values are entered into the **oid column** by auto-increment. Upon reaching 4 billion, server processes servicing commands that need to insert a new row into any system catalog table will search for an unused value (these can accumulate; **oid values** are released after deleting an object) in the `oid` column, which will slow down the execution of commands. You should not create billions of objects and then delete billions of them. It is also important to remember that vacuuming and freezing also works for system catalog tables.

# reg-types

- created for 11 tables of the system catalog
- are used to cast a text object name to an oid type value and back
- make it easier to write queries to system catalog tables by reducing the number of table joins
- Example: find the name of the TOAST table created for the pg\_tablespace table:

```
SELECT reltoastrelid, reltoastrelid::regclass
FROM pg_class where relname='pg_tablespace';
reltoastrelid | reltoastrelid
-----+-----
4185 | pg_toast.pg_toast_1213
```



## reg-types

To get data from the system catalog tables, you may need to join several tables. The rows of the system catalog tables are linked via the oid column, that is, a number. In PostgreSQL, you can create data types ( `CREATE TYPE` ) and cast types ( `CREATE CAST` ). This is also used by PostgreSQL developers. 11 data types and type casts were created that allow you to easily convert the oid (number) in a column of one of the 11 system catalog tables to the name of an object in this table and back. These types are called reg types. Using reg types and type casts allows you to write queries to the system catalog tables without using joins (JOIN), thereby simplifying the selection command. psql, when servicing its commands starting with "\", forms a SELECT command to the system catalog tables and sometimes uses type casts. Such SELECT can be viewed by setting the \pset ECHO\_HIDDEN on variable

The list of reg types can be viewed using the command `\dT reg*`

List of data types

| Scheme     | Name          | Description                          |
|------------|---------------|--------------------------------------|
| pg_catalog | regclass      | registered class                     |
| pg_catalog | regcollation  | registered collation                 |
| pg_catalog | regconfig     | registered text search configuration |
| pg_catalog | regdictionary | registered text search dictionary    |
| pg_catalog | regnamespace  | registered namespace                 |
| pg_catalog | regoper       | registered operator                  |
| pg_catalog | regoperator   | registered operator (with args)      |
| pg_catalog | regproc       | registered procedure                 |
| pg_catalog | regprocedure  | registered procedure (with args)     |
| pg_catalog | regrole       | registered role                      |
| pg_catalog | regtype       | registered type                      |

(11 lines)

Example: `SELECT relname, reltoastrelid::regclass FROM pg_class WHERE reltoastrelid>0 AND relnamespace='pg_catalog':text::regnamespace order by 1;` will return the TOAST table names of the 36 system catalog tables that have them.

# Frequently used psql commands

- `\l` - list of databases
- `\du \dg` - list of cluster roles
- `\dn` - list of database schemes
- `\db` - list of tablespaces
- `\dconfig *name*` - list of configuration parameters
- `\dfS pg*` - a list of system functions and procedures useful for administration. Some information about the operation of an instance and cluster can only be obtained using functions. Some service views use functions
- `\dvS pg*` - useful utility representations
- + symbol at the end of the command shows more information  
example: `\db+` will show the size and privileges



## Frequently used psql commands

`\l` - list ( l ist) of databases

`\d u` or `\d g` - list of roles ( u ser, g roup) of the cluster, `\drg` - assignments of roles to roles

`\dn` - list of database schemes ( namespace )

`\db` - list of tablespaces

`\d config *name*` - list of cluster configuration parameters ( config )

`\ddp` - a list of default privileges . This is a special type of **privilege** or revokable privilege specific to PostgreSQL.

`\d f S pg*` - a list of system functions and procedures useful for administration. Some information about instance and cluster operation can only be obtained using functions. Some service views use functions. Procedures were introduced to PostgreSQL later than functions, so "f" is used for procedures as well.

`\d vS pg*` - useful system ( S ystem) representations ( view )

`\d x` - list of installed extensions ( extention )

`\dy` - list of triggers for events that are usually created by extensions or administrators

When entering a command in psql, remember that you can press the tab key on your keyboard twice and psql will display a list of possible values that you can enter next:

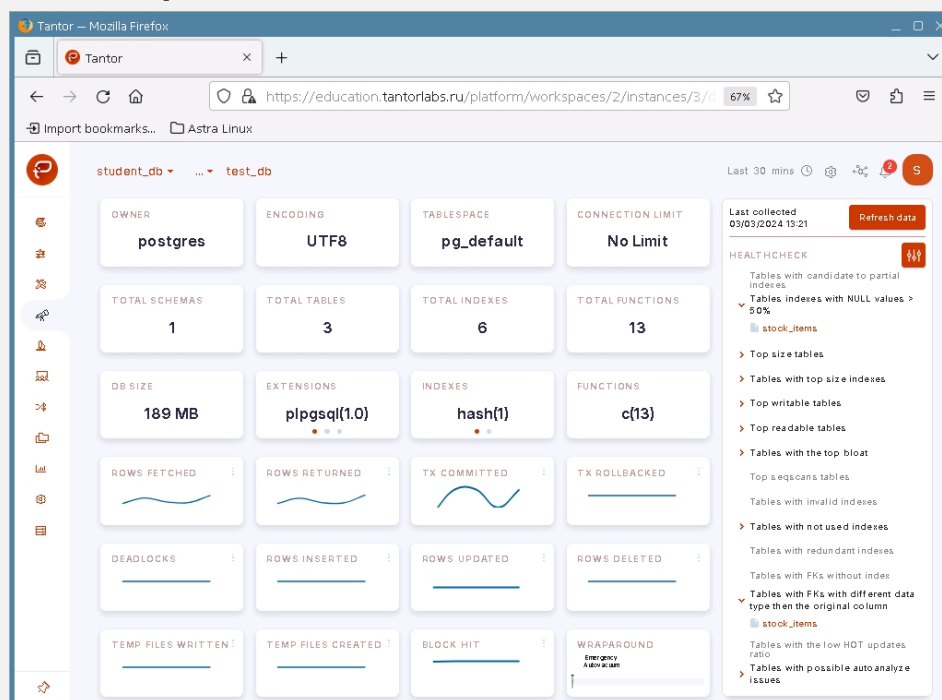
```
postgres=# \
```

```
Display all 108 possibilities? (y or n)
```

List of **useful** functions for the administrator:

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/functions-admin.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/functions-admin.html)

# Database Inspector in Tantor Platform



## Database Inspector in Tantor Platform

The Platform Inspector displays information about the database contents - the number and administration-relevant characteristics of SQL objects. It also analyzes and provides recommendations on the characteristics of objects that may be a problem.



# Demonstration

- Viewing a list of cluster databases
- Creating a database
- Renaming a database
- Database connection limit
- formatting psql output



## Demonstration

Viewing a list of cluster databases

Creating a database

Renaming a database

Database connection limit

formatting psql output

# Practice

1. Setting configuration parameters at different levels
2. Setting the search path in functions and procedures



## Practice

Setting configuration parameters at different levels

Setting the search path in functions and procedures



# 4

4b

## Physical structure



# PGDATA Cluster File Directory

- The cluster stores its files in the file system.
- The cluster directory is called PGDATA
- A directory or its subdirectories can be mount points, hard links, or symbolic links.
- Block devices and unformatted hard disk partitions are not used.
- By default, the operating system cache is used when working with cluster files.



## PGDATA Cluster File Directory

The database cluster files are stored in a directory called `PGDATA` , named after an operating system environment variable that is usually set to avoid specifying the directory to the cluster management utilities each time the utilities are called. The parameter (key) for the utilities is called `"-D directory"` or `"--pgdata directory"`. If you specify a parameter for the utility, it will override the value of the environment variable. Some utilities ( `pg_resetwal` ) require this parameter to be specified explicitly to avoid accidentally starting with an incorrectly set environment variable.

A cluster can store data files outside the `PGDATA` directory using "tablespaces," which we'll look at later in this chapter.

By default, the Tantor Postgres installer creates a directory

```
/var/lib/postgresql/tantor-se - 16 /data
```

for storing cluster files and services file

```
/usr/lib/systemd/system/tantor-se-server-16.service ,
```

where specifies the path to this directory. The `--edition` and `--major-version parameters` can be used to set other values for the installer. The rest of the Tantor Postgres utilities and software do not have default values, since several peer clusters can exist on the host. Each cluster has its own `PGDATA` directory . Each cluster is served by a single instance.

By default, the operating system cache is used when working with cluster files. The `debug_io_direct` developer configuration parameter allows you to set up work with data and log (WAL) files in direct i/o mode. This mode does not provide any practical advantages in performance or fault tolerance for PostgreSQL. This mode should not be used for working with data files.

PostgreSQL does not duplicate (does not multiplex) cluster files. Fault tolerance of working with files should be provided at lower levels - file system, hardware.

PostgreSQL uses the symbolic and hard links functionality of file systems. When administering `PGDATA` , you can use mount points, symbolic links, and hard links.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/storage-file-layout.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/storage-file-layout.html)

# PGDATA Cluster File Directory

- contains parameter files
- subdirectories

```
< ~/tantor-se-1c-17/data .[^]>
```

| .n            | Name          | Name                 | Name |
|---------------|---------------|----------------------|------|
| /..           | /pg_serial    | autoprewarm.blocks   |      |
| /base         | /pg_snapshots | current_logfiles     |      |
| /global       | /pg_stat      | pg_hba.conf          |      |
| /log          | /pg_stat_tmp  | pg_ident.conf        |      |
| /pg_commit_ts | /pg_subtrans  | postgresql.auto.conf |      |
| /pg_dynshmem  | /pg_tblspc    | postgresql.conf      |      |
| /pg_logical   | /pg_twophase  | postmaster.opts      |      |
| /pg_multixact | /pg_wal       | postmaster.pid       |      |
| /pg_notify    | /pg_xact      |                      |      |
| /pg_replslot  | PG_VERSION    |                      |      |



## PGDATA Cluster File Directory

PGDATA directory contains subdirectories with predefined names.

By default, the root of the PGDATA directory contains the cluster parameter text files:

postgresql.conf, pg\_hba.conf and pg\_ident.conf, although they can be located in other directories. The parameter file postgresql.auto.conf is located only in the root of PGDATA.

current\_logfiles - a text file with the name of the current file to which the message collector writes the server message log. The message collector is enabled by the logging\_collector configuration parameter (ALTER SYSTEM SET logging\_collector = on; ) Measuring the parameter requires restarting the instance. Using the message collector is recommended for industrial use or when there is a large volume of data written to the message log.

postmaster.opts - contains the command line options with which the instance was started

PG\_VERSION - contains the major release number

postmaster.pid - a "lock" file traditionally used in Linux. Contains the number (PID) of the main process of the instance; path to PGDATA, instance startup timestamp, instance port number, path to the Unix socket directory, IP address at which the instance is accessible, shared memory segment identifier (SHM). The segment size is small (56 bytes). Shared memory uses the mmap type by default. The type can be changed using the shared\_memory\_type parameter, but this is not necessary.

Main subdirectories:

base and global are directories of two tablespaces, they store data of cluster objects

pg\_stat and pg\_stat\_tmp directories of collected statistics. Active recording goes to the pg\_stat\_tmp directory, it is not worth placing it on the SSD (large recording volume), perhaps it is worth placing it in memory (in-memory file system).

pg\_tblspc - contains symbolic links to tablespace directories. It is convenient to see which cluster directories are located outside PGDATA.

pg\_wal - contains files ("segments") of the write ahead log (WAL). Loss of WAL files leads to the impossibility of starting the cluster

The log directory is created manually for the message log, the rest of the directories

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/kernel-resources.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/kernel-resources.html)

# Temporary objects

- are created in tablespace directories
- tablespaces for temporary files are specified by the `temp_tablespaces` parameter
- It is recommended to create a separate tablespace for temporary files
- temporary files are created if the data being processed does not fit into the process memory
- temporary files are created for temporary tables and their indexes when executing SQL commands that process large amounts of data (e.g. sorting, index creation)



## Temporary objects

The limit on the size of temporary files used by one process can be set by the **`temp_file_limit` parameter** . By default, there is no limit. The size of temporary files used by each process in the instance is limited. This parameter also limits the total number of temporary table files. If the limit is exceeded, the command executed by the process will be interrupted.

Temporary tables may be heavily used by applications.

When a temporary object is created, rows are created in the system catalog tables, and these are permanent storage objects. Regular files are also created in the file system. One transaction and one process work with a temporary table. Parallel processes cannot work with a temporary table; only the server process works with it.

If a temporary table is frequently cleared with the `TRUNCATE` command , this command (if you do not use extensions and assemblies that improve work with temporary tables) creates a new file in the file system with a new name and updates the `relfilenode` field in the `pg_class` table . The system catalog table file may grow in size and autovacuum may work more often. Statistics on temporary tables are also stored in permanent storage objects. Frequent creation of temporary tables with a large number of columns generates many rows in the system catalog tables. The system catalog tables can grow to tens of gigabytes.

Tantor Postgres has optimizations for working with temporary tables. The optimizations are enabled by **the `enable_delayed_temp_file` and `enable_temp_memory_catalog configuration parameters`**

When the **`enable_temp_memory_catalog` parameter is enabled** , no changes are made to the system catalog tables when temporary objects are created, deleted, or modified.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-client.html#GUC-TEMP-TABLESPACES](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-client.html#GUC-TEMP-TABLESPACES)

# Tablespaces

- are designed to allow a cluster to be located on multiple storage devices
- physically it is a directory in the file system
- are common objects of the cluster
- can store files of several cluster databases



## Tablespaces

Tablespaces are designed to allow a cluster to span multiple storage devices. The storage devices are mounted in different directories. A tablespace is a shared cluster object that represents a reference to a directory.

You can use the tablespace name in object creation commands, and the object files will be automatically created in subdirectories of that directory. You can grant USAGE privileges to roles on tablespaces. A tablespace has an owner. A tablespace is not part of a database or a schema, it is part of a cluster.

The reasons for creating tablespaces are as follows. The operating system may have file system mount points with different characteristics: space size, automatic space addition, performance, fault tolerance. The administrator can distribute database objects across these mount points (directories).

You can move objects between tablespaces, which will issue commands to the operating system to create, delete, and copy file contents block by block.

A tablespace that does not contain objects from any cluster database can be dropped.

# Tablespaces: Characteristics

- Subdirectories are created automatically
- **pg\_global** and **pg\_default** tablespaces are created .
- List tablespaces:  
    \db command and **pg\_tablespace** table
- the contents of the list are the same in all databases of the cluster
- **pg\_tablespace** - global system catalog table



## Tablespaces: Characteristics

After creation, the cluster has two tablespaces corresponding to the base and global subdirectories of the PGDATA directory :

```
postgres=# \db
```

```
List of tablespaces
```

```
Name | Owner | Location
```

```
-----+-----+-----
```

```
pg_default | postgres |
```

```
pg_global | postgres |
```

pg\_default tablespace is used by default for the template1, template0, postgres databases .

pg\_global tablespace is used to store global system catalog tables and should not be used to store user objects. This tablespace stores the pg\_tablespace table files .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/manage-ag-tablespaces.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/manage-ag-tablespaces.html)



# Tablespaces: Characteristics

- Without a cluster they don't exist
- Cannot move between clusters
- Cannot be backed up separately from the cluster
- Are not objects of schemes
- Have an owner role
- Tablespace directory:
  - created manually in the operating system
  - must be located in power loss tolerant file systems
  - It is recommended to create outside of PGDATA
  - You should not delete files in the directory manually, they are deleted automatically by SQL commands



## Tablespaces: Characteristics

Tablespaces are part of a database cluster. Even if they are not in PGDATA, they cannot be considered a stand-alone set of data files. The information about which objects are in which files is stored in the system catalog, not in the tablespace.

Tablespaces cannot be "detached" and "attached" to another database cluster. They cannot be backed up individually.

If a tablespace is damaged (file deleted, disk failure) and the instance is abnormally stopped, the instance will not start because it will need to restore the missing file blocks from the WAL log. The cluster will become completely unavailable. Therefore, you cannot place tablespaces with persistent storage objects on a file system that is not crash-tolerant (in RAM).

It is possible to place tablespaces only with temporary objects (temporary tables) if you are absolutely sure that there are no permanent storage objects in them on the file system in memory. In this case, you need to consider whether there is enough space for temporary tables. If the space runs out, the command to insert a row into a temporary table will return an error, the temporary table file will not be deleted. Only the table delete command can delete the file and free up space. The table truncation command may return an error, since it first creates a new file, and there may not be space for it.

An instance operates on the tablespace directory and its contents with the privileges of the user under which the instance processes are launched. When creating a tablespace at the file system level, the directory must be granted read-write privileges to the operating system user postgres.

# Tablespace Management Commands

- Create command:  

```
CREATE TABLESPACE name [OWNER role] LOCATION
'directory'
[WITH (parameter = value [, ...])]
```
- Change owner:  

```
ALTER TABLESPACE name OWNER TO role;
```
- Renaming:  

```
ALTER TABLESPACE name RENAME TO name;
```
- The database has a "default" tablespace, which contains the database's system catalog object files.
- The command to change the default tablespace for a database is:  

```
ALTER DATABASE database SET TABLESPACE name;
```



## Tablespace Management Commands

The database has a property called the default tablespace. It is where the system catalog object files are physically located. You can change the default tablespace, which will move the contents of the system catalog files to the new files.

Create tablespace command:

```
CREATE TABLESPACE name [OWNER role] LOCATION 'directory'
[WITH (parameter = value [, ...])]
```

Place the tablespace directory outside PGDATA .

The command to change the default tablespace for a specific database is:

```
ALTER DATABASE database SET TABLESPACE name;
```

Renaming a tablespace:

```
ALTER TABLESPACE name RENAME TO name;
```

Change of owner:

```
ALTER TABLESPACE name OWNER TO role;
```

Deleting a tablespace (the directory on disk is not deleted):

```
DROP TABLESPACE [IF EXISTS] name;
```

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-createtablespace.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createtablespace.html)

# Changing tablespace directory

- The cluster defines the tablespace directory by a symbolic link in the **PGDATA/pg\_tblspc** directory.
- Replacement procedure:
  - › Find the name of the symbolic link in the **PGDATA/pg\_tblspc** directory that points to the tablespace directory.
  - › Stop the instance
  - › Move tablespace directory
  - › Update the symbolic link
  - › Run the instance



## Changing tablespace directory

command to change the directory ( `LOCATION` property ) of a tablespace, since the tablespace may contain local object files of several databases in the cluster, and the session in which the command is issued should not see local objects of other databases. However, you can change the directory using the following procedure:

1)In the directory `PGDATA/pg_tblspc` there is a symbolic link whose name is the **oid** ( **number** ) of the tablespace. The link points to the directory of the tablespace:

```
ls -al | grep number
```

```
number -> /u01/postgres/my_tblspc
```

2)Make sure the **oid value** matches the name of the tablespace you want to move:

```
SELECT oid , spcname FROM pg_tablespace;
```

3)Stop the instance:

```
pg_ctl stop
```

4)Make sure the instance is stopped:

```
pg_controldata | grep down
```

```
Database cluster state: shut down
```

5)Move the tablespace directory to the desired location using an operating system or storage system command. You can move the directory within the same file system (mount point), or to any other:

```
mv /u01/postgres/my_tblspc /u02/postgres
```

6)Make sure that the user running the instance (postgres) has filesystem-level permissions to read and write to the directory and its contents.

7)Update symbolic link `PGDATA/pg_tblspc/ number` , which points to the tablespace directory:

```
ln -fs /u02/postgres/my_tblspc $ PGDATA/pg_tblspc/ number
```

8)Start the instance: `systemctl start tantor-se-server-16`

9)Check that the location has changed. For example, with the command `psql \db`

# Tablespace Parameters

- Four parameters: `seq_page_cost`, `random_page_cost`, `effective_io_concurrency`, `maintenance_io_concurrency`
- Overrides the parameter values set at the cluster level
- Affect the cost of the SQL command execution plan
- Can be specified when creating a tablespace:  
`CREATE TABLESPACE name WITH ( parameter = value [, ...] );`
- Can be changed later:  
`ALTER TABLESPACE name SET ( parameter = value [, ... ] );`
- You can reset the set value:  
`ALTER TABLESPACE name RESET ( parameter [, ... ] );`

|                | Read (MB/s) | Write (MB/s) |
|----------------|-------------|--------------|
| RND8K<br>Q32T1 | 494.43      | 336.17       |
| RND8K<br>Q64T1 | 504.69      | 297.60       |



## Tablespace Parameters

The current version of PostgreSQL has four parameters: `seq_page_cost`, `random_page_cost`, `effective_io_concurrency`, `maintenance_io_concurrency`, which can be set at the tablespace level. Setting these values affects the creation of command execution plans. The parameters are weights that are used by the planner to determine the cost of the execution plan. The parameters affect the planner's assessment of which resource is "more expensive" - the disk subsystem or the computing power of the processors. This can be useful if the tablespace is located on a storage system that is faster or slower than the storage system whose parameters are set in the cluster configuration files. Cluster configuration parameters with the same names:

`seq_page_cost` (float) - the cost of reading a block from disk when reading blocks sequentially. Files that store object data are divided into blocks. Sequential reading is considered if the block logically comes next - by offset from the beginning of the file. The instance does not know about the physical location of blocks in hard drive sectors. Default is 1.0

`random_page_cost` (float) - the cost of reading a block from disk with random access to file blocks. By default, 4.0 For SSDs, sequential and random access do not differ in speed, i.e. `random_page_cost` can be set equal to `seq_page_cost`. Decreasing `random_page_cost` relative to `seq_page_cost` inclines the scheduler to use the "Index Scan" access method instead of the "Seq Scan" access method. Simultaneously changing the values of both parameters changes the estimates of the cost of disk I/O relative to the cost of using CPUs.

`effective_io_concurrency` (integer) - Default is 1. Range is from 1 to 1000. A value of 0 disables asynchronous I/O (you should not set it to zero).

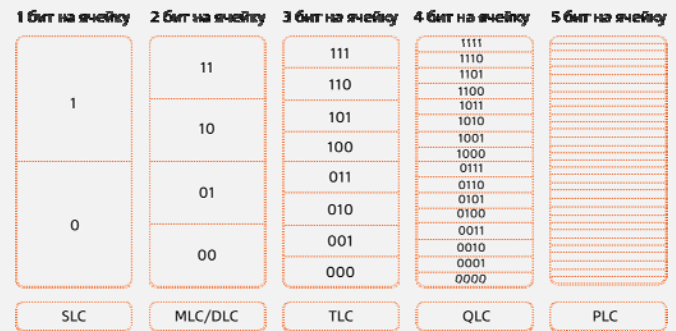
Sets a limit on the number of blocks that each server process will asynchronously read/write. For HDD-based storage systems, the starting point can be the number of hard drives. For SSDs, you can increase it to a value after which the acceleration of reading/writing with 8-kilobyte blocks stops growing significantly (for example, 64). This parameter is also taken into account by the scheduler when estimating the cost of Bitmap Index Scan.

`maintenance_io_concurrency` (integer) - Default is 10. Same meaning as `effective_io_concurrency`, but is used by background processes and server processes when executing data maintenance commands. For example, creating indexes, vacuuming. Its value must be no less than `effective_io_concurrency`.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-resource.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-resource.html)

# Working with log files

- `wal_recycle` and `wal_init_zero` parameters determine how the WAL log files are handled.
- By default, files are reused, newly created ones are filled with zeros.
- `PGDATA/pg_wal` stores log files, it can be mounted on a separate partition
- It is recommended to enable page image compression with the `wal_compression` parameter.
- When using an SSD, most of the file system partition must be free.



## Working with log files

The `PGDATA/pg_wal` directory is where the log files are created. The log records all changes to the cluster file data blocks (except for unlogged and temporary objects). This is a significant amount of space. By default, the `wal_recycle` configuration parameter is on. This means that files are not deleted, but renamed and their bodies are rewritten. Writing to the file bodies is done in a stream from the beginning of the file to the end (unless you switch to the next file with the `pg_switch_wal()` function). The second parameter is `wal_init_zero`, the default value is zero, which means: fill with zeros when creating files. When `wal_recycle = on`, files are reused and created infrequently, so the additional writing volume is small. When `wal_init_zero = off`, a command is given to write the last byte when creating a file to reserve space in the file system. Writing a byte rather than a block is optimal, since the operating system will use a block of an appropriate size.

If `PGDATA/pg_wal` is mounted on an SSD, it is worth ensuring that the volume of stored data does not exceed the volume of the "SLC cache" which is determined by the technology and algorithm of the controller. For TLC (triple level cell, 3 bits per cell), the volume of the "SLC cache" (a logical term meaning that the controller writes to the high-speed first layer that can withstand ~100 thousand write cycles and does not have time to transfer data to other layers because the SSD blocks occupied by WAL files are overwritten or cleared by discard) cannot be more than 1/3. If exceeded, then degradation of performance (depends on the algorithm of the controller) and durability occurs. In other words, when using SSD-based storage systems, the total volume of files on the `PGDATA/pg_wal` mount point should not be more than approximately 20% of the size. A large amount of free space will come in handy if the replica has difficulty receiving log data and the master will hold it. An example of an error related to lack of space. The server process that was unable to write data to the log is terminated:

```
LOG: server process (PID 6353) was terminated by signal 6 : Aborted
```

The instance crashes:

```
LOG: all server processes terminated; reinitializing
```

After restarting the instance, if there is still no space:

```
LOG: database system was not properly shut down; automatic recovery in progress
```

```
FATAL: could not write to file "pg_wal/xlogtemp.6479": No space left on device
```

It is worth mounting the WAL directory file system with the discard option (continuous TRIM) instead of the `fstrim` service, which is optimal for file systems storing data that does not change often. You can check whether DISCARD is enabled with the Linux command: `lsblk --discard`

The choice to leave `wal_recycle` enabled depends on the algorithm of the SSD memory controller and the file system. The `wal_init_zero` parameter should be disabled.

`wal_compression` parameter is disabled by default, allowing you to specify the compression algorithm that will be used to compress full page writes that are periodically written to the log. Possible values are `pglz`, `lz4`, `zstd`, `on`, `off`. The default is `off`.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-wal.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-wal.html)

[https://wiki.archlinux.org/title/Solid\\_state\\_drive\\_\(Russian\)](https://wiki.archlinux.org/title/Solid_state_drive_(Russian))

[https://en.wikipedia.org/wiki/Multi-level\\_cell](https://en.wikipedia.org/wiki/Multi-level_cell)

# The main data storage layer

- consists of files up to 1GB in size
- new files are created when the previous layer file reaches 1GB
- Maximum layer file size 32TB
- Work with files of all layers of persistent storage objects occurs through the buffer cache
- Work with files of all layers of temporary objects occurs through a buffer in the local memory of the server process
- Files of all layers are located in one directory of one tablespace.



## The main data storage layer

The object files in the tablespace are divided into types, which in PostgreSQL are called forks. All files are divided into 8K blocks. The minimum file size is 8K.

Object data is stored in the main layer files (main fork). First, the first main layer file is created and increased to 1 GB. Then the next file is created and increased to 1 GB, and then the following ones. The maximum size of a table (and any relation) is 32 terabytes (for a block size of 8 KB). Access to blocks of all layers of persistent storage objects occurs through a buffer cache common to all cluster processes. The size of the buffer cache is determined by the `shared_buffers` parameter.

Access to temporary object blocks (temporary tables and indexes on them, sequences) occurs through a buffer in the local memory of the server process. The buffer size is determined by the `temp_buffers` parameter . The value can be changed in a session, but only before the first access to a temporary object. Temporary object files have the same format as permanent storage objects.

Files of all layers are located in one tablespace in one directory and cannot be located in multiple tablespaces.

For regular objects, the file name prefix is a number and is stored in the `relfilenode` column of the `pg_class` table .

If the layer file ( `main`, `fsm` ) grows to 1Gb, a new file with the suffix " `.1` " is created. The following files will have the suffix " `.2` " and so on.

# Additional layers

## Free Space Map (fsm)

- not created for hash indexes
- created by vacuuming

## Visibility Map (vm)

- not created for indexes
- created by vacuuming
- two bits per base layer block
- the first bit set means that all lines of the block are current and there are no old versions
- the second bit set means that all rows of the block are frozen

## Initialization layer (init)

- created for unlogged tables and indexes
- file size one block without data



## Additional layers

For objects (except hash indexes), an "fsm" layer (free space map) is created. The files of this layer store a structure that reflects the availability of free space in the main layer blocks. The structure is organized not as a list, but as a balanced tree, so that processes can quickly find a block to insert a new record into the main layer block.

For relations (except indexes), a "vm" layer (visibility map and freeze map) is created. This layer file stores two bits per table's primary layer block. A one in the first bit indicates that all rows in the primary layer block are of the latest version (there are no rows that can be cleared). This bit is used by vacuuming and index only scan access methods; blocks with this bit are not accessed. If the second bit is one (the bit is set), this means that all rows on this page are frozen. This bit is used by vacuuming in freeze mode to skip blocks that were processed last time and have not changed since then. The file is created and updated by the process that performs vacuuming. If the file is missing (lost), it is recreated, and all primary layer blocks are processed.

Unlogged tables and indexes on them have an "init" layer consisting of a file of one block size (8Kb), which after an incorrect shutdown of the instance is copied to the location of the first file of the main layer (if there are other files, they are deleted) of the unlogged object and the object becomes empty.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/storage.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/storage.html)



# Location of object files

- If the object is located in `pg_default` then the files are located in the directory:  
`PGDATA/base/{oid database}`
- If in other tablespaces (the `reltablespace` column value in `pg_class` is not zero), then:  
`PGDATA/pg_tblspc/{reltablespace}/{oid of database}`
- Object file names start with `relfilenode`
- To get the location (relative to `PGDATA`) of the first data file of an object, use the `pg_relation_filepath(oid)` function.



## Location of object files

If the object is located in the default tablespace, its files are located in the directory:

`PGDATA/base/{oid of database from pg_database}`

If the object is located in other tablespaces (the value of the `reltablespace` column in `pg_class` is not zero), then the object files are located in the directory:

`PGDATA/pg_tblspc/{reltablespace from pg_class}/{oid of database}`

Object file names begin with `relfilenode` from `pg_class`.

For temporary objects, the file name has the form `t B _ FFF`, where `B` is a number that corresponds to the value in the name of the temporary schema in which the temporary object was created, `FFF` is the `rel filenode value of the pg_class` table. The values of the `relfilenode` and `oid` columns may not match, since `TRUNCATE`, `REINDEX`, `CLUSTER`, and other commands create a file with a new name, but do not change the `oid` of the object. Moreover, for some objects, the `relfilenode` value is zero.

To get the location (relative to `PGDATA`) of the first file of the main layer (main), use the function `pg_relation_filepath(oid)`

To get the file name prefix, use the `pg_relation_filenode(oid)` function.



# Tablespace and Database Sizes

- size of cluster tablespaces \db+ or pg\_tablespace\_size() function

```
postgres=# \db+
List of tablespaces
Name | Owner | Location | Permissions | Options | Size
-----+-----+-----+-----+-----+-----
pg_default | postgres | | | | 30 MB
pg_global | postgres | | | | 565 kB
postgres=# SELECT spcname, pg_size_pretty(pg_tablespace_size(oid)) FROM pg_tablespace;
 spcname | pg_size_pretty
-----+-----
pg_default | 30 MB
pg_global | 565 kB
```

- database size: \l+ or pg\_database\_size() function

```
postgres=# SELECT datname, pg_size_pretty(pg_database_size(datname)) FROM pg_database;
 datname | pg_size_pretty
-----+-----
pg_default | 30 MB
pg_global | 565 kB
```



## Tablespace and Database Sizes

The sizes of tablespaces of the entire cluster can be viewed using the `psql \db+` command.

```
postgres=# \db+
List of tablespaces
Name | Owner | Location | Permissions | Options | Size
-----+-----+-----+-----+-----+-----
pg_default | postgres | | | | 30 MB
pg_global | postgres | | | | 565 kB
```

You can also look at the `pg_tablespace_size(oid)` function :

```
postgres=# SELECT spcname, pg_size_pretty(pg_tablespace_size(oid)) FROM
pg_tablespace;
 spcname | pg_size_pretty
-----+-----
pg_default | 30 MB
pg_global | 565 kB
```

Database size command `\l+` or function `pg_database_size(name)` :

```
postgres=# SELECT datname, pg_size_pretty(pg_database_size(datname)) FROM
pg_database;
 datname | pg_size_pretty
-----+-----
postgres | 7737 kB
template1 | 7609 kB
template0 | 7377 kB
lab01iso88595 | 7537 kB
```

`pg_size_pretty()` function prints the number in a human-readable form by appending the characters **k B MB GB TB** .

# Sizing functions

- There are a set of functions for getting the size of object files
- `pg_relation_size(regclass, 'main' | 'vm' | 'fsm' init')` returns layer sizes
- `pg_indexes_size()` size of all indexes created on the table
- `pg_table_size()` table size (TOAST and all layers) without indexes
- `pg_total_relation_size()` table size including TOAST, all indexes and layers



## Sizing functions

Determining the size of an object can be useful to determine which objects take up the most space and require attention.

The list of functions that return the size of objects can be obtained by the command:

`\dfs *size` or by the query

```
SELECT proname, pg_get_function_arguments(oid) FROM pg_proc WHERE proname LIKE '%size' ORDER BY 1;
```

```
proname | pg_get_function_arguments
```

```
-----+-----
```

```
pg_column_size | "any"
```

```
pg_database_size | name
```

```
pg_database_size | oid
```

```
pg_indexes_size | regclass
```

```
pg_relation_size | regclass
```

```
pg_relation_size | regclass, text
```

```
pg_table_size | regclass
```

```
pg_tablespace_size | name
```

```
pg_tablespace_size | oid
```

```
pg_total_relatio n_size | regclass
```

```
(10 lines)
```

Functions can return the sizes of individual layers, the total size of a table with or without a TOAST table and indexes. A description of what function returns what can be found in the documentation section on administration functions:

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/functions-admin.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/functions-admin.html)

# Moving objects

- you can move files of tables, indexes, materialized views
- the contents of all layers are copied to new files
- sets an exclusive lock that is incompatible even with the `SELECT` command
- The `ALTER ALL IN TABLESPACE` command moves all objects of the same type in the current database.



## Moving objects

You can move table, index, and materialized view files from one tablespace to another.

When moving, files are read block by block and their contents are copied to new files. During the move, space in the tablespace directory is used where the objects are moved. After the move, the files in the original tablespace are deleted. The entire volume of moved data passes through WAL.

The second thing to consider is that locks placed on moved objects will not allow even `SELECT` commands to work with the objects , since almost all (except those launched with the `CONCURRENTLY` option ) require an `ACCESS EXCLUSIVE` level lock (exclusive mode of working with the object) . First, the move command is queued to receive a lock and waits until all transactions and any single commands finish working with the object that needs to be moved. `SELECT` commands can work for a long time. At the same time, the move command causes any commands wishing to work with the moved object to wait until it receives a lock and finishes the move.

Commands to move object files to another tablespace:

```
ALTER {TABLE | INDEX | MATERIALIZED VIEW } [IF EXISTS] name SET TABLESPACE
where;
```

```
ALTER {TABLE | INDEX | MATERIALIZED VIEW } ALL IN TABLESPACE name [OWNED BY
role [, ...]] SET TABLESPACE where [NOWAIT];
```

```
REINDEX [TABLESPACE where] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } [
CONCURRENTLY] name;
```

When using the `NOWAIT` option, an error is generated if the command cannot immediately acquire all locks on all affected objects.

There is a parameter `lock_timeout` that can be used to set the maximum time to wait for any command to obtain an explicit or implicit lock. If sessions are constantly working with the object, using this parameter can allow you to obtain a lock by setting an acceptable timeout.

`statement_timeout` parameter which must be greater than `lock_timeout` because the time it takes to obtain locks is taken into account. This parameter specifies the maximum time a command can execute before it is cancelled. For move commands, `statement_timeout` is unlikely to be useful.

# Change of scheme and owner

- you can change the owner of the object and the scheme of those objects that should have them
- To change the owner, use the command:  
`ALTER object_type name OWNER TO role;`
- To change the schema, use the command:  
`ALTER object_type name SET TO schema;`
- To mass reassign all role objects in one database to another role, use the command:  
`REASSIGN OWNED BY role TO role;`
- deleting objects belonging to a role in the database:  
`DROP OWNED BY name [CASCADE];`



## Change of scheme and owner

In addition to moving files to another tablespace, you can change the object owner and schema for those objects that should have them.

When changing a schema or owner, the changes are propagated to dependent objects. For example, indexes created on a table, integrity constraints, and sequences associated with columns are moved to another schema along with the table.

The owner and schema of an index always is and becomes the owner and schema of the table.

To change the owner, use the command:

```
ALTER object_type name OWNER TO role ;
```

To change the schema, use the command:

```
ALTER object_type name SET SCHEMA schema ;
```

These commands are not combinable with each other, are executed separately and require an exclusive lock, but for a short time.

To mass reassign all role objects in one database to another role, use the command:

```
REASSIGN OWNED BY role TO role;
```

There is also a command to delete objects belonging to a role in the database:

`DROP OWNED BY name [CASCADE];` The `CASCADE` option can be used to drop dependent objects owned by other roles.

# Reorganizing and moving tables with pg\_repack utility

- pg\_repack command line utility
- available in Tantor Postgres
- you need to install the extension in the databases where you need to reorganize the tables
- reorganization of table files with or without moving to another tablespace
- During the reorganization process, an ACCESS SHARE lock is set, allowing the execution of VACUUM, ANALYZE, SELECT, DML commands
- At the end of the reorganization, an exclusive lock is established for a short period of time



## Reorganizing and moving tables with pg\_repack utility

Tantor Postgres has a `pg_repack` extension that allows you to move objects to another tablespace without setting an exclusive lock on the objects for the duration of the operation. Instead, a lock of the most lenient level `ACCESS SHARE` is set. Such a lock is set by `SELECT` commands.

At the end of the move, an exclusive lock is acquired for a short time. You can set a timeout for acquiring this lock with the `--wait-timeout` option. After the timeout, `pg_repack` can cancel its operation by setting the `--no-kill-backend` option. By default, `pg_repack` cancels commands that prevent it from acquiring the lock. If after the same amount of time it still cannot acquire the lock, it will disconnect the backend processes with the `pg_terminate_backend()` function.

Moving objects to another tablespace is not the main purpose of `pg_repack`, this utility reorganizes object files to make the structure more compact.

You can specify the number of parallel sessions with the `--jobs` parameter to simultaneously rebuild multiple indexes on a single table in full table reorganization mode.

Reorganization of objects is started by the command line utility `pg_repack`, but for it to work the extension **must** be installed in the databases. To do this, simply execute the command `CREATE EXTENSION pg_repack;` in the databases whose objects you want to reorganize. Databases in which the extension is not installed are ignored by the utility.

Reorganization can be performed in different modes: analog `VACUUM FULL`, `CLUSTER`, `REINDEX`. Additional free space is required for the duration of the operation: the size of the objects being reorganized plus the changes in the lines that will accumulate during the transfer. The entire volume of transferred data passes through WAL logs.

The transfer is organized by creating a trigger that captures the changes and saves them to the change log table. A new table is created, the data from the original table is transferred to it, this is the longest part. After the transfer is complete, indexes are created on the new table. Then the accumulated changes are transferred from the change log table until there are a couple of dozen rows left in it, then an exclusive lock is set on the original table, these rows are transferred and the original table is replaced with a new one. When using integrity constraints with deferred verification on the table, errors in the extension are possible during the transfer of rows from the log table. The speed of operation is comparable to the speed of the `CLUSTER` command.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/be/pg\\_repack.html](https://docs.tantorlabs.ru/tdb/en/17_5/be/pg_repack.html)

# Reducing the size of table files with the pgcompacttable utility

- Utility written in Perl
- uses the standard pgstattuple extension
- Differences from pg\_repack:
  - › free space is equal to the size of the largest index, not twice the size of the table and indexes
  - › tables are processed with adaptive latency rather than full load
  - › cannot move files to another tablespace



## Reducing the size of table files with the pgcompacttable utility

The pgcompacttable utility is supplied with Tantor Postgres and is located in the `/opt/tantor/db/17/tools/pgcompacttable` directory .

The utility reduces the size of table and index files without heavy locking and without a sharp load that affects performance. Files can increase ("bloat") in size due to a large number of deleted rows or frequent row updates if autovacuum was unable to clean up old row versions.

Differences from pg\_repack:

- 1) The free space required for operation is equal to the size of the largest index. pg\_repack requires double the size of the table and indexes. pgcompacttable processes the contents of table files, indexes are rebuilt in turn, first the smaller, then the larger by file size
- 2) tables are processed with a delay to prevent sudden I/O spikes and delays in replication (if any). pg\_repack runs at maximum speed and load on the file system
- 3) cannot move files to another tablespace.

Before PostgreSQL version 17, there was a configuration parameter `old_snapshot_threshold`. When this parameter is set, the pgcompacttable utility cannot reduce file sizes, since VACUUM cannot perform the vacuum\_truncate phase. This is described in the documentation for the `old_snapshot_threshold` parameter. The pgcompacttable utility itself does not reduce file sizes, vacuum does.

Installation:

1)in databases you need to install the standard pgstattuple extension:

```
CREATE EXTENSION pgstattuple;
```

2)install Perl: `apt-get install libdbi-perl libdbd-pg-perl`

or `yum install perl-Time-HiRes perl-DBI perl-DBD-Pg`

3)For security reasons, others does not have rights to the

`/opt/tantor/db/17/tools` directory , the directory is owned by root and the root group. The postgres user does not have rights to this directory. This directory contains the pgcompacttable utility and other utilities. To access the pgcompacttable utility, you need to grant access rights to the utility. For example, with the following command: `sudo chmod 755 -R /opt/tantor/db/17/tools`  
[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pgcompacttable.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pgcompacttable.html)

# TOAST (The Oversized-Attribute Storage Technique)

- Regular tables (heap tables) store data "row by row"
- If the string does not fit in the data block, then TOAST technology is used.
- Four storage modes at the column level: PLAIN, EXTERNAL, EXTENDED, MAIN
- For small data types, storage in TOAST is not provided and the only mode is PLAIN
- For most other data types, the default mode is EXTENDED
- field-level compression is used, for small fields compression is ineffective



## TOAST (The Oversized-Attribute Storage Technique)

Regular tables (heap tables) store data "row by row" - all fields of one row physically next to each other, then all fields of another row, if these fields "fit" into one data block of 8 KB. If a row does not "fit" into a data block, then TOAST ( **The Oversized - Attribute Storage** Technique ) technology is used: some fields are transferred to a separate **TOAST service table**. The name of this table is not used in SQL commands and its use is completely transparent. You can set the storage mode of the fields of these columns on each column of the table using the ALTER TABLE name ALTER COLUMN name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN | DEFAULT } command. For example, for the EXTENDED mode set on columns, the fields of such columns will first be compressed and if a row with compressed fields fits into the block, the row will be saved in the table block. If the row does not fit into the block, then some of the row fields will be moved to the TOAST table. For each data type that may potentially not fit into the block (the data type that "supports" storage in TOAST), the storage mode is defined by default (called the "strategy" for storing fields of this type) and for most data types , the EXTENDED strategy is set . This mode is optimal if the SQL commands will process the entire field and the values are well compressed. If the values are poorly compressed or you plan to process the field values (for example, text fields with the substr, upper functions ), then it may be more effective to use the EXTERNAL mode . For data types that are small in size and are not intended for storage in TOAST (for example, the DATE type), the PLAIN storage "strategy" (default mode) is set and you cannot change the mode to another one using the ALTER TABLE command; the error " ERROR: column data type type can only have storage PLAIN " will be returned .

The storage method for 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 into the block, then separately determine for each row whether it is necessary to access the TOAST table rows, read its blocks and "glue" the parts of the fields ( chunk ) that are stored in it as rows of this table.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/storage-toast.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/storage-toast.html)



# 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 (chunks) of 1996 bytes:

```
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. Fixed-length data types do not support it, since their length is small and the same for any value ("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 ( $2^{30} = 1 \text{ Gb}$ ) of 1 or 4 bytes (32 bits) are allocated for the length at the beginning of the field in the block. 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 byte of the field header are not aligned. Values with four bytes of the field header are aligned along the `pg_type.typealign` boundary.

The fields taken out to TOAST are divided into parts - "chunks" (after compression, if it was applied) of 1996 bytes in size (the value is specified by the constant `TOAST_MAX_CHUNK_SIZE`), which are located in the rows of the TOAST table of 2032 bytes in size (the value is specified by the constant `TOAST_TUPLE_THRESHOLD`). **The values are chosen so that four rows fit into the 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](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-length data types are not compressed or TOASTed
- The storage strategy can be set with the command `ALTER 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 a variable length field stores the length of a field up to 127 bytes long
- 4 bytes at the beginning of a variable length 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 several blocks of table files. However, rows can be larger than 8Kb. TOAST is used to store them.

A btree index entry cannot exceed approximately one-third of a block (after compression of the indexed columns, if applied to the table).

TOAST supports varlena data types (`pg_type.typelen=-1`) . Fixed-length fields cannot be stored outside the table block, because there is no code written for these data types to implement storage outside the table block (in a TOAST table). In this case, the row must fit in one block and the actual number of columns in the table will be less than the 1600-column limit ( `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 length byte **are not aligned** , and **fields with 4 length bytes are aligned to `pg_type.typealign`.** Most variable-length types are aligned to 4 bytes (`pg_type.typealign=i` ). The lack of alignment provides a gain in storage space, which is noticeable for short values. But you should always remember to align **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.



# Field displacement in TOAST

- fields of lines longer than 2032 bytes are removed
  - › in this case, the fields are cut into parts of 1996 bytes
- the 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

The 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 the parts of the fields (chunk) that are stored in it as rows of this table.

A table can have only one associated TOAST table and one TOAST index (a unique btree index on the `chunk_id` and `chunk_seq` columns). The TOAST table OID is stored in the `pg_class.reltoastrelid` field.

When accessing each displaced 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.

The fields after compression (if any) are divided into parts (chunk) by **1996 bytes** :

```
postgres@tantor:~$ pg_controldata | grep TOAST
```

Maximum size of a TOAST chunk: **1996**

In PostgreSQL, a row is considered for TOASTing a portion of its fields if the row is larger than **2032** bytes. The fields will be compressed and considered for TOASTing until the row fits into 2032 bytes or `toast_tuple_target` bytes if the value was set with 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 of 1997 bytes will also generate 2 chunks, the second chunk of 9 bytes, the first of **1988** bytes.

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 string, 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 removal) of each field, the size of the string is checked. If the size does not exceed `toast_tuple_target` (default 2032 bytes), the string is saved to the buffer and the processing of the string is finished
- processing starts with the EXTENDED and EXTERNAL fields from largest to smallest
- 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 string buffer (or 2GB for sessions with the `enable_large_allocations=on` configuration parameter set ).

Four-pass displacement algorithm:

1) 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.

An 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 string, 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 of 1997 bytes or more create a **second** chunk of small size, due to which only 3 large chunks fit into the TOAST block
- for 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 larger than 2032 bytes, and the field will be cut into 1996-byte chunks. Because of this, a **small chunk will appear for a field larger than 1996 bytes**, which will be inserted by the server process into a block with a large chunk. For example, to insert 4 rows into a 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));
```

в блок TOAST поместится 3 длинных чанка:

```
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     |

The total size of a string with a long chunk is 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));
```

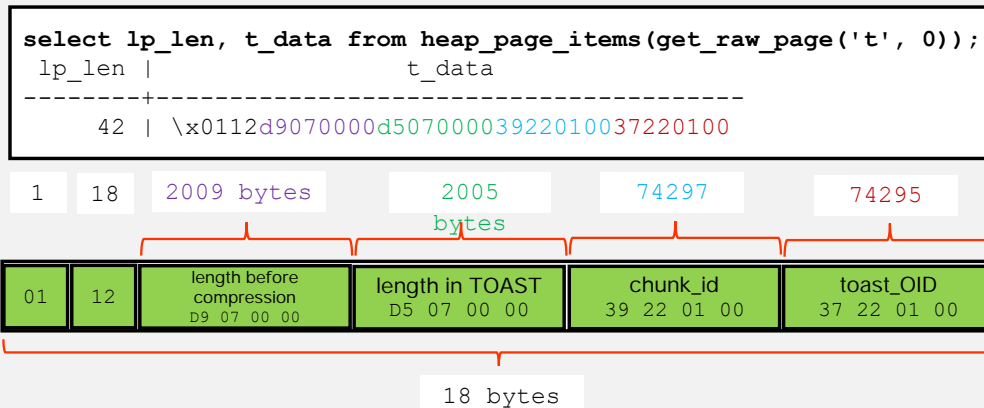
| lower | upper | special | pagesize |
|-------|-------|---------|----------|
| 48    | 1944  | 8184    | 8192     |

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

- any table in TOAST can have no more than  $2^{32}$  fields (~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:



## TOAST Limitations

In PostgreSQL there is no special service area at the end of table blocks:

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) stored as a single row in the TOAST table.

The main table field stores a pointer to the first chunk **of size 18 bytes** (regardless of the field size). These 18 bytes store the `varatt_external` structure, described in `varatt.h`:

the first byte has the value 0x01, this is a sign that the field is TOASTed;

the second byte is the length of this record (value 0x12 = 18 bytes);

4 bytes length of the field with the field header before compression;

4 bytes is the length of what is TOASTed;

4 bytes - pointer to the first chunk in TOAST (chunk\_id column of TOAST table);

4 bytes - oid of toast table (`pg_class.reltoastrelid`)

The chunk\_id column (oid type 4 bytes) can have 4 billion ( $2^{32}$ ) 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.

The MAIN mode is used for storage inside a block in compressed form, EXTERNAL - for storage in TOAST in uncompressed form, EXTENDED - for storage 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-length types, the PLAIN mode is set, which cannot be changed by the `ALTER TABLE` command, the error "ERROR: column data type type can only have storage PLAIN" will be returned.

# 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 in TOAST\_TUPLE\_TARGET
- TOAST\_TUPLE\_TARGET can only be overridden at the table level :
- ALTER TABLE t SET (toast\_tuple\_target = 2032) ;
- TOAST\_TUPLE\_THRESHOLD is not overridden
- default\_toast\_compression configuration parameter 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 TOAST storage until the rest of the row (complete: with row header) fits into TOAST\_TUPLE\_TARGET. This 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

Be the first person in the country, to see:

```
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)= 2032 .
```

**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-length field stores the length of a field up to 127 bytes long, the first 4 bytes at the beginning of a variable-length field store the length of a field for bytea longer than 126 bytes. Alignment is 4 bytes. 2008-4-4-4=1996.



# Columnar Storage: General Information

- reduce the complexity of accessing columnar data by storing column values together
- it is possible to compress data efficiently compared to field compression in heap tables
- To use the columnar storage method, it is enough to specify the name when creating a table:

```
CREATE TABLE name (...) USING columnar ;
```

- extension creates table access method **columnar**
- the list of access methods is in the `pg_am` table



## Columnar Storage: General Information

The idea of the columnar storage method (implementation of Hydra) is to reduce the labor intensity of accessing data in columns by storing column values together. With this storage method, the data of one column is stored physically next to each other, either entirely or in a large number of rows. Due to the fact that the data in each column is similar, it is possible to effectively compress data in large "sets" of rows (chunks). The size of the "set" can be set at the table level by the `columnar.chunk_group_row_limit` parameter .

To use the columnar storage method, it is enough to specify the storage method when creating a table:

```
CREATE TABLE name (...) USING columnar ;
```

Changing the storage format with the `ALTER TABLE .. SET ACCESS METHOD` command is not implemented. If you implement a function to change the storage method and call it, for example, `alter_table_set_access_method` , then this function will have to reload all data into new files with table locking. Non-blocking data reloading is a more universal and complex task that should be implemented by a separate extension and called, for example, `pg_reorg`.

Since the data storage differs from the usual one, the heap table access method cannot be used and the extension creates its own table ( `amtype = 't'` ) access method. The list of access methods is stored in the `pg_am` system catalog table:

```
SELECT * FROM pg_am WHERE amtype = 't';
```

```
oid | amname | amhandler | amtype
```

```
-----+-----+-----+-----
```

```
2 | heap | heap_tableam_handler | t
```

```
18276 | columnar | columnar_internal.columnar_handler | t
```

The **pg\_columnar extension** creates the `columnar` and `columnar_internal` schemas that it uses to store its objects.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/hydra.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/hydra.html)



# Columnar storage: features of use

- **UPDATE and DELETE are not supported**
- supported commands: TRUNCATE, INSERT (including one row), COPY
- parallel scanning is not implemented
- **supported** index types used in integrity constraints: **btree** , hash
- other index types are not supported: gist, gin, spgist, brin
- compatibility with table partitioning: partitions can have different storage formats
- no service columns ctid, xmin, xmax



## Columnar storage: features of use

Does the columnar format replace the heap format ? No. The heap format works more efficiently with single-row queries. In databases serving typical business tasks (OLTP - online transaction processing) such as sales management, warehouse and personnel records, queries to single rows are more common than retrieving a large number of rows.

Storing in columnar format is more efficient in cases of: periodic loading of a large set of rows into a table, reading only part of the columns, no updating and deletion of individual rows. The columnar format is convenient for data warehouses, where data is accumulated and analytical queries (processing a large number of rows for the purpose of creating a report or analyzing the accumulated data) are performed on them.

The columnar format does not support UPDATE and DELETE . TRUNCATE, INSERT (including one row) , COPY are supported . This is the main thing that limits the use of this storage method. When trying to execute unsupported commands, an error will be returned:

```
DELETE FROM perf_columnar WHERE id=0;
```

```
ERROR: UPDATE and CTID scans not supported for ColumnarScan
ctid service column in columnar format tables .
```

TOAST with columnar format is not used, since large values are stored internally. Parallel scanning is not implemented - the selection is performed by one server process. Indexes of the btree, hash type are supported for fast checking of integrity constraints ( PRIMARY KEY , UNIQUE which are supported ), as well as in the partitioning option. Index types gist, gin, spgist, brin are not supported , since index access is inefficient. The extension is compatible with table partitioning: a partitioned table can have sections using both heap and columnar storage formats.

`relnamespace='pg_catalog'::text::regnamespace order by 1;` will give the TOAST table names of the 36 system catalog tables that have them.

# Columnar Storage: Parameters

- the extension has parameters with the prefix "**columnar.**"
- Sequential writing to an ordered data table can improve compression, significantly reducing the size of indexes and the amount of data blocks to scan
- data is often ordered by time and called a "Time Series"
- The most efficient compression algorithm is zstd, it is installed by default
- When reading small amounts of data, using indexes can be more efficient



## Columnar Storage: Parameters

Sequential writing to a table of ordered data can significantly reduce the size of indexes (if they are created) and the size of the unpacked rowsets ( `chunk` ). The size of the unpacked data is reduced because it is typical to specify a filter condition on the column by which the data is ordered, and most of the requested data is stored together in the case of sequential insertion. For example, data for the last hour or the last hundred orders (the order number is generated by the sequence) are selected. Therefore, it is recommended to order the rows before inserting them into the table.

In practice, ordering by time is often encountered. Such data is called "Time Series", the rows are inserted sequentially in time. For example, sequential insertion into a table of measurement indicators of some parameter (stock price, vehicle coordinates) in time. Compression in such tables is usually more effective, since the values of adjacent fields are similar or even do not change (the stock price in successive transactions was the same).

The most efficient data compression method is zstd.

When reading small amounts of data, using indexes can be more efficient.

The extension has configuration options:

`columnar.chunk_group_row_limit`, `columnar.compression_level`,  
`columnar.stripe_row_limit`, `columnar.compression`, `columnar.planner_debug_level`

You can set options at the table level. The options can be viewed in the `options` view of the `columnar` schema.

```
SELECT * FROM columnar.options;
```

```
-[RECORD 1]-----+-----
relation | perf_columnar
chunk_group_row_limit | 10000
stripe_row_limit | 150000
compression | zstd
compression_level | 3
```

# Demonstration

- Directory for temporary files
- Moving a tablespace directory



## Demonstration

Directory for temporary files  
Moving a tablespace directory

# Practice

1. Creating a database connection
2. Tablespace Contents
3. Sequence file
4. Moving a table to another tablespace
5. Moving a table to another tablespace using pg\_repack
6. Using pgcompacttable
7. Columnar storage pg\_columnar



## Practice

Creating a database connection

Tablespace Contents

Sequence file

Moving a table to another tablespace

Moving a table to another tablespace using pg\_repack

Using pgcompacttable

Columnar storage pg\_columnar



# 5

5

## Logging



# Diagnostic log

- accumulates diagnostic messages from instance processes
- used for:
  - › diagnostics of problems
    - Monitoring and performance tuning
  - › security audit
  - › historical analysis of what happened when the instance was running
  - › query execution analysis

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*.log
23:17:09.415 [784] LOG: database system is ready to accept connections
23:17:09.732 [791] LOG: autotune successfully autotuned 13763 of 13763 previously-loaded blocks
23:27:09.762 [786] LOG: checkpoint starting: time
23:27:19.982 [800] STATEMENT : select * from tickets1 where ticket_no='0005432020304';
23:27:21.200 [800] ERROR : index "tickets1_ticket_no_idx" contains unexpected zero page at block 3
23:27:21.200 [800] HINT: Please REINDEX it.
```



## Diagnostic log

The PostgreSQL message log is used to monitor and analyze instance activity. Instance processes can generate messages about what they are doing. These messages are useful for:

- 1) **problem diagnostics** - whether processes encountered errors or unexpected situations
- 2) tuning and monitoring performance. For example, messages about long-running queries or long table vacuuming
- 3) security audit. For example, logging the creation of sessions, granting privileges.
- 4) historical analysis of what happened when the instance was running. For example, at what time the instance started and **began** accepting connections
- 5) analysis of **query execution** . For example, logging of query plans and command execution statistics.

Messages from all processes are directed to a single log. Tantor Postgres has `pgaudit` and `pgaudittofile` extensions that can be used to log security events to a separate file to avoid cluttering the diagnostic log with security audit messages.

# Message importance levels

- In the PostgreSQL core code, the extension library code, and the plpgsql code, messages are marked with severity levels.
- **log\_min\_messages** sets the severity levels of messages that will be sent to the diagnostic log
  - › Valid values and order of importance for **this** parameter: DEBUG5 , DEBUG4 , DEBUG3 , DEBUG2 , DEBUG1 , INFO , NOTICE , **WARNING, ERROR, LOG, FATAL, PANIC**
- **client\_min\_messages** sets the messages of which importance levels will be transmitted to the client that created the session
  - › Valid values and order of importance for **this** parameter: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, **NOTICE, WARNING, ERROR**

```
postgres=# \dconfig *_min_messages
List of configuration parameters
Parameter | Value
-----+-----
client_min_messages | notice
log_min_messages | warning
(2 rows)
```



## Message importance levels

In the PostgreSQL core code, the extension library code, and the plpgsql code, messages are marked with severity levels .

Configuration parameter **log\_min\_messages** sets the messages of which importance levels will be transferred to the diagnostic log (" log "). The default value is WARNING. This means that messages of levels "more important" than WARNING will be logged: **WARNING, ERROR, LOG, FATAL, PANIC** . Valid values and **order of importance** for this parameter are: DEBUG5 , DEBUG4 , DEBUG3 , DEBUG2 , DEBUG1 , INFO , NOTICE , **WARNING , ERROR , LOG , FATAL, PANIC** .

Configuration parameter **client\_min\_messages** sets the messages of which importance levels will be transmitted to the client that created the session. The default value is NOTICE . This means that messages of levels "more important" than NOTICE will be logged: **NOTICE, WARNING, ERROR** . The possible values and order of importance for this parameter are: DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, **NOTICE, WARNING, ERROR** .

The order of importance and the set of values for the two listed parameters **differ** .

There is no point in changing the default values.

plpgsql has the command RAISE {DEBUG, LOG, INFO, NOTICE, WARNING, EXCEPTION} 'format', expressions USING parameter = value; to generate messages. The EXCEPTION level is similar to ERROR, rolls back the transaction to an implicit savepoint before BEGIN and passes control to the EXCEPTION section, if such a section exists. Example:

```
postgres=# DO $$ BEGIN
 RAISE INFO 'info: %!', 'variable1' USING
DETAIL = 'info detail', HINT = 'info hint';
 RAISE EXCEPTION 'text: %!', 'variable' USING ERRCODE = 'P0001',
 DETAIL = 'error detail', HINT = 'error hint';
END; $$;
INFO: info: variable1!
DETAIL: info detail
HINT: info hint
ERROR: text: variable!
DETAIL: error detail
HINT: error hint
CONTEXT: PL/pgSQL function inline_code_block line 4 at RAISE
```

# Log location

- `log_destination` specifies the location where diagnostic messages will be output.
  - › valid values: `stderr`, `csvlog`, `jsonlog`, `syslog`
- if there are several locations, they will be output to all locations at the same time
- `logging_collector=on` starts a background process **logger** that intercepts `stderr` and sends messages to the `log_directory` directory.
- in the root of `PGDATA` a text file is created with the name `current_logfiles`, in which the location and current (where the recording is currently going) names of the diagnostic log files are written
- `log_filename` specifies the name of the file (in `log_destination` one value is `stderr`) or log files
  - › convenient value `log_filename = postgresql-%F.log`



## Расположение журнала

`log_destination` parameter allows you to specify a location, separated by commas, where diagnostic messages will be output. Valid values are: **stderr**, **csvlog**, **jsonlog**, **syslog**. If there are several locations, they will be output to all locations simultaneously. The default value is `stderr`, which means that messages are output in text form to the standard error stream. If the instance is started via `systemd`, then `stderr` is directed to the general linux log by default. If the instance is started by the `pg_ctl start` utility, then `stderr` is output to the terminal. If the instance is started by the `pg_ctl start -l path_to_file` utility, that is, with the `-l` or `--log=path_to_file` parameter, then the log is directed to a file.

In industrial operation, the instance is started by `systemd`. It is not convenient to use the general linux log, since it stores messages from instance processes mixed with messages from other operating system processes, and it is convenient to use the `logging_collector=on` parameter.

`logging_collector=on` parameter starts the background process **logger**, which intercepts `stderr` and directs messages to the `log_directory` directory, in which a file or files named `log_filename` are created. In order for `logging_collector` to be able to log messages, `stderr` and/or `csvlog` and/or `jsonlog` must be specified in `log_destination`. These values specify the format of the log messages. The `csvlog` and `jsonlog` formats are not created without **logger**. When `stderr` and/or `csvlog` and/or `jsonlog` are specified in `log_destination`, a text file named `current_logfiles` is created in the `PGDATA` root, which records the location and current (where the recording is currently being made) names of the diagnostic log files. An example of the contents of this file:

```
stderr log/postgresql-2025-12-25.log
csvlog log/postgresql-2025-12-25.csv
jsonlog log/postgresql-2025-12-25.json
```

`log_filename` parameter specifies the name of the log file or files. The default value is `postgresql-%Y-%m-%d_%H%M%S.log`. The file extension is valid for the `stderr` text format, for `csv` and `json` the file extension (`log`) is replaced by `csv` and `json`. The mask in the default value (`%H%M%S`) causes a file with a new name to be created each time the instance is started. A more convenient value is `postgresql-%F.log` (`%F` is equivalent to `%Y-%m-%d`).

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-logging.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-logging.html)

<https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html>



# Transferring syslog messages

- when `log_destination = syslog` messages are sent to the operating system service `syslog`
- Not all messages are output to `syslog`, all are output to **logger**
  - > **logger** is preferable - it does not lose messages
- Configuration parameters for interaction with `syslog`:

```
postgres=# \dconfig syslog*
List of configuration parameters
Parameter | Value
-----+-----
syslog_facility | local0
syslog_ident | postgres
syslog_sequence_numbers | on
syslog_split_messages | on
(4 rows)
```



## Transferring syslog messages

Messages can be passed to the operating system's `syslog` service .

`log_destination` parameter can be set to `syslog` . Configuration parameters for `syslog` are:

```
postgres=# \dconfig syslog*
List of configuration parameters
Parameter | Value
-----+-----
syslog_facility | local0
syslog_ident | postgres
syslog_sequence_numbers | on
syslog_split_messages | on
(4 rows)
```

Some types of messages may not appear in `syslog` output. For example, dynamic library linking error messages, error messages when executing scripts specified in `archive_command` configuration parameters . Therefore, it is recommended to use **logger** :

```
logging_collector=on;
log_filename=postgresql-%F.log
```

`syslog` is not reliable, it may truncate or lose messages, especially when they are needed. By default, `syslog` flushes every message to disk, which reduces performance. To disable this synchronous logging, you can add `"-"` before the file name in the `syslog` configuration file .

# Rotate diagnostic log files

- When using **logger**, log files may be rotated
  - › old files are not deleted
- `log_truncate_on_rotation` parameter allows time-based rotation (but not size-based rotation or instance startup) to overwrite existing log files rather than append to them.
- `log_file_mode` parameter sets the permissions on the diagnostic log files (default 0600 )
  - › value 0640 will allow members of the postgres group to read the files

```
postgres=# \dconfig *rotation*
List of configuration parameters
Parameter | Value
-----+-----
log_rotation_age | 1d
log_rotation_size | 10MB
log_truncate_on_rotation | off
(3 rows)
```



## Rotate diagnostic log files

To prevent log files from growing, **logger** provides for their rotation. When using syslog, rotation is configured in syslog. Parameters for configuring rotation:

```
postgres=# \dconfig *rotation*
List of configuration parameters
Parameter | Value
-----+-----
log_rotation_age | 1d
log_rotation_size | 10MB
log_truncate_on_rotation | off
(3 rows)
```

`log_truncate_on_rotation` parameter allows time-based rotation (but not size-based rotation or instance startup) to overwrite existing log files rather than appending to them. For example, if `log_filename=postgresql-%a.log` and `log_rotation_age=1d`, then a separate file will be created for each day of the week, and if `log_truncate_on_rotation=on`, then the files will be overwritten once per day.

`log_file_mode` parameter sets permissions on diagnostic log files. The value 0640 will allow members of the group to read the files. This parameter does not change permissions on the directory where the files are located.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/logfile-maintenance.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/logfile-maintenance.html)

# Diagnostic log

- `logging_collector` (default `off` ) recommended to set 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 " alter 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

The 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) . There are 15 levels defined in `elog.h`.

Enabling the log collector process:

`logging_collector=on` (default `off`). It is recommended to set the value to `on`. By default, messages are sent 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 write some messages (and it does the right thing), `logger` does not clear the `errlog` buffer and the instance processes generating messages are blocked until `logger` writes everything that has accumulated (which is also correct). In other words, `logger` does not lose messages, which can be important for diagnostics. Such a situation can occur due to a failure in writing to log files or enabling a 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` , defaults to `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 statements that fail with an error.

`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(s) is specified in the text file `PGDATA/current_logfiles`

Importance levels from most to least detailed for the log:

`DEBUG5` `DEBUG4` `DEBUG3` `DEBUG2` `DEBUG1` for debugging  
`INFO` messages, usually 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` stop server processes by main process

# Diagnostic parameters

- logging configuration parameters more than 35
- there are also more than 8 parameters for debugging SQL commands
- extensions can have 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
debug_print_switch | 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 | on
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 written to the log. If the value is zero, the execution duration of all commands is written. By default, -1 does not write 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 execution duration of commands increases; problems with commands: for example, an index is no longer used and the execution time of commands increases sharply. Example:

LOG: duration: 21585.110 ms

STATEMENT: CREATE INDEX ON test(id);

Duration and command are given.

`log_duration=off` logs the duration of all commands after their execution. 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 plus dml commands), all (all commands). By default, none. It is recommended to set to ddl. ddl commands usually set a higher blocking level, which increases contention. Using this parameter, you can identify or exclude the execution of the ddl command as a cause of reduced performance. Commands with syntax errors are not recorded by default. If you need to log commands with syntax errors, you need to set `log_min_error_statement=ERROR` (or more detail). Do you need to 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 an entry with `log_statement=ddl` set :

LOG: statement: drop table test;

# Monitoring temporary file usage

- `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 to identify 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

Let's look at examples of using the diagnostic log and logging parameters.

If there are a lot of commands and the log is cluttered, you can use the `log_min_duration_sample` and `log_statement_sample_rate` parameters . Parameter

`log_transaction_sample_rate` has a large overhead because all transactions are processed.

`cluster_name = 'main'` Empty by default. Recommended to set. The value is appended to the name of the instance processes, making them easier to identify. On a replica, `wal_receiver` is used for identification by default .

`log_temp_files='1MB'` logs the names and sizes of created temporary files at the time of their deletion. Why at the time of deletion? Because the 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 smaller 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/17\\_5/se/runtime-config-logging.html](https://docs.tantorlabs.ru/tdb/en/17_5/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. When such messages occur, 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

Logging is useful for monitoring the autovacuum.

`log_autovacuum_min_duration` , by default set to 10 minutes. If autovacuum exceeds this time while processing the table, a message will be written to the cluster log. When 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** are the CPU usage time. When vacuuming, processes can send blocks for writing and wait for the I/O operation to be performed, without loading the CPU.

First of all, it is worth looking at " **elapsed:** " - this is the duration of the autovacuum transaction, i.e. holding the horizon. For TOAST, there will be a separate entry about vacuuming (including aggressive) 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 paying 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** " are large, then this may explain a 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 the bottleneck.



# Monitoring checkpoints

- the first entry is written to the log when the checkpoint begins
- total = 09:31:35.070 - 09:27:05.095
  - › approximately equals to 270 seconds, which is 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 synchronized) - the number of data files (in tablespaces whose blocks are located in the buffer cache) to which writing was performed and for which an fsync call was sent at the end of the checkpoint
- longest=0.003 s (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 like this:

```
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
```

Как читать записи:

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. The total value = 09:31:35.070 - 09:27:05.095 which is approximately 270 seconds, which is obtained by multiplying checkpoint\_completion\_target \* checkpoint\_timeout (0.9\*300=270). The number of blocks that checkpoint should send for writing is calculated quite often, but towards the end of the interval the I/O load may suddenly increase and checkpoint may not be able to do it in time. 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. A large sync time indicates increased I/O load. These metrics apply 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 blocks of slru cache buffers, but their sizes are small. longest=0.003 s (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 checkpoint, 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
```



## Описание записей log\_checkpoints

log\_checkpoints создает записи в логе такого вида:

```
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
```

Как читать записи (продолжение):

- 4) **wrote 4315 buffers** the number of dirty blocks written by checkpoint. Along with checkpoint, dirty blocks can be written by server processes and bgwriter. **(26.3%)** is the percentage of the total number of buffer cache buffers specified by the `shared_buffers` parameter . In the example **4315 / 16384 \* 100% = 26.3366699%**
- 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` (the distance that was 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 is correct. The number of files to remove 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: time` 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` the following message will be displayed:

`LOG: checkpoint starting: wal`

If the checkpoint for wal starts earlier than `checkpoint_warning`, the following 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= '30s'`

For checkpoints after instance restart:

`LOG: checkpoint starting: end-of-recovery immediate wait`

8) `estimate=109699 kB` (distance that was expected) - updated by the formula:

if ( `estimate` < `distance` ) `estimate` = `distance`

else `estimate`=0.90\*`estimate`+0.10\*`distance`; (numbers are fixed in PostgreSQL code)

The estimate is calculated by the checkpoint code to estimate how many WAL segments will be used at the next checkpoint. Based on the `estimate`, at the end of the checkpoint it is determined how many files to rename for reuse and the rest 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 is correct. Such values should be most of the checkpoints. This is the purpose of displaying the `estimate value`. The volume of log records between checkpoints is `distance`.

9) Between checkpoints passed `09:27:05.095 - 09:22:05.087` = 300.008 seconds, which with high accuracy equals `checkpoint_timeout=300s`

## pg\_waldump utility and log\_checkpoint entries

- 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), then 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` is printed, but not printed in `redo`

```
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: 00000001000000080000000B
```

```
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 8/0BC98978;
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
```



## pg\_waldump utility and log\_checkpoint 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: 00000001000000080000000B
```

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 `pg_waldump` output in `lsn` and `prev zero` is printed, but not printed in `redo` . Before the number 8, zeros are also invisibly present, but their absence does not create confusion. You can remember that after the slash there must be eight HEX symbols.

# pg\_waldump utility and log\_checkpoints entries

- **lsn** 8/1164B2E8 end of checkpoint record
- **redo** 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 beginning of the previous log entry
- **distance** log size from start of previous checkpoint to start of completed checkpoint ' 8/0BC98978 '::pg\_lsn- ' 8/05177990 '::pg\_lsn

```
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: 00000001000000080000000B
```

```
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
```



## pg\_waldump utility 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, this record is read. All records from **redo** to **lsn** must be read and overlaid on the cluster files. After overlaying **lsn** The cluster files are considered consistent.

**prev** 8/1164B298 is the address of the beginning of the previous log entry. You can slide "backward" through the log. However, the log entries do not contain the LSN of the next log entry. Why? The address of the next log entry can be calculated by the **len (rec/tot) field: 148/148** , which stores the length of the log entry. The minimum length of a log entry is 26 bytes ( expected at least 26 ). In this case, the actual length of the log entry is padded to 8 bytes. The actual length of the entry 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 до 8 байт = LSN начала следующей записи**

The size of the journal entries can be determined from the log or control file entry. The recovery time depends on it.

The amount of WAL written at a checkpoint is calculated from these fields:

```
select pg_wal_lsn_diff('8/1164B2E8','8/BC98978'); = 94054768 = 91850kB.
```

volume from the beginning to the end of the checkpoint is 91850 kB.

The volume from the beginning of the previous checkpoint to the beginning of the completed one, that is, **the distance** between checkpoints:

```
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 .

# Connection logging

- detects whether there are too frequent connections and short sessions
- Spawning a session to execute a single request is not optimal
- The following parameters are used to log connections:

```
log_connections=on
log_disconnections=on
pgaudit.log_connections=on
pgaudit.log_disconnections=on
```

```
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
```



## Connection logging

Logging connections to an instance is useful for detecting excessively frequent connections and short sessions. There may be applications that operate in a connect-request-disconnect mode. The reason for the existence of such applications is the use of scripting languages that were used to create html pages. Each page was created by a single script. In the databases used by such applications, session creation was an inexpensive operation in terms of resource use, since the database functionality was fairly simple and designed for simple queries to single tables without authentication and access control. In the PostgreSQL DBMS, when a session is created, a process is spawned in the operating system, preparatory operations are performed (authentication, access rights verification, signal registration, memory structure allocation), which is relatively labor-intensive. Spawning a session to execute a single request is not optimal and leads to useless use of computing resources and memory. Oracle Database uses the same architecture. Industrial applications use languages and architectures that use connection pools at the application server level. PostgreSQL instance monitoring applications may connect to the database every few seconds or tens of seconds, execute a few queries, and disconnect.

Connection logging allows you to identify such applications and monitoring systems. To do this, it is enough to log each connection and its duration.

The second reason for using connection logging is the regulatory security requirements "connection audit". Auditing is used to determine in case of hacking of the software system what data and when was stolen in order to eliminate the consequences. For example, replacing payment card numbers or access codes that were stolen. Therefore, connection auditing can be enabled constantly and this is a given for the DBMS administrator.

The following parameters are used to log connections:

```
log_connections=on
log_disconnections=on
pgaudit.log_connections=on
pgaudit.log_disconnections=on
```

# log\_connections parameter

- `log_connections=on` writes to the cluster diagnostic log attempts to connect to an instance, authentication attempts, and successful authentication
- the value can only be changed at the cluster level
- Neither the role level nor the database level can set the parameter:

```
postgres=# alter user alice set log_connections = on;
ERROR: parameter "log_connections" cannot be set after connection start
```

- example messages:

```
LOG: connection received: host="10.0.2.15"
LOG: connection authorized: user=fff database=fff application_name=psql
FATAL: role "fff" does not exist
```



## Параметр log\_connections

The `log_connections=on` parameter records attempts to connect to an instance, authentication attempts, and successful authentication in the cluster diagnostic log. The parameter may generate multiple diagnostic log entries associated with a single connection. By default, the parameter is disabled. The value can only be changed at the cluster level, although the documentation declares the ability to change the value before establishing a connection, but this is incorrect.

Neither the role level nor the database level can set the parameter:

```
postgres=# alter user alice set log_connections = on;
ERROR: parameter "log_connections" cannot be set after connection start
```

To apply the value, simply reread the configuration:

```
alter system set log_connections = on;
select pg_reload_conf();
```

When attempting to connect to a non-existent default role, the log will contain the following line:

```
FATAL: role "fff" does not exist
```

When you enable this parameter, the following lines will be added:

```
LOG: connection received: host="10.0.2.15"
LOG: connection authorized: user=fff database=fff application_name=psql
FATAL: role "fff" does not exist
```

You can add attributes to the message using the `log_line_prefix` parameter, which can only be set at the cluster level. To change the parameter, simply reread the configuration files. By default, the parameter value is `'%m [%p] '` and the date, time, and process number in square brackets are added to the message:

```
2035-01-01 11:01:01.924 MSK [1773081]
```

By adding the value `%r` or `%h` to the parameter `log_line_prefix = ' %h '` you can add logging of the IP address or name of the client node. The IP address will be present in each message:

```
10.0.2.15 FATAL: role "fff" does not exist
```

## log\_disconnections parameter

- `log_disconnections=on` writes one message to the diagnostic log when the server process servicing the session stops
- The value can only be changed at the cluster level, as well as on the client before establishing a connection:

```
export PGOPTIONS="-c log_disconnections=on"
```

- Neither the role level nor the database level can set the parameter:

```
postgres=# alter database postgres set log_disconnections = on;
ERROR: parameter "log_disconnections" cannot be set after connection start
```

- example messages:

```
LOG: disconnection: session time: 0:00:00.007 user=postgres database=postgres
host=127.0.0.1 port=34298
```



## log\_disconnections parameter

`log_disconnections=on` parameter writes **one** message to the diagnostic log when the server process servicing the session stops. The message includes **the session duration** . By default, the parameter is disabled. The value can be changed at the cluster level, and, unlike the `log_connections` parameter , the `log_disconnections` parameter **can be** changed before creating a session at the session level:

```
export PGOPTIONS="-c log_disconnections=on -c work_mem=5MB"
```

```
psql -h 127.0.0.1 -c "show work_mem;"
```

```
work_mem
```

```

```

```
5MB
```

You can also change the parameter by setting the connection property in the JDBC driver.

Example of a log message:

```
tail -n 1 $PGDATA/log/postgresql-*
```

```
LOG: disconnection: session time: 0:00:00.007 user=postgres database=postgres
host=127.0.0.1 port=34298
```

The value can be changed by a role with the SUPERUSER attribute or by a role that has been granted privileges to change the parameter.

Neither the role level nor the database level can set the parameter:

```
postgres=# alter user alice set log_disconnections = on;
```

```
ERROR: parameter "log_disconnections" cannot be set after connection start
```

To apply the value, simply reread the configuration:

```
alter system set log_disconnections = on;
```

```
select pg_reload_conf();
```

The advantage of this parameter is that if a utility or client frequently connects to the database, you can set an environment variable on the client node before starting it and disable logging of their sessions. This reduces unnecessary messages in the cluster diagnostic log. The `log_connections` parameter is not changed in this way, since it is used for security logging, and disabling logging of connection attempts on the client side would be undesirable.



# pgaudit and pgaudittofile extensions

- `pgaudit` and `pgauditlogtofile` extensions can be used to direct messages about session creation and duration to a separate audit file or files
- For extensions to work, you need to download their libraries:

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile;
```

- Extensions operate independently and in parallel with the cluster log and are controlled by their own parameters, which are prefixed with "`pgaudit.`"
- `pgaudit.log_connections` and `pgaudit.log_disconnections` parameters are similar to the PostgreSQL parameters of the same name and can create similar entries in a separate audit file



## pgaudit and pgaudittofile extensions

When using the `log_connections` and `log_disconnections` parameters, messages are written to the cluster log. During production use, many other messages are written to this log. Logging connections is not needed for routine analysis and clutters the general log, making it difficult to read more important messages. It is desirable that logging of connections, ddl commands, and other commands be performed not in the cluster log, but in a separate file or files.

Tantor Posgres has the `pgaudit` and `pgauditlogtofile` extensions, which can be used to direct messages about session creation and duration to a separate audit file or files.

`pgauditlogtofile` extension redirects the records created by the `pgaudit` extension to a separate file or files. Without it, the records go to the cluster log. The `pgauditlogtofile` extension depends on the `pgaudit` extension and does not work without it. To use the extensions, you only need to load two libraries :

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile ;
```

Extension libraries register configuration parameters in the instance, which can be used to customize what is logged and where. Extensions operate independently and in parallel with the cluster log and are controlled by their own parameters, which are prefixed with "`pgaudit.`"

There are 18 parameters in version 16. 7 parameters are related to the `pgauditlogtofile` library, including the `pgaudit.log_connections` and `pgaudit.log_disconnections` parameters. These parameters are similar to the PostgreSQL parameters of the same name and can create similar records, but only in a separate audit file, not in the cluster log, which is a big advantage of these parameters. The advantage outweighs the disadvantages in the form of the need to load two libraries and the inconvenience of their use. Library parameters are set only at the cluster level, specifying these parameters in an environment variable leads to an error and the inability to connect, unlike the standard parameters: `export PGOPTIONS="-c pgaudit.log_connections=off"`

`psql`

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: parameter "pgaudit.log_connections" cannot be changed now
```

`pgaudit.log_disconnections` parameter, unlike the `log_disconnections` parameter, cannot be set when creating a session.

# Configuring pgaudit and pgaudittofile extensions

- **seven** parameters are related to the `pgauditlogtofile` library
- To create an audit log, you need to set the `pgaudit.log` parameter to at least 'misc'
  - > 'none' - no audit log file is created
  - > 'role' and 'ddl' `pgaudit.log_connections` and `pgaudit.log_disconnections` have no effect

```
postgres=# \dconfig pgaudi*
List of configuration parameters
Parameter | Value
-----+-----
pgaudit.log | none
pgaudit.log_autoclose_minutes | 0
pgaudit.log_catalog | on
pgaudit.log_client | off
pgaudit.log_connections | off
pgaudit.log_directory | log
pgaudit.log_disconnections | off
pgaudit.log_filename | audit-%F.log
pgaudit.log_level | log
pgaudit.log_parameter | off
pgaudit.log_parameter_max_size | 0
pgaudit.log_relation | off
pgaudit.log_rotation_age | 1d
pgaudit.log_rotation_size | 0
pgaudit.log_rows | off
pgaudit.log_statement | on
pgaudit.log_statement_once | off
pgaudit.role |
(18 rows)
```



## Configuring pgaudit and pgaudittofile extensions

The disadvantage of using extension parameters is that you need to set the `pgaudit.log` parameter to at least 'misc' to create an audit log. But the value 'misc' forces the logging of DISCARD, FETCH, CHECKPOINT, VACUUM, SET commands and bloats the audit log. **With the default value 'none', no log file is created. When set to 'role' and 'ddl', the parameters `pgaudit.log_connections` and `pgaudit.log_disconnections` have no effect.**

Installing the `pgauditlogtofile` extension with the command is useless because there are no objects in the extension:

```
create extension pgauditlogtofile;
\dxx+ pgauditlogtofile
Objects in extension "pgauditlogtofile"

(0 rows)
```

`pgaudit` extension includes two triggers and two trigger functions:

```
event trigger pgaudit_ddl_command_end
event trigger pgaudit_sql_drop
function pgaudit_ddl_command_end()
function pgaudit_sql_drop()
```

The substitution variable '`%F`' (or its equivalent `%Y-%m-%d`) in the audit log and cluster log names is more convenient than the default value (`%Y%m%d_%H%M`) in that it does not create a separate file when the instance is restarted. A new file is created once per day. Example of setting values:

```
alter system set pgaudit.log_filename = 'audit- %F .log';
alter system set log_filename = 'postgresql- %F .log';
```



# 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 one line is output, 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 .056 user=oleg database=db1
host=[vm1]
```

In the example the session duration is **4** seconds.

`log_connections=on` writes to the log 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 before authentication a server process is spawned, this is a labor-intensive operation. The parameter is useful for identifying problems when a client continuously tries to connect with an incorrect password or to a non-existent database or with a non-existent role. The disadvantage is that unsuccessful attempts differ only in **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'` . Default is 1 second, which is too low and creates significant overhead on busy instances.
- `log_startup_progress_interval='10s'` should not be disabled
- `log_recovery_conflict_waits=on` . Defaults to off . The startup process 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 receive messages in the diagnostic log when a process waits longer than: `deadlock_timeout='60s'` . The default is 1 second, which is too short and creates significant overhead on busy instances. It is recommended to configure the `deadlock_timeout` value so that messages about waiting for a lock are rare. As a first approximation, you can focus on the duration of a typical transaction (for a replica - the longest request).

In version 15, the parameter `log_startup_progress_interval='10s'` appeared , which **should not be disabled** (set to zero). If the startup process (performing recovery) encounters a long operation, a message about this operation will be written to the log. Messages will allow you to identify either problems with the file system or 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` . By default, 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 longer than `deadlock_timeout` . The delay can occur because the server process on the replica is executing a command or transaction (for read repeatability) and is blocking WAL application due to the `max_standby_streaming_delay` parameter (by default, 30s). Allows you to identify cases of replica lagging. Effective on the replica, on the master you can set it in advance. It is recommended to set 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 presentation, but there is 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

1. What information is included in the log?
2. Location of server logs
3. How information gets into the log
4. Adding csv format
5. Enabling the message collector



## Practice

What information is included in the log?

Location of server logs

How information gets into the log

Adding csv format

Enabling the message collector



# 6

6

## Security



# Users (roles) in a database cluster

- accumulates diagnostic messages from instance processes
- used for:
  - › diagnostics of problems
    - Monitoring and performance tuning
  - › security audit
  - › historical analysis of what happened when the instance was running
  - › query execution analysis

```
postgres@tantor:~$ cat $PGDATA/log/postgresql-*.log
23:17:09.415 [784] LOG: database system is ready to accept connections
23:17:09.732 [791] LOG: autotune successfully autotuned 13763 of 13763 previously-loaded blocks
23:27:09.762 [786] LOG: checkpoint starting: time
23:27:19.982 [800] STATEMENT : select * from tickets1 where ticket_no='0005432020304';
23:27:21.200 [800] ERROR : index "tickets1_ticket_no_idx" contains unexpected zero page at block 3
23:27:21.200 [800] HINT: Please REINDEX it.
```



## Users (roles) in a database cluster

In PostgreSQL, a role is the same as a user. A role is a shared cluster object. This means that once created, a role is visible in any database in that cluster. A role is similar to a group in other security systems.

Most objects (tables, procedures, functions, databases, schemas, etc.) must have one role that owns the object. As long as a role has objects in its ownership, the role cannot be deleted. The owner of an object can be changed.

Roles can have privileges (rights) to objects. For example, the privilege to create objects in a schema, the privilege to insert rows into a table, or execute a procedure. Privileges in PostgreSQL are analogous to object privileges in Oracle Database.

Roles have nine attributes (properties). Attributes can be changed after a role is created. A role can be renamed. Attributes can be likened to system or administrative privileges (privileges to perform actions without being tied to an object) in Oracle Database. For example, the SUPERUSER attribute is similar to the SYSDBA administrative privilege in Oracle Database, and the BYPASSRLS attribute is similar to the EXEMPT ACCESS POLICY system privilege.

Roles and schemas are different objects. Schemas are local database objects, roles are common cluster objects.

Roles are created with the CREATE ROLE or CREATE USER command, deleted with DROP ROLE, and changed with ALTER ROLE.

The difference between CREATE USER and CREATE ROLE is that the first command sets the LOGIN attribute **by default**, while the second sets the **NOLOGIN attribute**:

```
postgres=# create user alice;
```

```
CREATE ROLE
```

```
postgres=# create role bob;
```

```
CREATE ROLE
```

```
postgres=# \du
```

```
List of roles
```

```
Role name | Attributes
```

```
-----+-----
```

```
alice |
```

```
bob | Cannot login
```

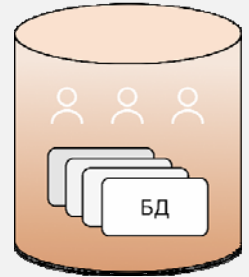
```
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS
```

```
https://docs.tantorlabs.ru/tdb/en/17_5/se/database-roles.html
```

# Users (roles)

- The list of roles can be viewed using the commands `\d u s` `\d g s`
- The list of roles is stored in the global table `pg_authid`
- There is a pseudo-role `public`, which includes all roles
  - > she can be given privileges

```
postgres=# select * from pg_authid where rolname='postgres'\gx
-[RECORD 1]-----
oid | 10
rolname | postgres
rolsuper | t
rolinherit | t
rolcreaterole | t
rolcreatedb | t
rolcanlogin | t
rolreplication | t
rolbypassrls | t
rolconndefs | -1
rolpassword | SCRAM-SHA-256$4096:oejDqb5wqdHcuVXBU0H/VA==...
rolvaliduntil |
```



## Users (roles)

The list of cluster roles can be viewed with the `\d u s` or `\d g s` (`u` - user, `g` - group) command or in the `pg_authid` table or the `pg_roles` view:

```
postgres=# \dus
```

| List of roles               |                                                            |
|-----------------------------|------------------------------------------------------------|
| Role name                   | Attributes                                                 |
| pg_checkpoint               | Cannot login                                               |
| pg_create_subscription      | Cannot login                                               |
| pg_database_owner           | Cannot login                                               |
| pg_execute_server_program   | Cannot login                                               |
| pg_maintain                 | Cannot login                                               |
| pg_monitor                  | Cannot login                                               |
| pg_read_all_data            | Cannot login                                               |
| pg_read_all_settings        | Cannot login                                               |
| pg_read_all_stats           | Cannot login                                               |
| pg_read_server_files        | Cannot login                                               |
| pg_signal_backend           | Cannot login                                               |
| pg_stat_scan_tables         | Cannot login                                               |
| pg_use_reserved_connections | Cannot login                                               |
| pg_write_all_data           | Cannot login                                               |
| pg_write_server_files       | Cannot login                                               |
| postgres                    | Superuser, Create role, Create DB, Replication, Bypass RLS |

```
postgres=# select * from pg_authid where rolname='postgres'\gx
```

```
-[RECORD 1]-----
oid | 10
rolname | postgres
rolsuper | t
rolinherit | t
rolcreaterole | t
rolcreatedb | t
rolcanlogin | t
rolreplication | t
rolbypassrls | t
rolconndefs | -1
rolpassword | SCRAM-SHA-256$4096:oejDqb5wqdHcuVXBU0H/VA==...
rolvaliduntil |
```

There is also a pseudo-role `public` , which includes all cluster roles:

```
postgres=# drop role public;
```

```
ERROR: cannot use special role specifier in DROP ROLE
```

```
postgres=# create role public;
```

```
ERROR: role name "public" is reserved
```

# Attributes (parameters, properties) of roles

- there are 7 attributes of the type on/off
- LOGIN - the right to create an initial connection to databases
- SUPERUSER - bypasses permission checks except for initial connection
- REPLICATION LOGIN - a role with these attributes (without the LOGIN attribute, the REPLICATION attribute is useless) has the right to connect via the replication protocol and reserve the entire cluster
- CREATEROLE - a role can create roles
  - › roles have no owner
  - › the created role is granted to the creating role with the ADMIN OPTION
  - › neither the WITH ADMIN grant nor the CREATEROLE attribute allow you to elevate your role's privileges by creating and switching to the created role
- Grant with ADMIN OPTION does not give the right to change the attributes CREATEROLE, BYPASSRLS, REPLICATION, CREATEDB, SUPERUSER
  - › a role can change these attributes for roles that have ADMIN OPTION only if it has the same attribute



## Attributes (parameters, properties) of roles

LOGIN - the right to create an initial connection to databases. Having connected while in a session with a database, you can switch in this session to the granted role with the SET ROLE command (return to the initial role with the RESET ROLE command). The role to which you switch in the session may not have the LOGIN attribute. You cannot change the database by switching to another role, this is only possible by creating a new connection (in psql, the `\connect` command ).

SUPERUSER - bypasses access rights checks except for the initial connection. Without the LOGIN attribute, a role with the SUPERUSER attribute cannot connect to any database.

CREATEDB - the role can create databases. Having created a database, the role will become the owner of the created database and will be able to delete this database. Only the owner or a role with the SUPERUSER attribute can delete a database.

REPLICATION LOGIN - a role with these attributes (without the LOGIN attribute, the REPLICATION attribute is useless) has the right to connect via the replication protocol and back up the entire cluster.

CREATEROLE - a role can create roles. Roles have no owner. The created role is granted by the creator with the ADMIN OPTION. This option allows changing attributes (password, INHERIT, CONNECTION LIMIT, VALID UNTIL), renaming, deleting a role granted with this option, granting and revoking this role from others, changing configuration parameters that are set at the role level (ALTER ROLE command name SET work\_mem = '16MB'), changing the role description with the COMMENT command, changing the SECURITY LABEL of this role. Therefore, by default, a role with the CREATEROLE attribute can change and delete roles created by it. If a role has the SUPERUSER attribute, only roles with the SUPERUSER attribute can delete it or change its properties. A grant with the ADMIN OPTION does not give the right to change the CREATEROLE, BYPASSRLS, REPLICATION, CREATEDB, SUPERUSER attributes. A role can change these attributes for roles for which it has ADMIN OPTION only if it has the same attribute. It is difficult to remember these rules. It can be considered that neither the WITH ADMIN grant nor the CREATEROLE attribute allow you to elevate your role's privileges by creating and switching to the created role.

BYPASSRLS - a role with this attribute is not affected by Row Level Security policies.

CONNECTION LIMIT - number of sessions (initial connections). By default, the number of sessions is unlimited (value -1).

VALID UNTIL '2030-11-01' - forty validity periods of a timestamp with time zone password.

# Атрибут INHERIT и GRANT WITH INHERIT

- The INHERIT attribute is set by default.
- Allows inheritance of rights granted to database objects
- If you set the NOINHERIT attribute on a role, it will not inherit the privileges of the roles that are given to it.
- When issuing a role, you can explicitly specify
  - › will the rights of the granted role work for the sessions of the grantee
  - › can a grantee switch to a grant role

```
postgres=# grant postgres to alice with inherit false, set true;
GRANT ROLE
postgres=# \c postgres alice
You are now connected to database "postgres" as user "alice".
postgres=> set role postgres;
SET
postgres=#
postgres=# grant postgres to bob with inherit true, set false;
GRANT ROLE
postgres=# \c postgres bob
You are now connected to database "postgres" as user "bob".
postgres=> set role postgres ;
ERROR: permission denied to set role "postgres"
```



## INHERIT and GRANT WITH INHERIT attribute

The INHERIT attribute is set by default. If a role is set to NOINHERIT, it will not inherit rights to specific database objects from roles that are granted to it, and it will need to switch to the granted roles to work with their object rights. If a role is set to NOINHERIT, it will stop inheriting by default the object rights that are granted to the roles that it is a member of. However, this can be overridden by specifying the WITH INHERIT option explicitly when granting the role with the GRANT ... WITH INHERIT true or WITH INHERIT false command. Example:

```
postgres=# grant postgres to alice with inherit false, set true;
GRANT ROLE
postgres=# \connect postgres alice
You are now connected to database "postgres" as user "alice".
postgres=> set role postgres;
SET
postgres=# select current_user, session_user, current_role, user;
 current_user | session_user | current_role | user
-----+-----+-----+-----
 postgres | alice | postgres | postgres
(1 row)
postgres=# grant postgres to bob with inherit true, set false ;
GRANT ROLE
postgres=# \connect postgres bob
You are now connected to database "postgres" as user "bob".
postgres=> set role postgres ;
ERROR: permission denied to set role "postgres"
```

**SET false** option does not allow you to switch to a role and gain the right to use its attributes (for example, SUPERUSER).

The **LOGIN, CREATEROLE, BYPASSRLS, REPLICATION, CREATEDB, SUPERUSER** attributes are never inherited . To use them, you must switch to a role that has this attribute using the SET ROLE command. You can return to the original role with which the session was created using the commands:

```
RESET ROLE; SET ROLE NONE; SET ROLE initial_role;
https://docs.tantorlabs.ru/tdb/en/17_5/se/role-membership.html
```



# Switching a session to another role and changing roles

- `SET [ SESSION | LOCAL] SESSION AUTHORIZATION role;` switches a session to another role
- The command can only be executed by the superuser.
  - › used to allow the superuser to switch session to another user and then switch back to the original one
- The current user can be changed with the `SET ROLE` command.
  - › checking of rights to objects is performed for the current user

```
postgres=# set session authorization alice;
postgres=> select current_user, session_user, current_role, user;
 current_user | session_user | current_role | user
-----+-----+-----+-----
alice | alice | alice | alice
postgres=> set role bob;
postgres=> select current_user, session_user, current_role, user;
 bob | alice | bob | bob
postgres=> reset role;
postgres=> select current_user, session_user, current_role, user;
alice | alice | alice | alice
postgres=> reset session authorization;
postgres=# select current_user, session_user, current_role, user;
postgres | postgres | postgres | postgres
```



## Switching a session to another role and changing roles

The `SET [ SESSION | LOCAL] SESSION AUTHORIZATION role` command switches the session to another role. `LOCAL` is used only in an open transaction and switches the session until the transaction ends.

The command can only be executed if the session was originally created (authenticated) by the superuser. This command can be used to allow the superuser to switch the session to another user and then switch back to the original superuser session.

```
postgres=# set session authorization alice;
postgres=> select current_user, session_user, current_role, user;
 current_user | session_user | current_role | user
-----+-----+-----+-----
alice | alice | alice | alice
postgres=> set role bob;
postgres=> select current_user, session_user, current_role, user;
 bob | alice | bob | bob
postgres=> set role pg_checkpoint;
postgres=> select current_user, session_user, current_role, user;
pg_checkpoint | alice | pg_checkpoint | pg_checkpoint
postgres=> reset role;
postgres=> select current_user, session_user, current_role, user;
alice | alice | alice | alice
postgres=> reset session authorization;
postgres=# select current_user, session_user, current_role, user;
postgres | postgres | postgres | postgres
```

`SET SESSION AUTHORIZATION` cannot be used in the `SECURITY DEFINER` function .

The current user can be changed with the `SET ROLE` command . The object rights check is performed for **the current user**. `SET ROLE` will switch to **any** role of which the role under which authentication was performed is a direct or indirect member.

The names of the functions **current\_user**, **current\_role**, **user** are synonyms . These functions are called without parentheses according to the SQL standard.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/functions-info.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/functions-info.html)

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-set-session-authorization.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-set-session-authorization.html)

# Predefined roles

- appeared in PostgreSQL starting with version 14
- 15 roles plus the public pseudo-role
- can grant a role with SUPERUSER attributes or a role that has WITH ADMIN permission on the role being granted
- service roles cannot be deleted
- useful to avoid giving superuser privileges to perform requested actions

```
postgres=# \duS
```

| List of roles               |              |
|-----------------------------|--------------|
| Role name                   | Attributes   |
| -----+-----                 |              |
| pg_checkpoint               | Cannot login |
| pg_create_subscription      | Cannot login |
| pg_database_owner           | Cannot login |
| pg_execute_server_program   | Cannot login |
| pg_maintain                 | Cannot login |
| pg_monitor                  | Cannot login |
| pg_read_all_data            | Cannot login |
| pg_read_all_settings        | Cannot login |
| pg_read_all_stats           | Cannot login |
| pg_read_server_files        | Cannot login |
| pg_signal_backend           | Cannot login |
| pg_stat_scan_tables         | Cannot login |
| pg_use_reserved_connections | Cannot login |
| pg_write_all_data           | Cannot login |
| pg_write_server_files       | Cannot login |



## Predefined roles

Before version 14, there were no predefined (predefined), i.e. service roles automatically created when creating a cluster. There was only a public service role, which includes all users (roles) of the cluster.

These service roles cannot be deleted:

```
postgres=# drop role pg_checkpoint;
```

```
ERROR: cannot drop role pg_checkpoint because it is required by the database system
```

These roles can be granted by a role with the SUPERUSER attributes or by a role that has the WITH ADMIN right to the role being granted.

The only member of the `pg_database_owner` role is always the current database owner role. `pg_database_owner` can own objects and receive permissions on objects. It makes sense to grant permissions to this role and make it the owner of objects, since when cloning a database or changing the owner of a database, there will be no need to change privileges and ownership. The rights granted to `pg_database_owner` (for example, in the `template1` database) will be acquired by the creator of the new database that clones it. By default, it owns the `public` schema, that is, the database owner controls the use of the `public` schema in his database.

`pg_signal_backend` has the ability to execute the `pg_cancel_backend(pid)` and `pg_terminate_backend(pid)` functions, which terminate the execution of commands or sessions other than superuser sessions.

`pg_read_server_files`, `pg_write_server_files`, `pg_execute_server_program` give the right to access files and run programs under the operating system user under which the instance (postgres) is launched. For example, change the contents of the `pg_hba.conf` file or delete PGDATA files.

`pg_monitor`, `pg_read_all_settings`, `pg_read_all_stats`, and `pg_stat_scan_tables` are given to roles for monitoring and performance tuning.

`pg_checkpoint` has permission to execute the checkpoint command;

`pg_maintain` has permission to execute `VACUUM`, `ANALYZE`, `CLUSTER`, `REFRESH MATERIALIZED VIEW`, `REINDEX`, `LOCK TABLE` commands on all objects, as if it had `MAINTAIN` permission on those objects.

`pg_read_all_data`, `pg_write_all_data` have the right to read and change data of all objects (tables, views, sequences), as if it had `SELECT`, `INSERT`, `UPDATE`, `DELETE` rights on these objects and `USAGE` rights on all schemas.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/predefined-roles.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/predefined-roles.html)

# Rights to objects

- When an object is created, an owner is assigned
- By default, the owner and superuser have rights to the created object.
- Other roles need rights
- Each type of object has its own set of rights.
  - › The names of the rights are given in the table
- Rights are granted and revoked using the GRANT and REVOKE commands.

| right        |   | types of objects                                              |
|--------------|---|---------------------------------------------------------------|
| SELECT       | r | LARGE OBJECT, SEQUENCE, TABLE (and similar), COLUMN           |
| INSERT       | a | TABLE, column                                                 |
| UPDATE       | w | LARGE OBJECT, SEQUENCE, TABLE, COLUMN                         |
| DELETE       | d | TABLE                                                         |
| TRUNCATE     | D | TABLE                                                         |
| REFERENCES   | x | TABLE, COLUMN                                                 |
| TRIGGER      | t | TABLE                                                         |
| CREATE       | C | DATABASE, SCHEMA, TABLESPACE                                  |
| CONNECT      | c | DATABASE                                                      |
| TEMPORARY    | T | DATABASE                                                      |
| EXECUTE      | X | FUNCTION, PROCEDURE                                           |
| USAGE        | U | DOMAIN, FDW, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE |
| SET          | s | PARAMETER                                                     |
| ALTER SYSTEM | A | PARAMETER                                                     |
| MAINTAIN     | m | TABLE                                                         |



## Rights to objects

When an object is created, it is assigned an owner. The owner is the role whose permissions were used to create the object. This can be **current\_user** - the current role in which the session is running or an inherited role (which was given with WITH INHERIT true). For most object types, by default, the owner and superusers have permissions to the created object. For example, the right to delete this object.

The right to modify or delete an object is an inalienable right of the object owner and cannot be taken away or transferred. This right, like others, is inherited by roles that have been GRANT the owner role. The owner of an object can be changed. This can be done by the superuser or the current owner of the object using the ALTER command, but only if the owner can switch to the new owner role. Example:

```
postgres=# alter database demo owner to bob;
ALTER DATABASE
postgres=# alter database demo owner to public;
ERROR: role "public" does not exist
postgres=# revoke ALL on database demo from public;
REVOKE
postgres=# revoke connect on database demo from public;
REVOKE
```

REVOKE Team **does not generate an error** if there was no privilege being revoked.

The pseudo-role `public` cannot be assigned as the owner of a database.

The owner of an object can revoke the rights to his object. However, the owner can manage the rights and grant himself the rights again.

To allow other roles to use an object, you must grant them rights to that specific object ("object privileges").

Rights are granted (presented) and revoked by the GRANT and REVOKE commands.

Each type of object (database, table space, configuration parameter, table, function, sequence, etc.) has its own set of rights.

The keywords used in the GRANT and REVOKE commands are: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, USAGE, SET, ALTER SYSTEM, MAINTAIN .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/ddl-priv.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/ddl-priv.html)

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-grant.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-grant.html)

# Viewing object permissions in psql

- The rights are displayed as a list:
- to\_who\_was\_given= `privileges` /who\_gave
- If there is nothing before the "=" sign, it means `public` (available to everyone)
- The " \* " after the letter means that the right is granted `WITH GRANT OPTION` .
- The " + " at the end indicates that this is not the last item and the list continues on the next line.

```
postgres=# \l
 List of databases
 Name | Owner | Access privileges
-----+-----+-----
demo | postgres | postgres=Ctc/postgres+
 | | alice=C*c/postgres
postgres=# GRANT ALL PRIVILEGES ON ALL TABLES IN
SCHEMA public TO alice, bob WITH GRANT OPTION;
GRANT
```

| Object type             | All rights | PUBLIC Rights | psql command |
|-------------------------|------------|---------------|--------------|
| DATABASE                | Ctc        | Tc            | \l           |
| DOMAIN                  | U          | U             | \dD+         |
| FUNCTION ,<br>PROCEDURE | X          | X             | \df+         |
| FDW                     | U          | No            | \dew+        |
| FOREIGN SERVER          | U          | No            | \des+        |
| LANGUAGE                | U          | U             | \dL+         |
| LARGE OBJECT            | rw         | No            | \dl+         |
| PARAMETER               | sA         | No            | \dconfig+    |
| SCHEMA                  | UC         | No            | \dn+         |
| SEQUENCE                | rwU        | No            | \dp          |
| TABLE (and similar)     | arwdDxtm   | No            | \dp          |
| COLUMN                  | arwx       | No            | \dp          |
| TABLESPACE              | C          | No            | \db+         |
| TYPE                    | U          | U             | \dT+         |



## Viewing object permissions in psql

The list of psql commands is given in the table on the slide. For example, for databases:

```
postgres=# \l
```

```
 List of databases
 Name | Owner | Encoding | .. | Access privileges
-----+-----+-----+---+-----
demo | postgres | UTF8 | .. | postgres=Ctc/postgres+
 | | | .. | alice=C*c/postgres
```

The rights are displayed as a list of elements ("aclitem"), where each element represents:

to\_who\_was\_given = `privileges` / `who_gave`

If there is nothing before the "=" sign, it means `public` - available to everyone.

The " \* " **after** the letter means that the right is granted **WITH GRANT OPTION** .

The " + " at the end indicates that this is not the last item and the list continues on the next line.

Example of granting privileges:

```
postgres=# GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO alice, bob
WITH GRANT OPTION GRANTED BY postgres;
```

**GRANTED BY** , since only the current user can be specified. Granting a privilege on behalf of another user is not implemented and is present for compatibility with the SQL standard.

**WITH GRANT OPTION** gives the recipient role the right to grant the received rights to other roles. The pseudo-role `public` cannot be granted rights with `GRANT OPTION` .

There is no right to delete an object (DROP), since it cannot be revoked or given; it belongs to the role that owns the object.

`ALL PRIVILEGES` or `ALL` for short means that all privileges allowed for the object type are granted.

**public** pseudo-role is given privileges by default on databases (Temporary - create temporary tables and other temporary objects , connect - connect ), routines (eXecute - execute ) , languages (Usage - create routines), data types, domains at the time of object creation.

# DEFAULT PRIVILEGES

- `ALTER DEFAULT PRIVILEGES` allows you to set the privileges that apply to objects that will be created in the future for schemas, tables, views, external tables, sequences, routines, types (includes domains)

- роль `public` получает права:

- › `CONNECT` and `TEMPORARY` (creating temporary tables) for databases
- › `EXECUTE` for functions and procedures
- › `USAGE` for languages, data types, domains

```
alter default privileges REVOKE ALL on routines from public;
alter default privileges REVOKE ALL on types from public;
revoke all on database demo from public;
revoke connect on database p2 from public;
\l
List of databases
Name | Owner | .. | Access privileges
-----+-----+---+-----
demo | postgres | .. | postgres=CTc/postgres+
 | | .. | alice=c/postgres
p2 | postgres | .. | =T/postgres +
 | | .. | postgres=CTc/postgres
revoke all on language plpgsql from public;
\dL+
 List of languages
Name | Owner | Trusted | .. | Access privileges
-----+-----+---+-----
plpgsql | postgres | t | .. | postgres=U/postgres
(1 row)
```



## DEFAULT PRIVILEGES

`ALTER DEFAULT PRIVILEGES` command allows you to set the rights that apply to objects that will be created in the future. The command does not change the rights assigned to existing objects. You can set **DEFAULT PRIVILEGES** for schemas, tables, views, external tables, sequences, routines, types (including domains). You cannot set **DEFAULT PRIVILEGES** for functions and procedures separately: `FUNCTIONS` and `ROUTINES` are considered equivalent for the command.

Recall that the `public` role receives the following rights: `CONNECT` and `TEMPORARY` (creation of temporary tables) for databases; `EXECUTE` for functions and procedures; `USAGE` for languages, data types, and domains. The owner of the object can revoke (`REVOKE`) these rights. It is more convenient to use the `ALTER DEFAULT PRIVILEGES` command to automatically execute the `REVOKE` command, which revokes privileges from the `public` role immediately after the creation of a subroutine and type (applies to domains):

```
alter default privileges REVOKE ALL on routines from public;
alter default privileges REVOKE ALL on types from public;
```

The `ALTER DEFAULT PRIVILEGES` command can perform not only the `revoke` command, but also the granting of privileges when creating an object.

To revoke privileges on databases and languages, you will have to use the `REVOKE` command:

```
revoke all on database demo from public;
revoke connect on database p2 from public;
```

```
\lList of databases
Name | Owner | .. | Access privileges
-----+-----+---+-----
demo | postgres | .. | postgres=CTc/postgres+
 | | .. | alice=c/postgres
p2 | postgres | .. | =T/postgres +
 | | .. | postgres=CTc/postgres
revoke all on language plpgsql from public;
\dL+
 List of languages
Name | Owner | Trusted | .. | Access privileges
-----+-----+---+-----
plpgsql | postgres | t | .. | postgres=U/postgres
```

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-alterdefaultprivileges.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-alterdefaultprivileges.html)

# Row-level security (RLS)

- disabled by default
- RLS is not mandatory access control (MAC)
- If RLS is enabled and there are no permitting policies, then access is denied
- policies are created with the CREATE POLICY command, for example:  

```
CREATE POLICY name ON table AS PERMISSIVE FOR ALL TO role USING (predicate);
```

  - › There can be several policies on the table, they can be PERMISSIVE and/or RESTRICTIVE and can be combined with AND and OR
- RLS is enabled at the table level with the command:  

```
ALTER TABLE name [ENABLE | DISABLE | FORCE |NO FORCE] ROW LEVEL SECURITY;
```
- enable with ENABLE option:
  - › RLS applies to everyone except the owner and roles with the SUPERUSER or BYPASSRLS attribute.
- RLS does not apply to integrity constraint checks.
  - › integrity constraints cannot be violated



## Защита на уровне строк (Row-level security, RLS)

Row-level security is disabled by default. Specifies a predicate (condition) by which access to rows is restricted for users. In Oracle Database, a similar option is called Fine-grained access control (FGAC), controlled by a package of procedures called DBMS\_RLS. This is one of the "options for options", since similar functionality can be implemented using views, which is simpler and more efficient. RLS is not Mandatory access control (MAC), which restricts access using labels on each row. In Oracle Database, an option similar to MAC is called Label Security, which was introduced in 1998 in version 8i. MAC does not add functionality and degrades performance, and is used where formal data "protection" requirements must be implemented. RLS and MAC act in addition to regular access rights (Discretionary access control, DAC). If regular access rights to the schema and table are not present, then there will be no access to the table.

First, policies are created using the CREATE POLICY command. For example:

```
CREATE POLICY name ON table AS PERMISSIVE FOR ALL TO role USING (predicate);
```

The functions in the predicate are executed with the rights of the user executing the query.

There can be multiple policies, they can be PERMISSIVE and/or RESTRICTIVE and can be combined with AND and OR.

Next, RLS is enabled at the table level with the command:

```
ALTER TABLE name [ENABLE | DISABLE | FORCE |NO FORCE] ROW LEVEL SECURITY;
```

You can use the wildcard character "\*" in the table name.

If RLS is enabled with the ENABLE option, then RLS applies to everyone except the owner and roles with the SUPERUSER or BYPASSRLS attribute. If RLS is enabled with the FORCE option, then RLS also applies to the table owner. If RLS is enabled and there are no permitting policies, then access is denied.

RLS does not apply to integrity constraint checks. This means that there are indirect ways to check for the existence of a row. For example, you could try to insert a duplicate value into a column that forms a primary key. If an error occurs, you can infer that the row exists.

A complex structure of policies and access rights violates the principle of security: ease of use by administrators. Complex structures create a false impression of security and increase the likelihood of errors that create gaps in protection.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-createpolicy.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createpolicy.html)

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/ddl-rowsecurity.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/ddl-rowsecurity.html)



# Connecting to an instance

- Initial authentication parameters are set in two files `pg_hba.conf` and `pg_ident.conf`
- Files are edited manually
- `pg_hba.conf` file can be viewed in the `pg_hba_file_rules` view
- The location of the files can be viewed using the `hba_file` and `ident_file` configuration parameters:

```
postgres=# \dconfig *_file
```

| List of configuration parameters      |  |                                                                       |
|---------------------------------------|--|-----------------------------------------------------------------------|
| Parameter                             |  | Value                                                                 |
| -----                                 |  |                                                                       |
| <code>config_file</code>              |  | <code>/var/lib/postgresql/tantor-se-1c-17/data/postgresql.conf</code> |
| <code>enable_delayed_temp_file</code> |  | <code>off</code>                                                      |
| <code>external_pid_file</code>        |  |                                                                       |
| <code>hba_file</code>                 |  | <code>/var/lib/postgresql/tantor-se-1c-17/data/pg_hba.conf</code>     |
| <code>ident_file</code>               |  | <code>/var/lib/postgresql/tantor-se-1c-17/data/pg_ident.conf</code>   |



## Connecting to an instance

When connecting to an instance initially, the client is authenticated. The initial authentication parameters are set in two text files , `pg_hba.conf` (host-based authentication) and `pg_ident.conf` (identification, user name mapping file).

The location of the files can be viewed using the `hba_file` and `ident_file` configuration parameters:

```
postgres=# \dconfig *_file
```

| List of configuration parameters      |  |                                                                       |
|---------------------------------------|--|-----------------------------------------------------------------------|
| Parameter                             |  | Value                                                                 |
| -----                                 |  |                                                                       |
| <code>config_file</code>              |  | <code>/var/lib/postgresql/tantor-se-1c-17/data/postgresql.conf</code> |
| <code>enable_delayed_temp_file</code> |  | <code>off</code>                                                      |
| <code>external_pid_file</code>        |  |                                                                       |
| <code>hba_file</code>                 |  | <code>/var/lib/postgresql/tantor-se-1c-17/data/pg_hba.conf</code>     |
| <code>ident_file</code>               |  | <code>/var/lib/postgresql/tantor-se-1c-17/data/pg_ident.conf</code>   |
| <code>ssl_ca_file</code>              |  |                                                                       |
| <code>ssl_cert_file</code>            |  | <code>server.crt</code>                                               |
| <code>ssl_crl_file</code>             |  |                                                                       |
| <code>ssl_dh_params_file</code>       |  |                                                                       |
| <code>ssl_key_file</code>             |  | <code>server.key</code>                                               |
| (10 rows)                             |  |                                                                       |

By default, the files are located in PGDATA and are created when the cluster is created.

**Files are edited manually** , there are no commands for editing them.

view the contents of the `pg_hba.conf` file in the `pg_hba_file_rules` view , which displays the current contents of the file. The view is useful for checking if there are any typos in the file. If the error column is not empty, then there is an error in the file line.

For changes in `pg_hba.conf` and `pg_ident.conf` to take effect, you need to reread the configuration, for example, with the function

```
select pg_reload_conf();
```

Files can include the contents of other files using the `include`, `include_if_exists`, `include_dir` directives . For example:

```
include_dir /var/lib/postgresql/tantor-se-1c-17/direcrory
```

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/client-authentication.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/client-authentication.html)

# pg\_hba.conf file

- File format - one record per line
- A record consists of several fields separated by spaces and/or tabs.
- Field contents may be enclosed in double quotes.
- entries (lines) closer to the beginning of the file prevail, unlike parameter files (postgresql.conf)
- The current contents of the file can be viewed through the view:

```
postgres=# select rule_number r, right(file_name, 11) file_name, line_number l, type, database, user_name
user, address, left(netmask, 15) netmask, auth_method auth, options opt, error from pg_hba_file_rules;
 r | file_name | l | type | database | user | address | netmask | auth | opt | error
-----+-----+---+-----+-----+-----+-----+-----+-----+-----+-----
 1 | pg_hba.conf | 117 | local | {all} | {all} | | | trust | |
 2 | pg_hba.conf | 119 | host | {all} | {all} | 127.0.0.1 | 255.255.255.255 | trust | |
 3 | pg_hba.conf | 121 | host | {all} | {all} | ::1 | ffff:ffff:ffff: | trust | |
 4 | pg_hba.conf | 124 | local | {replication} | {all} | | | trust | |
 5 | pg_hba.conf | 125 | host | {replication} | {all} | 127.0.0.1 | 255.255.255.255 | trust | |
 6 | pg_hba.conf | 126 | host | {replication} | {all} | ::1 | ffff:ffff:ffff: | trust | |
(6 rows)
```



## pg\_hba.conf file

The file format is one record per line. Comments begin with the " # " symbol, empty lines are ignored. A record can be continued on the next line by ending the line with the " \ " symbol (escape the carriage return symbol \r). A record consists of several fields separated by spaces and/or tabs . Field contents can be enclosed in double quotes.

Records are scanned from the beginning of the file to the end, **records (lines) closer to the beginning of the file take precedence** : if the connection details fall under a record (line), then that line determines the action, and subsequent lines are not scanned.

The PostgreSQL configuration files can include the contents of other files using the `include`, `include_if_exists`, and `include_dir` directives . The path to a file or directory can be specified as absolute or relative, and can be enclosed in double quotes. For the `include_dir` directive , the contents of all files in the directory whose names do not begin with a period and end with **.conf will be included**.

The `include_dir` is ambiguous. The order of entries is important. If there are multiple files in a directory, the first file included will prevail. Files are included according to the C sorting rules: numbers come before letters, and uppercase letters come before lowercase letters. The `pg_hba_file_rules` view allows you to see the exact order of entries.

To connect to a database, a user needs permissions in `pg_hba.conf` and `CONNECT` privilege for the database. Instead of listing user names in the file, it is easier to use `CONNECT` privilege for the database, so as not to bloat the contents of the file. PostgreSQL has an inconvenience for administrators: by default, `CONNECT` privilege is given to all users ( `public` ) and it is not possible to disable it using `DEFAULT PRIVILEGES`.

```
select rule_number r, right(file_name, 11) file_name, line_number l, type, database, user_name
user, address, left(netmask, 15) netmask, auth_method auth, options opt, error from
pg_hba_file_rules;
```

| r | file_name   | l   | type  | database | user  | address   | netmask         | auth  | opt | error |
|---|-------------|-----|-------|----------|-------|-----------|-----------------|-------|-----|-------|
| 1 | pg_hba.conf | 117 | local | {all}    | {all} |           |                 | trust |     |       |
| 2 | pg_hba.conf | 119 | host  | {all}    | {all} | 127.0.0.1 | 255.255.255.255 | trust |     |       |

The view displays the contents of the file at the time the request is executed, and the file may not yet be applied (re-read). The curly brackets are an array.



# Contents of pg\_hba.conf

- **тип соединения:**
  - > local - соединение через UNIX-сокеты
  - > host - any (with or without encryption) connections via TCP/IP
- **Database name:**
  - > all - all bases
- **replication - connection via physical replication protocol**
- **IP address with or without subnet mask:**
  - > all - all IPv4 and IPv6 addresses
  - > samehost - from the IP addresses of the host on which the instance is running

```
postgres@tantor:~$ tail $PGDATA/pg_hba.conf
TYPE DATABASE USER IP-ADDRESS IP-MASK METHOD
local sameuser all peer
local samerole all md5
host all all 127.0.0.1/32 trust
host db1,db2,@file1 +bob,@file2 10.0.0.0 255.0.0.0 scram-sha-256
host "/^db\d{2,4}$" /^.*desk$ samehost ident map=omicron
host all all all reject
```



## Contents of pg\_hba.conf

The entries in the file contain:

### 1) connection type:

**local** - "local" (from the same node) connection via UNIX socket. If you change the permissions on the socket file, you can restrict access to the instance from local users of the operating system using the `unix_socket_permissions` and `unix_socket_group` configuration parameters .

**host** - any ( with or without encryption ) connections via TCP/IP. Variations: `hostnossl`, `hostssl`, `hostgssenc` (gss = kerberos, with encryption), `hostnogssenc` (kerberos without encryption).

### 2) database name:

**all** - all bases

**sameuser** - the database name matches the name of the role with which the connection will be established

**samerole** ( `samegroup` ) - the database name matches the name of one of the granted roles

**replication** - connection via physical replication protocol (but not logical), the database name is not specified via physical replication protocol

Database and user names can be specified separated by commas. If the name begins with a slash, then the regular expression follows the slash. Names can be enclosed in double quotes. If the name begins with the " @ " symbol, then it is followed by the file name, the contents of which are substituted in this place. Several regular expressions and/or names can be specified, separating them with commas.

### 3) user name (roles):

**all** - any name

**+role** - the plus symbol means any users who have the specified role

### 4) IP address:

for local is absent, for other connection types IPv4, IPv6, CIDR IPv4 are specified (the number of bits in the network mask is separated by a slash)

**all** - all IPv4 and IPv6 addresses

**0.0.0.0/0** - all IPv4 addresses

**:::0/0** - all IPv6 addresses

**samehost** - from the IP addresses of the host on which the instance is running

## Contents of pg\_hba.conf (continued)

- authentication method:
- **trust** - establish a connection without checks, including a password
- **reject** - unconditional refusal of connection
- **peer** - the client operating system user name must match the role name, there is an optional **map** parameter
- **scram-sha-256** or **md5** - check password
- **gss** - authentication via kerberos protocol
- **ldap** - authentication with ldap server
- **cert** - request an SSL certificate from the client

```
postgres@tantor:~$ tail $PGDATA/pg_hba.conf
TYPE DATABASE USER IP-ADDRESS IP-MASK METHOD
local sameuser all
local samerole all
host all all 127.0.0.1/32 trust
host db1,db2,@file1 +bob,@file2 10.0.0.0 255.0.0.0 scram-sha-256
hostssl "/^db\d{2,4}$" /^.*desk$ localhost cert
host all all 1.1.0.0/16 reject
```



## Contents of pg\_hba.conf (continued)

**samenet** - from the IP address in the subnet of the host on which the instance is running

The hostname or domain name can be specified, but is not recommended, as reverse name resolution will be used, which will lead to delays in establishing connections.

5) Authentication method if the connection matches the previous record fields:

**trust** - establish a connection without checks, including a password.

**reject** - unconditional refusal of connection

**peer** - only for connections via UNIX socket. The client operating system user name must match the name of the cluster role under which the connection is established. There is an optional parameter **map**

**scram-sha-256** - checks a password that should be stored as a **scram-sha-256** hash

**md5** - checks the password, which should be stored as a **scram-sha-256** or **md5** hash

**password** - should not be used, as the password will be transmitted in clear text

**gss** - authentication via kerberos protocol. There are parameters **map**, **krb\_realm**, **include\_realm** .

**ldap** - authentication by ldap server. There are 13 parameters and two bind modes.

**cert** - request an SSL certificate from the client, by default the role should match the CN, but this can be overridden by the optional **map** parameter .

**radius** , **pam** , **ident** can also be used .

6) Authentication parameters (optional). Parameters are specific to authentication methods and are specified in the format **parameter=value**.

**map** parameter refers to a line in the **pg\_ident.conf** file .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/gauth-pg-hba-conf.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/gauth-pg-hba-conf.html)

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/client-authentication.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/client-authentication.html)

# pg\_ident.conf name mapping file

- For peer, gss, ident methods, you can map the name returned by the authentication service to the cluster role under which the client wants to establish a session.
- **MAPNAME** names are referenced by the `map=map1` parameter in the `pg_hba.conf` file entries
- `pg_ident_file_mappings` view allows you to view the current contents of a file:

```
postgres=# select map_number r, right(file_name, 13) file_name, line_number l,
map_name, sys_name, pg_username, error from pg_ident_file_mappings;
 r | file_name | l | map_name | sys_name | pg_username | error
-----+-----+-----+-----+-----+-----+-----
 1 | pg_ident.conf | 73 | map1 | astra | postgres |
 2 | pg_ident.conf | 75 | map1 | astra | alice |
(2 rows)
postgres=# \! tail -n 4 $PGDATA/pg_ident.conf
MAPNAME SYSTEM-USERNAME PG-USERNAME
map1 astra postgres
astra can also connect with the alice role
map1 astra alice
```



## pg\_ident.conf name mapping file

For the peer, gss, ident methods , you can map the name returned by the authentication service to the cluster role under which the client wants to establish a session.

`pg_ident_file_mappings` view allows you to view the current contents of a file:

```
postgres=# select map_number r, right(file_name, 13) file_name, line_number l,
map_name, sys_name, pg_username, error from pg_ident_file_mappings;
```

```
 r | file_name | l | map_name | sys_name | pg_username | error
-----+-----+-----+-----+-----+-----+-----
 1 | pg_ident.conf | 73 | map1 | astra | postgres |
 2 | pg_ident.conf | 75 | map1 | astra | alice |
(2 rows)
```

```
postgres=# \! tail -n 4 $PGDATA/pg_ident.conf
MAPNAME SYSTEM-USERNAME PG-USERNAME
map1 astra postgres
astra can also connect with the alice role
map1 astra alice
```

**MAPNAMEs** are referenced by the `map=map1` parameter in `pg_hba.conf` file entries.

The view displays the contents of the file at the time the query is executed, and the file may not yet be applied (re-read). For changes in `pg_hba.conf` and `pg_ident.conf` to take effect, you need to re-read the configuration, for example, with the function

```
select pg_reload_conf();
```

The file can include the contents of other files using the `include`, `include_if_exists`, `include_dir` directives .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/auth-username-maps.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/auth-username-maps.html)

# Practice

1. Create a new role
2. Setting attributes
3. Create a group role
4. Creating a diagram and table
5. Granting a table access role
6. Deleting created objects
7. Location of configuration files
8. View authentication rules
9. Local changes for authentication
10. Checking the correctness of the settings
11. Cleaning up unnecessary objects



## Practice

Create a new role  
Setting attributes  
Create a group role  
Creating a diagram and table  
Granting a table access role  
Deleting created objects  
Location of configuration files  
View authentication rules  
Local changes for authentication  
Checking the correctness of the settings  
Cleaning up unnecessary objects



7a

## Physical backup



# Types of backups

- Hot - no downtime, no instance shutdown, no interruption of client sessions
- Cold - on a correctly (with final checkpoint) stopped instance
- Autonomous or self-sufficient



## Types of backups

A database cluster physically consists of files in a file system. An instance does not duplicate files, all files are stored without duplicates. Losing any file can lead to data loss, which is usually not acceptable.

Files can be lost or damaged for various reasons. For example, an intruder or a program ("computer virus") can erase cluster files. Disk mirroring will not help in this case. PostgreSQL has many backup methods. The most optimal in terms of simplicity, cost, and fault tolerance for a typical cluster is to use physical replication, which we will consider in a separate chapter.

Backup copies can be:

1) Hot - without stopping the instance.

2) Cold - if the instance is stopped correctly before the backup. Stopping the instance and performing the backup means downtime, which is usually undesirable. However, it can occur in practice. The cluster uses one or more mount points. If the file system supports snapshots, then on a stopped cluster they can be made from mount points in a short time. The cluster downtime will be small.

3) Self contained. A set of files that is sufficient to start an instance and provide access to the data image at the time the backup was made. Such copies may be created periodically (e.g. quarterly) and stored (retained) for a certain period of time.

For hot physical backups, the concept of a consistent state means that the copy has journal data at the time the backup ended. Cold backups are correctly considered consistent if the instance has been stopped. A hot backup can be made consistent by rolling journal files (WAL logs) onto it before the backup ended.

In either case, when a consistent copy starts, it will look for a log file containing a record of the checkpoint pointed to by the control file (or `backup_label` file if present). If the log file is missing, the instance will not start. Consistency only makes the instance start faster.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/backup-file.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/backup-file.html)

# Incremental backups

- can be useful if the cluster size is large
- increases the complexity of backup procedures, the likelihood of error and the inability to restore the cluster
- Procedure:
  - First, a full backup is created using the `pg_basebackup` utility
  - later, incremental backups are created using the `pg_basebackup` utility with the `-i manifest` or `--incremental=manifest` parameter
  - For recovery, incremental backups are superimposed on the full backup using the `pg_combinebackup` utility
- when creating incremental backups, the files in the `PGDATA/pg_wal/summaries` directory are used



## Incremental backups

Incremental backup is justified when a regular backup takes a long time or creates an increased load on input/output. A long time can be considered, for example, if the backup does not have time to complete during non-working hours (overnight). The idea of incremental backup is that a full backup is created once and based on the fact that a small part of the data file blocks has time to change in a day (or the desired intervals between backups), it is enough to backup only the changed blocks.

Starting with version 17, the backup utility `pg_basebackup` is called with the `-i manifest` parameter or `--incremental= manifest` can create incremental backups. The parameter must specify the path to the **manifest file the backup relative to which the incremental backup will be created**. Both of these backups form a link in the backup chain.

To determine which blocks should be included in the incremental backup, the utility uses WAL summaries, which are stored in the `PGDATA/pg_wal/summaries` directory. If the files are missing, the incremental backup is not created and the utility returns an error.

A full backup is made once, and then an incremental backup is made once a day (or at another frequency). The frequency is selected based on the volume of log files generated by the cluster during this time. The volume of log files does not depend on the volume of the cluster. After creating any (and incremental) copy, log files can be deleted, freeing up space.

The disadvantage of incremental backup is the **greater** complexity of the procedures. The greater the complexity, the greater the likelihood of making a mistake during the recovery process and not being able to restore the cluster.

Using an incremental backup means overlaying it with the `pg_combinebackup` utility on a full backup. You can overlay several incremental backups on a full backup. `pg_combinebackup` checks that the backups specified to it form a valid chain.

It is not recommended to disable checksum calculation, especially when using incremental backups.

# What is reserved?

- PGDATA directory
- tablespace directories
- Files known to pg\_basebackup as temporary are not backed up.



## What is reserved?

The cluster contains:

1) data files - binary files with a block size of 8Kb. They contain data of database objects, system catalog and service files: control file, `xact ( clog )` files, multitransactions, subtransactions and others.

2) WAL or "write-ahead log" or simply "log" (legacy `xlog` ). The word "log" can also be used to refer to the text files of an instance's messages. In this chapter, we will use it to refer to the write-ahead log. It consists of files of 16 MB by default. The log records changes to blocks of data files and, at checkpoint frequency, can write complete images of the data blocks that have changed since the previous checkpoint.

3) text parameter files and other binary and text files and directories located in `PGDATA` that are not recognized by backup utilities as temporary.

The logic of backup is as follows. A copy of data files and other files is made. This can be done using operating system utilities, but this is inconvenient, since it is necessary to provide for a checkpoint and synchronization of cluster files or file systems (`sync`) on which the cluster files are located. It is worth using the `pg_basebackup` utility or third-party utilities that automate both backup and recovery. Then the logs created from the moment the checkpoint is completed before the backup and until the very last moment while the instance is running are saved or transferred to a safe place (if you want to recover to the last moment, and this is usually a necessary requirement for any important data). In this way, "cluster file backups" and "log archive" are created.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/continuous-archiving.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/continuous-archiving.html)



# Recovery procedure

- Stop instance
- Restore its contents from a full backup
- Apply incremental backups if they were created
- Launch instance



## Recovery procedure

Recovery:

1) detection of corruption. The instance may be terminated abnormally and the attempt to start it will fail. Or it may continue to work, but generate errors when accessing some of the data. If the instance has not been terminated, it will need to be terminated in emergency mode, since it is unlikely to be terminated correctly.

2) Restore everything from the backup. If the cluster size is large, you can try to restore only those files that are different. The only guarantee is calculating the checksums of each file and comparing them. If the recovery utility has such modes, you can use them. Manual use, for example, `rsync` (in the checksum calculation mode) can increase the recovery time, since it adds time for planning and entering recovery commands ("think time"). It is optimal to have command files and simple instructions in case of failures, this will reduce the recovery time by eliminating time for thinking and possible errors.

3) Start the instance. The startup process, having found the `backup_label` file in the PGDATA root, will start restoring (rolling forward logs) from the LSN specified in it, and not in the `pg_control` file. Files are applied in the strict order of their creation by the instance, gaps (absence of a file or damage to blocks (they are recognized, since log blocks are protected by checksums) in its body are critical - they cannot be crossed. Therefore, logs can be called a "chain" of log files (WAL segments), since damage to a link in the chain breaks it. There are techniques for restoring with gaps, but you should not rely on them. If there are no gaps, but the record cannot be applied to the data file (the file is missing), then the roll forward also stops. Log records are applied to data blocks in a strict order. When applied, it is checked that the record can be applied to the data block. By default, journals contain full images of changed blocks ( `full_page_writes` ), and even if data blocks are damaged (torn "torn", split), they can be restored.

# Pre-Write Log Files

- are located in PGDATA/pg\_wal
- log files are not duplicated
- the size of the current log file is the same as the others
- server and background processes of the instance write to the log files



## Pre-Write Log Files

The presence of log files from the checkpoint that is initiated at the start of the backup is critical for recovery. How are log files stored?

The PGDATA/pg\_wal directory is where the cluster instance processes write to the log files. The files are not duplicated by the cluster instance. We can say that each WAL file in this directory is represented in one instance (not a cluster instance, but a file instance). At any given time, there is a current log file where the instance processes write (or the last file where they wrote, if the instance is terminated). The size of this file is always equal to the size of other files (16 MB by default) because when the next file is created, its size is immediately set. The size is set either (the `wal_init_zero` parameter ) by the command to write the last byte to a file of size `wal_segment_size` , or by commands to write empty blocks up to the size `wal_segment_size` . This is necessary to reserve space in the file system in advance and prevent the instance from running out of space, as well as for fault tolerance and speed: changing the file size is an operation with file system metadata. Depending on the mount settings, the file system can "log only metadata" (the word "log" refers to file systems, they also have logic to protect against power failure), and if the file size changes frequently (the file size in the file system is metadata), in the event of a power failure, the last blocks of the file will either be lost, or the speed of writing to the file will be low.

# LSN (Log Sequence Number)

- The log can be thought of as a concatenated set of log files (most of which have already been deleted) starting with the very first file generated when the cluster was created.
- LSN - 64-bit monotonically increasing number indicating the address (offset from the first byte of the first log file) accurate to a byte in the cluster log
- Instance processes write variable length log records, LSN is used to specify the address of the start of the log record
- physically, writing to files is done in 8Kb blocks (wal\_block\_size)



## LSN (Log Sequence Number)

The log files are written to by the instance processes in variable-length records. The address of each record is designated by a 64-bit "LSN" (Log Sequence Number), which is the log byte sequence number since the cluster was created (the time when writing to the log began). The LSN can be said to specify the "offset from the beginning of the log" or "position in the write-ahead log." The LSN can also be said to be a monotonically increasing integer that points to a log record.

LSN values are present in many places: data blocks, the control file, and the log records themselves. An LSN can be used to reconstruct the name of the log file that contains the record to which the LSN points.

The very first file has the name `00000001 00000000 000000 01`. The name consists of three parts of 8 characters. Each number is 32-bit, written in hexadecimal form. The maximum number is FFFFFFFF (32 ones in binary notation). The first number is the "time line" number (Time Line, TLI, time branch, incarnation). This number is increased by one when the cluster instance opens after the recovery procedure in order to prevent the log file from being erased, since recovery does not always fall on the 16-megabyte boundary of the log, the LSN to which the cluster was restored may point to a byte in the middle of the file. In this case, before recovery, the first half of the file contains the necessary records, the second - bytes with the value zero.

When the maximum values are reached, there is no LSN wrap and no timelines. The maximum LSN value is quite large: 16777216 terabytes.

Physically, writing to log files is done in 8-kilobyte blocks. The block size is specified by the `wal_block_size` configuration parameter, which is set when building PostgreSQL and does not change. Log records are protected by checksums.

The size of the log file (WAL segment), the size of the log block, `TimeLineID` are stored in the cluster control file (`pg_control`), so knowing the LSN, you can determine the name of the file that contains the variable-length record that the LSN points to.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/wal-internals.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/wal-internals.html)

# Log file names and LSNs

- LSN is represented as two 32-bit numbers written in hexadecimal (HEX) form, separated by a slash: `XXXXXXXX / YY ZZZZZZ`
- `XXXXXXXX` - "senior" 32 bits of LSN
- `YY` - the "higher" 8 bits of the "lower" 32-bit number
- `ZZZZZZ` - offset in the 16-megabyte log file relative to its beginning
- The log file size determines the maximum size of the log buffer (`wal_buffers`)
- `0000000N XXXXXXXX 000000 YY` - names of 16MB log files



## Log file names and LSNs

Let's take a closer look at LSN. You may have wondered why the default log file size is 16MB?

In text form, which is used in message files, command options, and functions, LSN is represented as two 32-bit numbers written in hexadecimal form (HEX), separated by a slash: `XXXXXXXX / YY ZZZZZZ`. `XXXXXXXX` is the "higher" 32 bits of LSN. If the log file size is 16 MB (the default value), then `YY` is the "higher" 8 bits of the "lower" 32-bit number. `ZZZZZZ` - offset in the 16-megabyte log file relative to its beginning. Leading zeros are not output: `00000001 / 0A 000FFF` will be output as `1 / A 000FFF`, which complicates perception.

The maximum log file size is 1 GB, the minimum is 1 MB, and can take values in powers of two (16, 32, 62, 128, 256, 512 1024 MB). For example, if you set the log file size to 256 MB, the LSN will look like `XXXXXXXX / Y ZZZZZZ`. If 1 MB (such a small size should not be used because `wal_buffers` will not be larger than 1 MB), then: `XXXXXXXX / YYY ZZZZZ`. Other file size values do not have such a clear division by digits. The log file size determines the maximum size of the log buffer in the shared memory of the instance, which is set by the `wal_buffers` parameter. By default, if the `shared_buffers` size is greater than 512 MB, the log buffer is set to the maximum value of 16 MB.

The size of the log files can be set when creating the cluster with the `initdb --wal-segsize=size` utility or after creating the cluster with the `pg_resetwal -wal-segsize=size` utility.

The names of the log files also depend on the file size. For a size of 16MB, the format is: `0000000N XXXXXXXX 000000 YY`. The second 8 characters are the most significant 32 bits of the LSN, then 6 zeros, then 2 characters of the most significant 8 bits of the least significant 32-bit number. For a size of 256MB, the format is: `00000001 XXXXXXXX 00000 YYY`. The first 8 characters are the number of the transition to a new timeline.

# Startup recovery process

- startup - the process of restoring cluster files using the journal
- when changing the timeline, text files `0000000N.history` with information about timelines are created in the `PGDATA/pg_wal` directory, they cannot be deleted and will be backed up
- a new timeline appears if a restore was performed or a replica became a master
- the purpose of timelines is to ensure that when you restore to a point in the past, new log files do not overwrite the old ones and that you can return to previous timelines



## Startup recovery process

During a full recovery, information about which log record was generated most recently before the instance crashed is not written anywhere. The recovery process reads the log records sequentially and if it sees that the next log record contains "garbage", it stops recovery. When reading the next record, the recovery process first looks for the location where the size of the log record should be. If there is an unrealistic value in this location, it stops recovery; if there is a real value, it looks for the location with the checksum. If the checksum does not match, it stops recovery. An example of a message in the instance message file:

```
LOG: invalid record length at CA/277E2A88: expected at least 26, got 0
```

The recovery process expected to see a number no less than 26 (the minimum size of a journal entry), but saw zeros.

The recovery process can form the file name because it knows the timeline number, log block size, file size, and LSN from the control file. When the timeline changes, text files `0000000 N.history` with information about timelines are created in the `PGDATA/pg_wal` directory. The log that was formed on the previous timeline does not contain data about the new line, because the instance of the previous timeline that formed the log file has stopped.

# Functions for working with logs

- `pg_switch_wal()` switches writing to a new WAL file
- `pg_create_restore_point('text')` creates a LSN log entry with a text label
- `pg_walfile_name('LSN')` returns the name of the WAL file that contains the LSN record.
- `pg_current_wal_flush_lsn()` LSN of the end of the last redo log record considered to be safely flushed
- `pg_wal_lsn_diff(LSN,LSN)` bytes between two LSNs



## Functions for working with logs

Let's look at the description of useful functions for working with logs. The documentation contains similar descriptions, but worded differently (it may even seem less clear).

`pg_switch_wal()` switches writing to a new WAL file, the old one is not written to, even though it has the same size as the other log files.

`pg_create_restore_point('text')` creates a log record LSN with a text label. The function returns the LSN of the beginning of this log record. The label can be specified in the `recovery_target_name` parameter to specify that the logs should be rolled back to the record with the label. If you create multiple labels with the same name, then the recovery will stop as soon as it encounters a record with this label.

`pg_walfile_name('LSN')` returns the name of the WAL file in which a record with the specified LSN should be found. The result is calculated based on the data in the control file.

`pg_walfile_name_offset(LSN)` shows not only the calculated file name, but also the offset in bytes relative to its beginning.

`pg_current_wal_lsn()` shows the LSN **of the last** byte ("end") of the last log record written to the current log file. Up to and including this LSN, processes in the operating system must see it written if they are to read the log file.

`pg_current_wal_flush_lsn()` LSN **of the last** byte of the last redo log record that is considered to be securely flushed (`fsync` or other method returned a result). Specifies the LSN up to and including which redo log records should be flushed after a power failure.

`pg_current_wal_insert_lsn()` LSN **of the last** byte of the last log record that instance processes wrote in the log buffer, and that log record may not have been written to disk yet. Used by instance processes to determine the LSN of their record that they will start writing.

The command-line utility `pg_waldump filename` can be used to obtain a list of the LSNs of the beginning of log records and their contents from a WAL file in text form.

`pg_lsn` is a data type. This data type has a cast of type `'literal'::pg_lsn`, a subtraction operator (or a function `pg_wal_lsn_diff(LSN,LSN)`), which can be used to get the difference in bytes between two LSNs - the volume of log data.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/functions-admin.html#FUNCTIONS-ADMIN-BACKUP](https://docs.tantorlabs.ru/tdb/en/17_5/se/functions-admin.html#FUNCTIONS-ADMIN-BACKUP)

# Cold backup

- the instance must be stopped gracefully
- `pg_basebackup` utility cannot backup cluster if instance is down
- The operating system utilities `cp`, `rsync`, `tar` are used
- Copies the entire PGDATA and tablespace directories
- an offline copy is created on which an instance can be run
- advantage - simplicity
- the main disadvantage is the interruption of service to cluster clients



## Cold backup

A cold backup is a backup on a correctly stopped cluster. The result is an autonomous copy of the cluster. Autonomous or self contained means that it contains all the files needed to start the instance and provide access to the data.

Backup technique: the location of PGDATA , the tablespace directory (symbolic links in PGDATA/`pg_tblspc` ), the instance is stopped correctly, it is checked that the instance processes are not left in memory and the found directories are copied by any utilities.

The features are described in detail in the documentation. For example, you can use file system snapshots if it allows or perform preliminary copying, and after stopping the cluster, update the files with the `rsync` utility in the checksum calculation mode. The main advantage of cold backup is that simplicity is lost. It is more practical to backup a running cluster, for example, with the `pg_basebackup` utility .

The created copy can be used with accumulated log files (log archive), for example, for a full recovery.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/backup-file.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/backup-file.html)



# Configuration parameter `full_page_writes`

- enabled by default
- protects against page torn
- It is not recommended to disable
- enabled for the duration of the backup using the `pg_basebackup` utility
- when replica is backed up, it must be enabled on the master



## Configuration parameter `full_page_writes`

By default, the `full_page_writes` parameter is enabled. This means that the entire contents (8 KB) of each data block ("page") are written to the log the first time this block is changed (changed in the buffer cache buffer) after each checkpoint. This is a large amount of data. Why is this amount of data needed? The block size is 8 KB, and the block size of an HDD, SSD, or file system is not always 8 KB, but more often 4 KB. In the event of a power failure, 4 KB may be written to a data block, and they are not duplicated, while the other 4 KB will not be written and will remain from the previous version of the block. Such a block is called split or "torn". The checksum in the block will not match and it will be considered damaged. The block may refer to a system catalog object and the instance may not start.

Why is the first page change written to the journal, but not the second? Because recovery starts at the LSN of the start of a checkpoint that was completed before the backup began. Entire block images are read from the journal into the buffer cache (not from the data file), and changes from the journal are applied to them. If the buffer cache is small, blocks are written to their places in the data files as needed. If a block were written on the second change, then during recovery the journal would first record the block change and it would be read from the data file, where it may be corrupted, the journal entry could not be applied, and recovery would stop with an error like:

PANIC: WAL contains references to invalid pages

The recovery process does not know that a page image may be encountered later in the log. For such errors, you can use `ignore_invalid_pages=on` (only if `full_page_writes` was enabled) in the hope that a full page image will be encountered later. If `full_page_writes=off`, then you should not use `ignore_invalid_pages`.

The presence of full page writes allows you to not depend on whether the operating system flushes modified pages of data files to disk from its cache, and this is a write cache, or on the order of block flushing. The operating system can work with 4K blocks in its cache and write them in any order. If at least somewhere on the way from the instance process to the disk sector (or SSD disk controller) a block smaller than 8K is used, then the probability of getting a large number of broken blocks when the power is lost or the operating system crashes is high and you should not disable the parameter.

When backing up from a replica, `full_page_writes` must be enabled on the master.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-wal.html#GUC-FULL-PAGE-WRITES](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-wal.html#GUC-FULL-PAGE-WRITES)



# Backup utility pg\_basebackup

- creates a backup copy of the cluster on a running instance
- It is not possible to create a copy of individual databases or database objects, only the entire cluster
- uses replication protocol
- uses a more secure backup mode "pull" - accepts files, rather than "push" - transfers files
- can create incremental backups



## Backup utility pg\_basebackup

Creates a backup copy of the entire cluster on a running instance. Does not stop service, does not block user sessions.

Creates a copy of the entire cluster. Cannot create a copy of individual databases or objects.

The utility connects to the instance via TCP or Unix socket. By default, it creates two connections via the replication protocol, which does not have regular SQL commands, but does have commands for retrieving files from the server file system. The first connection creates a backup, and the second connection starts transmitting log files in parallel.

To connect to an instance via the replication protocol, you can grant individual permissions to cluster roles. The utility is included in the standard delivery.

The utility creates a backup copy on the host it is running on by default. It can be run on a host other than the host where the instance is running. The cluster will be backed up via a network connection.

During the backup, enables `full_page_writes` if the parameter was disabled.

By default, creates a temporary replication slot. When backing up, you should use temporary or permanent replication slots so that while the backup is being created, the cluster does not delete the log files needed to restore this backup.

After the backup is completed, the utility switches the log or waits for it to be switched, and accepts the log on which the backup was completed. Only after receiving the log file on which the backup was completed is an autonomous backup obtained.

Can perform backup by connecting to an instance serving a physical replica (copy) of the cluster without loading the instance of the primary (master) cluster. This is called "backup offloading".

Since version 17, PostgreSQL can create incremental backups. A copy of the entire cluster is created; parts of the cluster (individual databases, table spaces) cannot be backed up. To monitor the backup process, there is a view called `pg_stat_progress_basebackup`.

Accepts files ("pull" mode) - this is **safer**, because if the host where the cluster is running is hacked, the attacker will not be able to connect to the host with backups. Before destroying the cluster, attackers first search for backups and erase them. The "push" mode (**when the backup host connects to the host with backups**) is **not safe**. When using backup utilities that work in this mode, after the backup, you need to isolate the host with backups from the network to avoid damage to the backups if the host where the backup cluster is running is hacked. This is why you should not use the `archive_command` parameter to transfer logs; instead, you should use the `pg_receivewal` utility, which works in "pull" mode, just like the `pg_basebackup` utility.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-pgbasebackup.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-pgbasebackup.html)

# pg\_verifybackup utility

- checks files in backups created by the `pg_basebackup utility`
- calculates file checksums (CRC32C) and compares them with the values in the `manifest_file`
- compares files with the list of files in the manifest file
- without manifest file it doesn't work
- Checks for the presence of log entries needed to synchronize backup files - backup autonomy
- does not check files `postgresql.auto.conf`, `standby.signal`, `recovery.signal`



## Утилита `pg_verifybackup`

Why use this utility? If you want to check that the files in the backup were not damaged during storage, and that the necessary log files for synchronization were received during the backup. The probability of not receiving them is low, so there is no need to check the backups after their creation. The utility does not provide guarantees, only test recovery and subsequent data unloading at the logical level (`pg_dump` and `pg_dumpall`) provide them.

By default, `pg_basebackup` creates a manifest file (`manifest_file`). This is a text file in json format with checksums for each backed up file using the CRC32C algorithm. The contents of the manifest itself are protected by a checksum using the SHA256 algorithm. There is no need to change the algorithms.

Should I disable the creation of the manifest file when backing up? No, I shouldn't. If I don't change the algorithms, there is no extra load.

If the manifest file exists and has not been deleted, the `pg_verifybackup utility` can be used to check that the files match the manifest, i.e. have not been damaged during storage. The utility produces a report on missing, modified, and added files. The utility also checks the self-sufficiency (autonomy) of the backup - whether it is possible to synchronize the backup copy at the time of backup completion using the log files (if you did not refuse to back them up). The check is performed using the `pg_waldump utility`, checking for the presence of the required log records in the backed up log files. The list of required records was transferred by the instance to the `pg_basebackup utility` during the backup and placed in the manifest file.

The utility does not check the files `postgresql.auto.conf`, `standby.signal`, `recovery.signal` and their presence.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-pgverifybackup.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-pgverifybackup.html)

# Magazine archive

- Methods for organizing a magazine archive:
- copying WAL segments with the command in the `archive_command` parameter
- Receiving a stream of log records and storing them using the `pg_receivewal` utility
- use of physical replicas
- using `wal-g` utility to transfer logs via S3 protocol



## Magazine archive

A backup can be offline. To be able to restore from it to the latest point in time, you will need to roll the log files onto this copy from the time the backup was created until the latest point. By default, the cluster stores log files in the `PGDATA/pg_wal` directory for the purpose of restoring the consistency of the cluster files after an instance crash, i.e., from the beginning of the last checkpoint. Log files can be retained for a long time by configuration parameters, but the `PGDATA/pg_wal` directory may not be the best place to store logs if it is an expensive storage device, or for protection against deletion by attackers. Backups and logs should be stored on a host that is not accessible by the backup host, if possible, to prevent an attacker from deleting the backups.

Methods for organizing a magazine archive:

- 1) the `archive_command='command'` parameter and `archive_mode=on` . This method has a drawback - the current log file (to which the instance processes write) will only start to be copied when the file is no longer current. If the current file is lost, then the data in it will be nowhere to be retrieved and transactions will be lost. This is unacceptable.
- 2) `pg_receivewal` utility . This utility can receive log data without delay. The disadvantage is that you need to automate the launch of the utility and restart it in case of failure.
- 3) use third-party extensions that automate backups. For example, the `wal-g` utility, which can transfer log files via the S3 protocol.
- 4) use physical replicas. The `wal_receiver` process of a physical replica instance operates according to the redundancy protocol and the same logic as the `pg_receivewal` utility .

# No loss (Durability)

- The redundancy architecture must ensure full recovery of committed transactions.
- The backup architecture must protect against deletion of backups in case of hacking of the host with the cluster instance: the initiator of backups must be a host different from the one on which the cluster instance is running
- The current log file should not be a single point of failure.
- Using `pg_receivewal` or a physical replica is a way to ensure that there is no loss if the current log gets corrupted



## No loss (Durability)

Logs can be retrieved from the "archive", but it is important for full recovery to roll forward records from the most recent log file, which may not have had time to be transferred to the archive. Losing even one committed transaction is usually unacceptable (Durability property from the ACID transaction properties). Log archives do not guarantee that they contain all transactions, and the last log on the disk of a damaged cluster may not be saved, for example, as a result of a disaster (fire, flood, destruction of the building where the file storage systems are located). The log file in the `PGDATA/pg_wal` directory should not be a "failure point". Using `pg_receivewal` and/or a physical replica with confirmation of transaction commit will ensure that transactions are not lost in the event of a complete loss of the cluster host with all disk systems (disaster).

Commit confirmation of transactions is configured by the `synchronous_commit` and `synchronous_standby_names` parameters .

Mounting `pg_wal` on redundant storage systems can protect against disk failure, but will not protect against an attacker who can erase the log file. In the latter case, one may ask: should I keep archives or hold files in `pg_wal`? Technically, it is more convenient to keep archives than to configure file retention in `pg_wal`. Also, copying to archives frees up space on the expensive high-speed device where `pg_wal` is located . It is also worth considering that for security purposes, the cluster host must not have access to backups and log archives. If an attacker gains access to the cluster host, the first thing the attackers do is delete all backups. The hosts where the backups are stored should be physically disconnected from the network (at the hardware level, network ports) after performing the backup so that in case of full access to the software systems, the attacker cannot erase the backups and it would be possible to recover.

Is a physical replica enough? If the master host is unstable, there is a theoretical possibility that `walsender` will push a corrupted log record to the replica. Such a record could theoretically corrupt the replica. To protect against this, you can use a replica with delayed (for several hours) application of log records. The delay in application is set by the `recovery_min_apply_delay` configuration parameter on the replica.

# pg\_receivewal utility

- connects via replication protocol
- accepts journal entries without delay
- It is recommended to use the replication slot
- can receive logs from both the master and the physical replica
- scripts for autostart ( **systemd** ) are missing
- can compress log records
- the advantage is that it is the initiator of the connection, which allows for security
- can commit transactions in synchronous commit mode
- gives the current log file the suffix **.partial**



## pg\_receivewal utility

`pg_receivewal` utility connects via the replication protocol and receives a stream of log records as they are generated on the instance and stores the received records in files. The file names and sizes are the same as those generated by the instance. The utility calls the current file as `name.partial` to avoid confusion.

`pg_receivewal` accumulates log data in memory by default and saves it to a file when the file is closed. If you want the utility to write received data without delay, you need to run the utility with the `--synchronous` parameter . The same mode should be used if the utility will commit transactions in the synchronous commit mode set by the `synchronous_commit` parameter. This parameter determines after completing which level of log record processing the process will issue a COMMIT COMPLETE message to the client. Values:

`remote_apply` - not applicable to `pg_receivewal` , only physical replicas can commit a transaction. It is not worth setting, since the transaction commit speed drops sharply.

`on` - the default value. The transaction is committed after `pg_receivewal` or the replica receives a response from its operating system that it has written the log pages to disk (performed `fsync`)

`remote_write` - the `pg_recievewal` or `wal receiver` process of the replica has sent a command to its operating system to write log blocks to disk. The operating system may hold them in its file system cache, and if the power goes out, the blocks may be lost. This value is a reasonable choice if the probability of failure of the primary host and then the backup is small, and the `on` value leads to performance degradation that cannot be eliminated by other means (for example, the `commit_siblings` parameter or replacing `fsync` on the `pg_recievewal` side )

`local` - the transaction is committed after writing to the local log file and `fsync` (the default method)

`off` - should not be set at the cluster level. Can be set by application developers at the session or transaction level.

If `synchronous_standby_names` is not set, then it is equivalent to `local` and the current log is the single point of failure.

The utility can compress the saved logs.

It is recommended to use a replication slot. Without a replication slot, the utility may not receive some of the log files upon restart, in which case it will not be possible to go through the loss during recovery. The absence of gaps in log records is important. When using a replication slot, the utility will request missing log files after restart.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-pgreceivewal.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-pgreceivewal.html)

# Replication slot

- three types: physical, temporary physical, logical
- are used to hold log files
- If the client does not accept log data (stopped), then the log files will be held and fill all the free space in the `PGDATA/pg_wal` directory . To prevent this from happening, you should set a limit with the **`max_slot_wal_keep_size` parameter**
- **`pg_replication_slots`** view contains a list of all replication slots.



## Replication slot

When an instance is running, log records are generated and stored in log files. The cluster retains log files for recovery purposes after an instance is shut down improperly. These are files that contain log records from the start of the last completed checkpoint, and are retained unconditionally. You can also configure the cluster settings to specify how many files will be retained and the conditions for deletion.

Replication slots are used to hold log files for physical and logical replication purposes, as well as backup and replica creation.

Clients (`pg_receivewal` , `pg_basebackup` , `walreceiver` processes , logical replication worker instances) connecting via the replication protocol can specify a replication slot name. The presence of slots retains log files that were not received using those slots.

Slots are created and deleted by replication protocol commands and SQL functions and commands. Physical replication slots are created on and are specific to the master cluster. Each replica uses its own slot. A temporary slot exists only for the duration of a single replication session and holds logs only for the duration of the session.

If LSN is not specified when creating a slot, it is set when the client first connects. If the client does not accept log data (stops), the log files will be retained and fill all available space in the `PGDATA/pg_wal` directory . To prevent this from happening, you should set a limit with the **`max_slot_wal_keep_size` parameter** .

`pg_replication_slots` view contains a list of all replication slots that currently exist in the database cluster, as well as their current state.

To create a physical or temporary physical slot, you can use the `pg_create_physical_replication_slot` ('name') function.

To drop a slot `pg_drop_replication_slot` ('name').

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/functions-admin.html#FUNCTIONS-REPLICATION](https://docs.tantorlabs.ru/tdb/en/17_5/se/functions-admin.html#FUNCTIONS-REPLICATION)



# Create a basic backup

- **pg\_basebackup** is best run on the host where the backup will be stored (the backup host)
- **-D** parameter specifies the path to the directory
- by default creates two connections to two `wal-sender` processes . The first connection transfers data files, the second transfers logs
- You can limit the speed of the first connection backup
- initiates a checkpoint before copying begins
- creates **backup\_manifest** and **backup\_label** files



## Create a basic backup

`pg_basebackup` utility can create backups in plain and tar formats. We will not consider the second format, everything written below refers to the plain format.

To create a backup, simply specify the directory with the `-D directory` or `--pgdata=directory` parameter . If the directory does not exist, the utility will create it and all directories in its path, if they are missing. If the directory exists, it must be empty, this protects against overwriting files that may be important. The directory is created on the host where the utility is running.

If tablespaces were created in the cluster ( `PGDATA/pg_tblspc` contains symbolic links), then the directories that the symbolic links point to will be created. That is, the structure of the tablespace directories will be the same on the cluster and the host where the backup is created. If the backup is created on the same host, then you will need to specify the "mapping" - list the tablespace directories and where to back them up using the parameter:

`-T from=to` or `--tablespace-mapping=from=to`

All directories should be specified by their absolute paths, not relative ones. You can list extra directories, there will be no error. If you do not specify any directory, an error will be issued that the directory is not empty.

Symbolic links located inside the `pg_tblspc` subdirectory of the backup directory will point to new directories.

`-P` or `--progress` parameter will show what phase of the backup the utility is in.

`-r rate` or `--max-rate=rate` parameter allows you to limit the rate at which data is backed up to reduce I/O load. The range is from 32 KB/s to 1024 MB/s. The log transfer rate is affected only if the log transfer method is `fetch` , which makes no sense to use.

At the beginning of the master backup, the utility initiates a checkpoint. By default, the checkpoint is performed according to the value of the `checkpoint_completion_target` parameter in order not to load the input/output, i.e. its duration can be estimated as `checkpoint_timeout*checkpoint_completion_target` . If you want to perform the checkpoint as quickly as possible, you can use the `-c fast` or `--checkpoint=fast` parameter.

The `-t` or `--target` parameter can (but does not need to) back up to a directory on the cluster host, or back up "to nowhere" ( `--target=blackhole` ). The latter mode can be used to measure performance: what part of the backup time is spent reading files.

The utility backs up directories and files that are unknown to it. Therefore, you should not store files in `PGDATA` that you would not like to see in the backup, such as large message files.

The utility creates `backup_manifest`, `backup_label` files . The `backup_label` file contains the same data as in the backup `pg_control` file .

The utility **does not** back up files that are known not to need backing up. Such files are described in the documentation: [https://docs.tantorlabs.ru/tdb/en/17\\_5/se/continuous-archiving.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/continuous-archiving.html)

# wal-g backup utility

- additional client application supplied with Tantor Postgres
- Used to back up cluster and log files via S3 protocol
- supports creation of "delta copies" (incremental copies)
- Supports backup speed limit, file integrity check, concurrency level setting for file upload and download



## wal-g backup utility

WAL-G is an optional client application supplied with Tantor Postgres.

Used to back up the cluster and log files via the S3 protocol, which is also used by cloud file storage. Software such as minio is available for internal network backup.

One of the utility's advantages is the ability to make incremental (also known as differential) backups. In WAL-G, these are called "delta copies." They store file pages that have changed since the previous backup.

Throttling is supported - limiting the speed of reading files and the speed of downloading to the storage location, checking the integrity of files, setting the degree of parallelism for downloading and uploading from the file storage.

Streaming of log records is not supported; logs are transferred via files (WAL segments).

Uses "push" mode - transfers files.

If the storage supports the S3 protocol and does not support network mounting (usually NFS), then WAL-G may be useful.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/wal-g.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/wal-g.html)



# Demonstration

- Resizing WAL files



## Demonstration

Resizing WAL files

# Practice

1. Create a basic cluster backup
2. Launching an instance on a cluster copy
3. Log files
4. Checking the integrity of the backup
5. Consistent backup
6. Deleting log files
7. Creating a log archive using the pg\_receivewal utility
8. Synchronous transaction commit and pg\_receivewal
9. Minimizing transaction data loss



## Practice

Create a basic cluster backup

Launching an instance on a cluster copy

Log files

Checking the integrity of the backup

Consistent backup

Deleting log files

Creating a log archive using the pg\_receivewal utility

Synchronous transaction commit and pg\_receivewal

Minimizing transaction data loss



7b

## Logical backup



# Logical redundancy

- formation of a text file or files allowing to recreate objects
- Saves objects in the state they were in at the start of the unload.
- The following utilities are used:
  - > `pg_dump`
  - > `pg_restore`
  - > `pg_dumpall`
  - > `psql`
  - > `COPY` command
  - > `psql` command `\copy`



## Logical redundancy

Backup at the logical level in PostgreSQL is the formation of a text file or files (" **dump** ") that allow you to recreate objects and their data, which do not differ from the application logic point of view from the image of the objects being backed up. After recovery, the data and objects are in the state they were in at the start of the dump. The file contains SQL commands or text, based on which SQL commands can be generated.

Logical backup tools:

1) `COPY TO` command

2) `psql` command `\copy to`

`pg_dump` command line utility

`pg_dumpall` command line utility

5) Logical Copy Recovery Tools:

6) `COPY FROM` command

7) `psql` command `\copy from`

`pg_restore` command line utility

9) `psql`

The functionality of logical backup and recovery is determined by the parameters of these tools and is detailed enough to suit any needs.

# Examples of use

- Transition to a new major version
- Changing cluster parameters that do not change after cluster creation
- Checking data integrity
- Getting script to recreate objects
- Reservation of objects
- Unloading a part of a cluster: a separate database, schema contents and other objects



## Examples of use

Logical backup allows you to copy data and/or objects to another database on the same or another cluster of the same or another version and manufacturer. Why would you want to do this?

- 1) To migrate to a new major version of PostgreSQL. If the unload and load times are acceptable, this is the best way
- 2) Changing cluster or database parameters that cannot be changed without recreating the cluster or database
- 3) To ensure that the data is not damaged. Only logical level unloading can guarantee this.
- 4) For simple dumping of the contents of a single database. Physical backup with pg\_basebackup does not allow dumping of databases separately.
- 5) Transfer data to other storage systems, such as DBMS from other manufacturers, or load data from third-party sources
- 6) Get a text command file (script) to install the application.
- 7) Quickly and easily reserve objects and data at any level (cluster, databases, database objects, global objects), obtaining a complete copy (at one point in time)

# Comparison of logical and physical redundancy

| opportunity                                                     | L | F |
|-----------------------------------------------------------------|---|---|
| dump the contents of a separate database                        | + | - |
| restoration of part of the objects                              | + | - |
| restore to an arbitrary point in time                           | - | + |
| does not depend on the version, build, or software manufacturer | + | - |
| ensuring fault tolerance (Durability)                           | - | + |
| uses a replication protocol                                     | - | + |
| network backup                                                  | + | + |
| ease of use                                                     | + | + |



## Comparison of logical and physical redundancy

Logical and physical backup have different purposes of use. Physical backup is used to be able to restore data to the most recent point in time, i.e. without losing transactions. Logical backup is not able to do this, it can only restore data to the moment of unloading. Therefore, logical backup should not be considered as the only backup method.

Logical backup is useful for quickly creating a copy of a part of a cluster or transferring objects between databases. Physical backup creates a copy of the entire cluster, and its size can be large.

One of the advantages of PostgreSQL logical backup is that the format of the created files ("dump") is text with standard SQL commands, and not a proprietary binary format.

# COPY .. TO command

- used for highly efficient data unloading and loading into a single table
- You can dump the contents of a table or the result of any SQL command that returns data.
- The data is uploaded to:
  - a) a file in the server's file system
  - b) passed to the standard input (stdin) of the server's command line utility
  - c) the standard output stdout. If the COPY command is called over a network connection, the standard output is passed over the network connection to the client program



## COPY .. TO command

Unloading features:

- 1) You can unload by specifying the table name (but not the view) or any SQL command that returns data: WITH, SELECT (of any complexity), the VALUES command, commands with the RETURNING expression (INSERT, UPDATE, DELETE). The command must be enclosed in parentheses, the table name is not required. Usually, the table name is used if you need to unload all rows, or SELECT with the WHERE clause if you need to unload some rows. The VALUES command is not very common, but it is a standard SQL command.
- 2) You cannot specify a view name in place of a table name, but you can use a view name in a SQL command.
- 3) There are ten parameters that can be used to configure the format and features of the unloading: encoding, quotation symbols, escaping, how to handle NULL (empty values), whether to enclose the text in quotation marks, whether to display the column names in the first line:

```
COPY table [(columns)]
| (SELECT|VALUES|.RETURNING)
TO 'file'|PROGRAM 'command'
| STDOUT
WITH (
FORMAT text | csv | binary
DELIMITER 'character'
NULL 'marker'
HEADER true | false
QUOTE 'character'
ESCAPE 'character'
FORCE_QUOTE (columns) | *
FORCE_NOT_NULL (columns)
FORCE_NULL (columns)
ENCODING 'encoding_name');
```

The options marked in blue are those that can only be specified when unloading, not loading into a table. Two variations of the COPY command syntax are supported (for compatibility with PostgreSQL versions 9 and 7). The syntax variations differ in the order of the keywords. You should know this, as you can find examples with this syntax in books. The binary format can be processed faster than the text and CSV formats, but it is less portable and you can only unload and load into the same data type, not within a family of types. The COPY command is not part of the SQL standard and is specific to PostgreSQL.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-copy.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-copy.html)

# COPY .. FROM command

- Loading is performed into one table
- Data is loaded from:
  - › file on the host where the instance is running
  - › standard output of an arbitrary executable program
  - › from standard input stdin
- During the loading process, you can immediately set the line freeze flag
- An optional WHERE expression can be specified. Subqueries cannot be used in the expression.
- **pg\_stat\_progress\_copy** view - tracking the progress of the command both during loading and unloading



## COPY .. FROM command

There are nine parameters that can be used to customize the format and features of the download:

```
COPY table [(columns)]
FROM 'file'|PROGRAM 'command'
| STDIN
WITH (
FORMAT text | csv | binary
FREEZE true | false
DELIMITER 'character'
NULL 'token'
DEFAULT 'expression'
HEADER true | MATCH
QUOTE 'character'
ESCAPE 'character'
ENCODING 'encoding_name'
[WHERE expression] ;
```

Options marked in blue are those that can only be specified when loading into a table, not when unloading.

- **FREEZE** marks rows as frozen during loading. In this case, there is no need to update blocks in the future for freezing purposes. The table into which the data is loaded must have been created or truncated in the same transaction in which the COPY command is executed.

- **HEADER MATCH** is used to check that the column names and their order in the first row of the loaded data and in the table match. It can be used as an additional check that the columns are not mixed up during unloading and loading.

- An optional WHERE clause can be specified. Subqueries cannot be used in this clause. The expressions do not see the changes that the COPY command itself makes. This last point should only be taken into account if the WHERE clause calls functions with a VOLATILE volatility level and you expect them to see changes, but they do not.

- **DEFAULT** specifies a literal. If it is encountered in the input data, the default value set in the table definition will be inserted. Analog: insert into .. values (., DEFAULT, ...)

While the COPY command (both TO and FROM) is running, you can monitor its progress through the **pg\_stat\_progress\_copy** view .

**psql** has a **\copy command** . The **\copy** syntax is similar to COPY, but the actions are performed by the **psql** utility. The difference from COPY is that **psql** on the host where **psql** is running works with the file, not the server process. Since stdin and stdout are directed to the client when connecting over the network, the COPY command can work with files on the client using I/O redirection.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-copy.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-copy.html)



# psql \copy command

- **\copy** is a psql command
- the syntax is similar to the COPY command
- the command uses lower case and is case sensitive
- is typed in one line
- ";" at the end of the command is not necessary
- works with files where the psql utility is running, the COPY command works with files through a server process on the server host
- for large amounts of data COPY is more efficient



## psql \copy command

`\copy` is a psql command . The syntax of `\copy` is similar to the syntax of the COPY command , but the actions are performed by the psql utility . Differences from COPY :

- 1) `\copy` is typed on one line, COPY can be typed on multiple lines
- 2) `\copy .. from` in CSV format incorrectly handles single value in a row `\.` as the end of input and the following lines does not load
- 3) COPY allows variable substitution, backtick expansion (the ``` character ). For `\copy` , the end of the string is always treated as arguments to `\copy` , and neither variable substitution nor backtick expansion is performed in these arguments.
- 4) Number of lines processed `\copy .. to stdout` does not display
- 5) `copy ... to stdout` directs the output to the same location as the output of psql commands. To read/write psql's standard input/output, regardless of the source of the current command or the `\o` option, you can use `from pstdin` or `to pstdout`
- 6) The psql utility works with the file on the host where psql is running . This is slower than the server process working with the file. For large amounts of data, COPY is more efficient.

Because `stdin` and `stdout` are directed to the client when connecting over a network, the COPY command can operate on files on the client using I/O redirection. Instead of `\copy .. to` , you can use `COPY ... TO STDOUT` and terminate it with `\g name` or `\g | program` .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-psql.html#APP-PSQL-META-COMMANDS-COPY](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-psql.html#APP-PSQL-META-COMMANDS-COPY)

# pg\_dump utility

- creates a logical backup copy (dump) of a database or part of it
- unloads data from one database consistently - at one point in time
- To unload data from tables, the default command is COPY
- exports in one of four formats: **plain**, **custom**, **directory**, **tar**
- **directory** format can be unloaded in several streams
- for **custom** and **directory** the **-Z** parameter can be used to select the compression algorithm and level **zstd**, **lz4**, **gzip**, **0**
- You can avoid creating a file and use a **pipe** :  
pg\_dump parameters | psql parameters  
pg\_dump -F c parameters | pg\_restore parameters



## pg\_dump utility

`pg_dump` is a utility for creating a logical backup copy of the database contents. The utility connects to one database in the same way as `psql`, uses regular SQL commands, sets the lowest level `ACCESS SHARE` locks, the same as the `SELECT` command. This lock is needed so that the objects being dumped are not deleted during dumping. The only lock incompatible with `ACCESS SHARE` is `ACCESS EXCLUSIVE`. The utility dumps data consistently, that is, at one point in time. The utility can dump data in parallel using several processes, while consistency at one point in time is preserved. For this, the standard functionality is used - snapshot export. By default, it uses the highly efficient `COPY` command for dumping, but it can also form a set of `INSERT` commands. It can dump object definitions without data and vice versa. It has flexible settings that allow you to select in detail which types of objects to dump.

Exports data in one of four formats:

- 1) **plain** - by default. A script with a set of SQL commands is generated. The `psql` utility is used for loading. The main disadvantage is that you cannot specify multiple processes for simultaneous unloading.
- 2) **custom** - dumps in archive format, compressed by default. For restoration, the `pg_restore` utility is used, which can read the generated files. You cannot dump in several streams, but you can restore. Can be used with a pipe:  
`pg_dump -F custom parameters | pg_restore parameters`
- 3) **directory** - creates a directory in which separate files and a table of contents file will be created for each table and lob. The `pg_restore` utility is used for recovery. You can specify the number of threads that will simultaneously unload data - this is the main advantage compared to the custom format. Recovery can also be performed in several threads.
- 4) **tar** - similar to directory, but not parallelized or compressed. `pg_restore` is used for recovery. It has no advantages over the directory format.

`pg_dump` executes `SELECT` statements at a low level.

Using a pipe allows you to direct stdout to stdin of the `psql` utility and reload data without creating a file, which can save space in the file system.

By default, compression is used for custom and directory formats. Compression can significantly (tens of times) slow down the download. You can use a faster algorithm - **-Z zstd** or disable compression with the **-Z 0** parameter.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-pgdump.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-pgdump.html)

# Parallel unloading

- possible only in directory mode
- the number of worker processes is specified by the `-j` or `--jobs` parameter
- will be interrupted if after the unloading has started commands are given that set an exclusive lock on the objects being unloaded
- During the unloading process, **ACCESS SHARE locks are set** on all unloaded objects.
- If there are a large number of objects, you can increase the values of the parameters **max\_locks\_per\_transaction**, **max\_connections**



## Parallel unloading

The unloading time depends almost linearly on the volume of unloaded data. Since the data is processed at the logical level during the unloading process, the bottleneck may be the CPU core that will be served by `pg_dump`. To reduce the unloading time, unloading in several threads can be used. The unloading will be performed by the server process and worker processes. One table can be unloaded by one worker process. Unloading in parallel mode is possible only in the directory format. The other three formats are unloaded in one thread. The number of threads should be specified by the `-j N` or `--jobs=N` parameter. During the unloading,  $N+1$  sessions with the database will be created. The server process servicing `pg_dump` will create a snapshot and export it. The worker processes will use this snapshot so that the unloading is performed at the same point in time (is consistent).

In parallel mode, at the start of work, the server process requests `ACCESS SHARE` level locks for all objects that will be unloaded by worker processes. This is done to prevent objects from being deleted while the unloading is in progress. The number of such locks is limited by the value `max_locks_per_transaction * (max_connections + max_prepared_transactions)`. If the number of objects (tables) being unloaded exceeds this number, the server process will return an error about exceeding the number of locks and will terminate without starting the unloading. In this case, you can calculate the number of tables planned for unloading and increase `max_locks_per_transaction` on the primary cluster. On all replicas, the values of the above parameters must be no less than on the master.

with the `ACCESS SHARE` mode are those that set the highest level lock - `ACCESS EXCLUSIVE` (exclusive access). These are the `VACUUM FULL`, `DROP`, `ALTER`, `TRUNCATE`, `LOCK IN ACCESS EXCLUSIVE MODE`, `REFRESH MATERIALIZED VIEW` commands, as well as commands that can set this lock for a short time at the end of their work. If any session requests a lock on an object in exclusive mode, the lock request will be queued and will not allow other sessions to obtain a lock on the object until the `lock_timeout` parameter, if it was set, expires. Any attempt to access this object will be queued, following the exclusive lock. Since worker processes use their own sessions, they request an `ACCESS SHARE` lock before unloading data from an object and are queued following `ACCESS EXCLUSIVE`. **To prevent infinite waiting, worker processes request a lock in NOWAIT mode**. If the worker process fails to obtain the lock, the entire unload will be terminated.

# pg\_restore utility

- processes dumps in **custom, directory, tar** format
- **plain** text file from archives
- You can restore only object definitions without loading data by specifying the **--schema-only** parameter
- There is a specific option for loading using the "table of contents" file (TOC, title of contents):
  - generate archive contents file with **--list** parameter
  - comment out (delete, move) object lines
  - download archive specifying contents **--use-list**
  - objects not listed in the table of contents will not be loaded



## pg\_restore utility

`pg_restore` restores a database or objects from a backup created by `pg_dump` in all modes except text. Text mode creates a file that is executed by `psql`, not `pg_restore`.

Works in three modes:

- 1) If the `-d name` or `--dbname=name` parameter is specified, where the parameter value is the database name or connection string, `pg_restore` connects to this database and restores the archive contents to it. If you specify the key but do not specify the value, the `PGDATABASE` environment variable will be used. If the variable is not specified, the operating system user name will be taken as the database name. In this mode, loading into multiple sessions is possible for input dump files in custom or directory format. The tar format does not support parallel loading. Parallel processes perform the longest operations, such as loading data into tables and creating indexes.
- 2) If the `-l` or `--list` parameter is specified, a list of archive objects (TOC, table of contents) is output. The list file can be edited to avoid loading some of the objects. The edited list file is passed using the `-L file` or `--use-list=file` parameter.
- 3) If the `-d` and `-l` parameters are not specified, but `-f` is specified, a script with SQL commands is created. The generated `pg_restore` script will correspond to the output of `pg_dump` in plain format. The script is generated by a single process.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-pgrestore.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-pgrestore.html)

# pg\_restore capabilities

- you can restore only object definitions without loading data
- load objects of only part of the schemes
- do not restore ownership rights
- load into the default tablespace for the database
- restore part of partitions of partitioned tables
- restore only specified tables, indexes, views, sequences, functions, procedures, triggers



## pg\_restore capabilities

`pg_restore` utility , which are specified by parameters.

- 1) `-s` or `--schema-only` option to restore only the object definitions without loading the data. You can later load the data itself by specifying the `--data-only` option . This will load the table rows, lo, and set the sequence values. It makes sense to use `--disable-triggers` to disable triggers before loading the rows into the tables.
- 2) parameters `--clean` `--if-exists` before creating the object, the `DROP IF EXISTS` command is generated . Without the second parameter, informational messages are output to `stderr` ( usually this is not required)
- 3) `--create` create a database. In the connection parameter `-d` you will need to specify any existing database to issue the command to create a database and connect to it
- 4) `--exit-on-error` exit if an error occurs. By default, the utility continues to run and outputs the number of errors at the end of the work.
- 5) `-I` name. Generate a command to create the specified indexes. You can specify the parameter multiple times if you want to create multiple indexes.
- 6) to load the contents of not all, but only part of the schemes, you can use the `-n` or `-N` parameters
- 7) `--no-owner` do not restore ownership. Used if the set of roles in the cluster differs from those in the original cluster.
- 8) `-P` restore only the specified routines (procedures and functions)
- 9) `-t` restore only the listed "relations" (tables, views, materialized views, sequences, external tables)
- 10) `-T` restore only the specified triggers
- 11) `-x` or `--no-privileges` or `--no-acl` do not generate `GRANT`, `REVOKE` commands
- 12) `--section` restore table sections
- 13) `--no-tablespaces` option allows you to strip `CREATE` commands of tablespace names. Objects will be loaded into the default tablespace. This is used if the cluster does not have the tablespaces that were in the original cluster.

# pg\_dumpall utility

- Creates a single script that allows you to restore an image of the entire cluster
- Unloads shared cluster objects
- Runs `pg_dump` in plain mode sequentially for all databases to be dumped
- The script is text-based and contains SQL commands.
- The script can be executed in `psql`
- There is no command to create a cluster in the script, you need to create a cluster, start an instance and specify any database for the initial `psql` connection
- Most of the utility parameters refer to `pg_dump`, which the utility will run
- unloads into one stream
- you can use **pipe** :  
`pg_dumpall parameters | psql parameters`



## pg\_dumpall utility

Creates a script that allows you to restore the cluster image, i.e. all cluster objects in all databases and shared objects. The script contains SQL commands, it can be executed in `psql` and restore all databases and their contents.

The utility dumps the cluster's shared objects (roles, tablespaces, and permissions granted for configuration parameters) and sequentially runs `pg_dump` for each database in the cluster in plain mode. Connecting requires connecting to the instance multiple times for each database. If password authentication is used, you may need to enter the password multiple times, so it is convenient to use authentication that does not require entering a password.

The script does not contain a command to create a cluster. When running the generated script, `psql` must connect to a cluster instance, which can be created using the `initdb` utility. It is also necessary that the tablespace directories are located in the same paths as they were in the original cluster. It is not necessary to create the tablespaces themselves - the commands to create tablespaces will be present in the script.

The utility dumps the contents of the cluster into one stream. `pg_dump` is launched sequentially and dumping from different databases starts at different times. The contents of each database are dumped consistently - at the time `pg_dump` is launched .

Using a pipe allows you to direct stdout to stdin of the `psql` utility and reload data without creating a file, which can save space in the file system.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-pg-dumpall.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-pg-dumpall.html)

# pg\_dumpall capabilities

- has many parameters, most of them are `pg_dump` parameters
- you can only dump shared cluster objects  
`--globals-only`
- you can only dump role definitions  
`--roles-only`
- you can dump only tablespace definitions `--tablespaces-only`
- You can avoid unloading some databases  
`--exclude-database=template`
- do not add tablespace names to commands  
`--no-tablespaces`



## pg\_dumpall capabilities

The utility has many options, but most of them relate to the `pg_dump` utility , which will be run by `pg_dumpall` .

The `-g` or `--globals-only` parameter allows dumping common cluster objects: roles and tablespace definitions. It is used when you need to speed up copying the cluster contents: first, the roles and tablespaces are reloaded, and then the dump is launched in parallel for each database in the desired mode. For example, in parallel:

```
pg_dump --format=directory --jobs=N
```

`--clean` generates DROP commands for databases, roles, tablespaces. Useful even with an empty cluster, as the built-in postgres and template1 databases will be recreated and will have the properties that these databases had in the original cluster (localization parameters). `--if-exists` is usually used with this switch.

`-r` or `--roles-only` dump only roles, without databases and tablespaces

`-t` or `--tablespaces-only` dump only tablespaces, without databases and roles

`--exclude-database=pattern` do not dump databases with names matching pattern

`--no-tablespaces` Do not add tablespace names to commands. With this option, all objects will be created in the default tablespace.

Statistics are not downloaded and commands for their collection are not created. After downloading, you can collect it without waiting for automatic collection.

`--binary-upgrade` option is intended for use by the `pg_upgrade` utility ( with `--globals-only` or `--schema-only` ) to preserve the data file names of objects. Use for other purposes is not recommended or supported.



# Large size lines

- The text and bytea data types can store data up to 1GB in size.
- Strings larger than 1GB may require special handling
- Fields may increase in size when uploaded in text form



## Large size lines

### The first problem

The text and bytea data types can store fields up to 1GB in size. During the unloading (COPY) or processing of data by any commands, a buffer is allocated whose size cannot exceed 1GB. By default, the COPY command outputs field values in text format. In this format, special sequences such as `\r` `\t` `\b` are used for characters such as newline, tab, backspace, which occupy two bytes. In this format, a field containing special characters can exceed 1GB. When unloading a field bytea in text form, its size also increases and an error will be issued:

```
ERROR: out of memory
DETAIL: Cannot enlarge string buffer containing 1073741822 bytes by 1 more
bytes.
```

In this case, you can use the binary format: `COPY .. TO .. WITH BINARY;`

### The second problem

When processing strings, memory is allocated dynamically, increasing by the size of the field, and when unloading a string, an error may occur:

```
ERROR: out of memory
DETAIL: Cannot enlarge string buffer containing 536870913 bytes by 536870912
more bytes.
```

When unloading any type of data, including from lob, the row size cannot exceed 1GB. Such fields will have to be unloaded in parts: by columns; filtering rows and unloading problematic rows separately by fields.

utility parameter `-B` or `--no-large-objects` allows you not to dump lob. For working with lob, there are functions `lo_import()` and `lo_export()`.

### Comment

When working with large strings, server processes may try to allocate memory greater than 1 GB. For example, the current size of the string buffer is 999 MB, an attempt is made to increase it for processing another 1 GB field, a request is sent to the operating system to allocate another 1 GB. If there is no physical memory for this 1 GB, then this server process (or any process) receives signal 9 (SIGKILL) from oom-kill. If there is enough physical memory, then the server process returns "ERROR: out of memory" to the client and continues working.



# enable\_large\_allocations parameter

- Tantor Postgres DBMS parameter that increases the StringBuffer size 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



## Parameter enable\_large\_allocations

Tantor Postgres 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` and `pg_dumpall` utilities. The size of a table row field of all types cannot exceed 1 GB, but there may be several columns in the table and the row size may exceed both a gigabyte and several gigabytes.

`pg_dump` utility may refuse to dump such rows because it does not use the `WITH BINARY` option of the `COPY` command. For text fields, a non-printable character occupying one byte will be replaced by a sequence of printable characters occupying two bytes (for example, `\n`), and the text field may 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. The StringBuffer is allocated dynamically during the processing of each line, not when the server process starts. If there are no such lines, the parameter does not affect the operation of the server process.

This problem occurs with the row of the config table of the 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;
```

# Demonstration

- Handling Large Strings



## Demonstration

### Handling Large Strings



8a

## Physical replication



# Physical replication

- One main (primary, master) database cluster - allows changes to be made to data
- One or more standby clusters
- Standby clusters (physical replicas or simply replicas) receive log data and apply it to their files
- Replicas can serve read requests - hot standby mode
- Replicas are a physical backup of the primary cluster that is updated



## Physical replication

So far we have considered working with a single cluster, served by a single instance on a single host. A single host can fail, as can the data center in which the host is located. For High Availability (HA) of database content, you need to use at least one more host with its own file storage system and make sure that if the first host fails, the second host has the same data as the first and can serve client applications.

In this chapter we will look at the simplest and most common solution for ensuring high availability - replication of changes (log records) in data at the physical level (data file pages) - "physical replication".

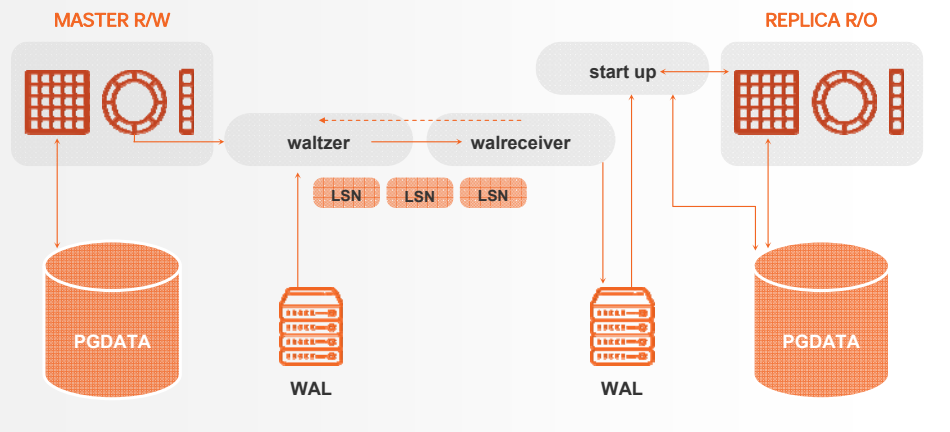
Usage model: There is a cluster with which client applications work. It is called the primary or master cluster. There is only one primary cluster in a configuration using physical replication. A physical backup copy of the files of this cluster is made on a standby host. This copy is called a standby cluster or physical replica or simply "replica". The transfer of log data to the standby cluster host is configured. An instance is launched on the standby host. The instance accepts and applies changes to the files of the standby cluster. There may be several such standby clusters, they can be located on different hosts.

The standby cluster is usually opened in read mode (hot standby) and can service requests. At the same time, the standby cluster continues to apply changes to its files and they become visible to sessions connected to the instance servicing the standby cluster. Long, analytical requests that usually generate reports can be transferred to the standby cluster.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/high-availability.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/high-availability.html)

# Master and replica

- The master and replicas must use the same major version of PostgreSQL
- the cluster is replicated as a whole
- tablespace directories may differ



## Master and replica

The master and replicas must use the same PostgreSQL master version. The entire cluster is replicated, including all databases. It is not possible to exclude some objects from replication. Tablespace directories may differ, since the tablespace directory is only pointed to by a symbolic link in the `PGDATA/pg_tblspc` directory .

Replicas cannot be modified, so they cannot create their own log records. The replica log files contain the master's log records.

Replicas can forward redo logs from the master via the replication protocol to other clients, such as other replicas. This is called cascading replication. Replicas that receive redo logs from a non-master cannot commit transactions in synchronous mode and cannot be specified in the `synchronous_standby_names` parameter.

# Replicas and archive of the magazine

- Typically, streaming replication is used using a replication slot: the walreceiver process connects to the walsender process
- A log archive can be used, filled by the command in the `archive_command` parameter or by the `pg_receivewal` utility (with a replication slot)
- the replica can use both the replication slot and if it cannot get the log file through it, then the directory with the log archive, if it exists



## Replicas and archive of the magazine

You can transfer log records to replicas using all available methods. For example, a replica can take log files from an arbitrary directory, such as the directory where the `pg_receivewal` utility or any other utility (for example, specified in the `archive_command` parameter ) places the received files. **Bigger** capabilities are provided by receiving log records via the replication protocol using a replication slot, just like the `pg_receivewal` utility . Only instead of `pg_receivewal` , a background process of the replica instance called `walreceiver` is used .

A replica can be configured to use either a replication slot or log files (the `restore_command` parameter on the replica). If the replica is unable to retrieve a log record via the replication protocol for any reason, it will execute the command specified in `restore_command` and if the command is successful, it will attempt to read the log file. In doing so, the replica will attempt to restore the connection via the replication protocol and will do so if it can retrieve log records via the replication protocol.

# Setting up the wizard

- Configure the ability to connect replicas using the replication protocol
- Check the values of the configuration parameters:
  - > **wal\_level** default replica
  - > **max\_walsenders** default 10
  - > **max\_replication\_slots** default 10
  - > **max\_slot\_wal\_keep\_size** default -1 no limit, should set limit



## Setting up the wizard

The main cluster (master) is most likely already in use and successfully serving client applications. You can create and configure a replica without downtime for servicing clients by the master. The replica is connected via the replication protocol, you need to configure the authentication parameters for the role under which the replica will be connected. You may need to change the values of the cluster configuration parameters that do not change without restarting the instance. Parameters:

**wal\_level** (default `replica` ) Must be `replica` or `logical` . Changing the value requires an instance **restart**

**max\_walsenders** (default 10) One replica uses one connection to walsender, but can reconnect in case of network failure, and the previous connection can exist up to `walsender_timeout` .

**pg\_basebackup** can use two connections. Changing the value requires re-reading the configuration parameters.

**max\_replication\_slots** (default 10) Must be at least the number of existing slots, otherwise the instance will not start. Each replica (regardless of cascading), `pg_receivewal`, `pg_basebackup` can use one slot. Changing the value requires **restarting** the instance.

**max\_slot\_wal\_keep\_size** (default -1, unlimited) The maximum size of log files that can remain in the `pg_wal` directory after a checkpoint for replication slots. If a replica uses a replication slot and does not connect to the master, the log files are kept by the master for that slot. If no limit is set, the log files will fill the entire file system and the instance will crash. To prevent this, set a limit. However, the replica will have to get the log files from somewhere else or the replica will have to be deleted. If the replica is no longer needed, remember to delete its slot. Changing the value requires rereading the configuration.

**walsender\_timeout** (default 60 seconds) Specifies the time period after which inactive connections via the replication protocol are terminated. Changing the value requires re-reading the configuration.

**synchronous\_standby\_names** and **synchronous\_commit** Configured after replicas are created to ensure protection against transaction loss if the master is lost. Can be changed without restarting the instance.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-replication.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-replication.html)

# Creating a replica

- creating a backup copy using the `pg_basebackup` utility
- The `-R` parameter sets the configuration parameters for the replica
- The `--create-slot ( -C )` and `--slot=name ( -S )` options create a slot to hold WAL logs until the replica instance is started.
- set `cluster_name` configuration parameter
- check the values of the configuration parameters to ensure they match the capabilities of the replica host
- Configure the service to automatically start the replica instance and start the replica instance



## Creating a replica

After the master is configured, you can start creating a replica. The replica can run on the same node as the master, but this does not protect against host loss, so it is used for training and testing purposes. In production use, the replica should run on a host different from the master. To simplify the setup, it is worth using the same `PGDATA` and tablespace directories as on the master, but this is not necessary.

When creating a replica using the `pg_basebackup` utility, it is convenient to:

- 1) use the `-C` (`--create-slot`) and `-S` (`--slot=name`) parameters to create a permanent replication slot. This slot will be used to pass logs to the `pg_basebackup` utility, and after its completion, the slot will not be deleted, it will hold the log files so that the master does not delete them before the replica is connected
- 2) use the `-R` (`--write-recovery-conf`) parameter. The following configuration parameters will be written to the `postgresql.auto.conf` file of the replica:
  - a) `primary_conninfo` - the address to which `pg_basebackup` connected to the master. The parameter specifies the address and network connection parameters with which the `walreceiver` process of the replica instance will connect to the master instance. `walreceiver` connects to `walsender` on the master instance
  - b) `primary_slot_name` - the name of the replication slot that the `pg_basebackup` utility used and which holds log files until the `walreceiver` of the replica connects to the master. The parameter has no effect if the cluster is not a replica or `primary_conninfo` is not specified`standby.signal` file is created in `PGDATA`, the presence of which tells the startup process not to complete recovery, but to be in the continuous recovery mode, the cluster is not opened for writing. If the `recovery.signal` file is present, `standby.signal` is in effect.
- 3) After creating the replica, set the value of the `cluster_name` configuration parameter on it. Changing the value requires restarting the replica instance. The parameter sets the default value for the `application_name` option of the `primary_conninfo` parameter. `application_name` sets the name of the replica, which can be used on the master in the `synchronous_standby_names` parameter. Also, the value of `cluster_name` will be output in the name of the instance's server processes, which is convenient for monitoring. If the `cluster_name` value is not set or is empty, then the `walreceiver` value is used for `application_name`.
- 4) Check and, if necessary, change the values in the parameter files `postgresql.conf`, `pg_hba.conf`. These files are copied from the master and may not be suitable for the replica host. For example, the replica host may have less physical memory, which will not be able to accommodate `shared_buffers`. If the replica is on the same host as the master, then the `port` parameter must be changed.
- 5) Configure the service to automatically start the replica instance and start the replica instance



# Replication slots

- are used by replicas
- list of slots in `pg_replication_slots` view
- Create function  
`pg_create_physical_replication_slot('name')`
- Drop a slot of any type  
`pg_drop_replication_slot('name')`
- Create a copy of a slot  
`pg_copy_physical_replication_slot('name', 'name_to_create')`



## Replication slots

There is no reason not to use replication slots. The slot is used by both the `pg_receivewal` utility and the replica. There are three types of slots: physical, temporary physical, logical. Logical is used for logical replication of changes in the tables of two primary clusters. Temporary slots are used in the process of creating an autonomous backup, usually intended for creating a clone or restoring to the end of the backup. Physical replication slots are used to transfer (broadcast) log records to replica clusters.

It is convenient to create a physical replication slot when creating a backup, which will be a backup cluster. This will allow you to start a replica instance "seamlessly" (without losing log files in the time interval between the end of the backup and the launch of the replica instance). When a replica instance is launched, the `walreceiver` process is launched, which receives log records and saves them in the `PGDATA/pg_wal` of the replica. The startup process is also launched, which rolls the contents of the `PGDATA/pg_wal` directory and periodically (the `wal_retrieve_retry_interval` parameter) checks whether anything new has appeared there.

Functions for working with physical slots:

`pg_create_physical_replication_slot('name', false, false)` - the slot must be given a name. The second parameter is important, by default it is false - the LSN is reserved when the streaming replication client first connects. If true, the LSN for this replication slot must be reserved immediately. The third parameter by default is false - the slot is physical permanent, if true then it is temporary.

`pg_drop_replication_slot('name')` - drops a slot of any type

`pg_copy_physical_replication_slot('name', 'name_to_create', false)` - creates a slot and initializes it with the LSN of an existing slot. Used if the same backup is used when creating two replicas.

The list of replication slots can be viewed in the `pg_replication_slots` view.

# Configuration parameters on replicas

- on the master they do not play a role, but can be pre-installed
- many do not change without restarting the replica instance
- by default:
- replica is servicing requests ( `hot_standby= on` )
- feedback is disabled ( `hot_standby_feedback= off` )
- the replica applies log records without delay ( `recovery_min_apply_delay=0` )



## Configuration parameters on replicas

Some configuration parameters configure the operation of the replica. During the operation of the master-replica configuration, one of the replicas can become the master, and the former master can become the replica. This is called changing the roles of database clusters in physical replication. The following parameters can be set in advance on the master, and when creating replicas, these parameters will be used on the replicas:

`walreceiver_status_interval` defaults to 10 seconds. Feedback will be sent no more often than once per this interval. The event horizon of databases on the master will be moved no more often than this interval.

`wal_retrieve_retry_interval` defaults to 5 seconds. Time a replica waits for log data to arrive from any source (streaming replication, local `pg_wal` log archive) before retrying the retrieve (`walreceiver` sends a `walsender` request and waits for a response, `startup` executes `restore_command`, `startup` reads `PGDATA/pg_wal`)

`recovery_min_apply_delay` defaults to zero. Will be discussed later.

`hot_standby` is on by default. Determines whether it will be possible to connect to the instance and execute queries (hot standby) or not (warm standby). The parameter plays a role only in replica or recovery mode. The value affects the behavior of the instance during replica recovery and maintenance. For example, if `hot_standby=off`, then the value of another parameter `recovery_target_action=pause` acts as shutdown, and if `hot_standby=on`, then as promote. The parameter changes only when the instance is restarted. If `hot_standby=on`, then the following parameters are in effect:

`hot_standby_feedback` ("feedback") - off by default. Sets whether the `walsender` of the replica (in the `hot_standby=on` mode, since there are no queries on the replica when off) will inform the `walsender` from which it receives logs about the queries it is currently executing. With cascade replication, data from all replicas (in the cascade) is transmitted to the master. The master holds the "database event horizon" for the longest query (or transaction in the `REPEATABLE READ` mode) among all replicas on which feedback is enabled. This leads to the fact that obsolete row versions are not removed not only by (auto)vacuum, but also by fast cleanup (HOT cleanup), but thanks to this, queries on the replica do not receive the "snapshot too old" error and have the opportunity to finalize and issue all the data.

`walreceiver_timeout` defaults to 60 seconds. A replica's `walreceiver` can detect that `walsender` has not responded and reconnect.

`max_standby_streaming_delay` and `max_standby_archive_delay` are 30 seconds by default. The maximum allowed delay time for WAL application.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-replication.html#RUNTIME-CONFIG-REPLICATION-STANDBY](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-replication.html#RUNTIME-CONFIG-REPLICATION-STANDBY)

# Hot replica

- enabled by default (parameter **hot\_standby=on** )
- the replica accepts connections and executes queries
- **pg\_basebackup** utility can perform backup of replica instead of master (backup offloading)
- temporary tables on replica cannot be used, **pg\_variables** extension can be used



## Hot replica

A physical replica is serviced by its own instance. A replica can be used to service commands that do not change data - queries. When migrating the reading logic, it is worth considering that the relevance of the data returned by the replica cannot be guaranteed. If the `synchronous_commit` configuration parameter on the master is set to `remote_apply` , then the replica can return data to its sessions (this behavior cannot be guaranteed) earlier than to the master. That is, if there are two sessions from a client to the master and the replica, in these sessions a `SELECT` command is issued simultaneously to rows that have just been changed by a transaction in a parallel session of the master, the session with the replica can return the data changed by this transaction, and the master session will not return it. It is impossible to guarantee the synchronicity of the return of the same data. It is not worth migrating the entire reading load to the replica. A part of the application logic that builds reports and executes analytical queries can be transferred to the replica. These are queries whose execution time significantly exceeds the replication lag (the delay in transferring and rolling back log records) and it does not play a role in the application logic.

By default, the `hot_standby=on` configuration parameter and the physical replica operates in hot standby mode - it can service commands that do not change data. For example, the `SELECT`, `WITH`, `COPY TO` selection commands , as well as the `BEGIN TRANSACTION`, `COMMIT`, `ROLLBACK` commands - these commands are needed to be able to execute queries at a single point in time, which is implemented by opening a transaction on the replica in `REPEATABLE READ` mode. The `SERIALIZABLE` level is not supported and does not differ for reading from `REPEATABLE READ` :  
`ERROR: cannot use serializable mode in a hot standby`  
`HINT: You can use REPEATABLE READ instead. The results of the`  
`COMMIT` and `ROLLBACK` commands will not differ, they are used only to close a transaction that did not change anything. Using temporary tables is impossible.

One of the useful features of the replica is that backup utilities can create backup copies by connecting to the replica, thereby removing the load from the master by transferring the backup to the replica.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/hot-standby.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/hot-standby.html)

# Feedback to the master

- feedback is disabled by default `hot_standby_feedback=off`
- feedback keeps the entire master's cleanup horizon to the duration of the longest query on the replica or transaction bases (REPEATABLE READ transaction level)
- Parameters can be used instead of feedback
  - > `max_standby_streaming_delay` and
  - > `max_standby_archive_delay`



## Feedback to the master

By default, `hot_standby_feedback=off` and the master does not take into account that `SELECT` commands are executed on replicas. This means that `DROP` commands can be executed on the master, passed to replicas, applied by the startup process, and the `SELECT` that accessed the object will not find it and return an error. And after `DROP DATABASE` on the master and executing this command on the replica, the sessions on the replica with this database will be interrupted. Commands for changing objects are rarely executed on the master, and there is no point in refining queries if they decided to delete an object. Vacuuming (including automatic) on the master, which cleans out old versions of rows, has a practical effect on queries on the replica. Old versions of rows are formed after their deletion or update, but not after inserts. A query on a replica can be interrupted even if vacuuming was not performed on the table, but due to a `HOT` (Heap-Only tuples) update.

If you want queries on replicas to be executed without errors, you can:

1) `max_standby_streaming_delay` and `max_standby_archive_delay` parameters to the duration of the longest query. If the query exceeds this time, it will fail with an error, not always, but only if there is a conflict. The delay in applying conflicting log records can increase the lag of the replica from the master ("lag") up to the values of these parameters. All sessions on the replica will issue data with a delay. Also, if you want to make the replica a master, there may be a delay in applying log records to eliminate the lag.

2) enable feedback. This will affect the master - it will not be able to clean up old row versions, since queries on replicas will hold the event horizon of the master's databases. Horizon retention affects vacuuming and `HOT` cleanup.

# Horizon Monitoring

- Estimation of the current database horizon:
  - › **backend\_xmin** from **pg\_stat\_activity**
  - › Duration of the longest running query or transaction:  
**max(now()-xact\_start)** from **pg\_stat\_activity**
  - › **xmin** column of the **pg\_replication\_slots** view when using replication slots
  - › Column **backend\_xmin** of view **pg\_stat\_replication** on master - what walsender got from feedback

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start)) secs, pid, datname database,
state from pg_stat_activity where backend_xmin IS NOT NULL OR backend_xid IS NOT NULL order by
greatest(age(backend_xmin), age(backend_xid)) desc;
```

| age    | secs        | pid    | database | state               |
|--------|-------------|--------|----------|---------------------|
| 175455 | 1425.651346 | 255554 | postgres | idle in transaction |
| 1      | 0.001878    | 255547 | postgres | active              |
| 1      | 0.001213    | 255626 | postgres | active              |



## Horizon Monitoring

Checking whether the database horizon is shifted is important to assess whether autovacuum can effectively clean up old row versions, and HOT can perform in-page cleanup, and to assess the impact of enabling feedback.

The number of queries cancelled in the replica databases since the statistics were reset can be viewed in the **pg\_stat\_database\_conflicts** view **on** the replica.

Monitoring queries and transactions on cluster databases:

```
select age(backend_xmin), extract(epoch from (clock_timestamp()-xact_start))
secs, pid, datname database, state from pg_stat_activity where backend_xmin IS
NOT NULL OR backend_xid IS NOT NULL order by greatest(age(backend_xmin),
age(backend_xid)) desc;
```

| age    | secs        | pid    | database | state               |
|--------|-------------|--------|----------|---------------------|
| 175455 | 1425.651346 | 255554 | postgres | idle in transaction |
| 1      | 0.001878    | 255547 | postgres | active              |
| 1      | 0.001213    | 255626 | postgres | active              |

**pg\_replication\_slots** view 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 **backend\_xmin** column contains the oldest transaction ID ("xmin") of the replica if feedback is enabled (**hot\_standby\_feedback=on** ).

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/monitoring-stats.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/monitoring-stats.html)

# Horizon Monitoring

- 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;
```

- 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;
```



## Horizon Monitoring

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.

The horizon of the cluster databases 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;
```

## Parameters `max_slot_wal_keep_size` and `transaction_timeout`

- `max_slot_wal_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` defaults to zero (timeout disabled). Allows any transaction or single command that exceeds the specified time period to be rolled back, not just idle ones. Protects against database horizon holdup and file bloat.



### Parameters `max_slot_wal_keep_size` and `transaction_timeout`

To prevent unlimited space usage, it is worth checking or setting the values of the following parameters.

`max_slot_wal_keep_size` defaults to -1 (no limit). The maximum size of log files that can remain in the `pg_wal` directory after a checkpoint for replication slots. If a slot is enabled and a client does not connect, the log files are kept. 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 6543) 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's worth setting a limit. However, a replica that fails to get the logs and they are erased will have to get the log files from somewhere else, or the replica will have to be deleted and recreated.

`transaction_timeout` is zero by default, timeout is 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 parameter applies to both explicit transactions (started with the `BEGIN` command) and implicitly started transactions corresponding to a single statement. 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.

`statement_timeout` + `idle_session_timeout` do not protect against transactions consisting of a series of short commands with short pauses between them (for example, a long series of fast `UPDATES` in a loop). The `old_snapshot_threshold` parameter can be used to protect against long `SELECT` commands. It should not be set on physical replicas. In version 17, `old_snapshot_threshold` was removed and `transaction_timeout` can be used to replace it.



# Master settings that should be synchronized with replicas

- If you change the values of these parameters on the master, then on the replicas these values must match the values on the master.
- Changes to these parameters are recorded in the log.
- if the replica detects that the value on the master has become greater, then the overlay of log records will be suspended or the replica instance will be stopped
- List of parameters:
- `max_connections`, `max_prepared_transactions`, `max_locks_per_transaction` limit the maximum number of object-level locks
- `max_wal_senders`
- `max_worker_processes`



## Master settings that need to be synchronized

Some parameters require attention. If you change the values of these parameters on the master, then these values on the replicas must match the values on the master. Since the master-replica roles can change, it is worth making the values of these parameters **the same** on all clusters, so as not to monitor the values after changing roles. If you need to increase the values of these parameters, you must first increase them on all replicas, and then make changes on the master. If you need to decrease the values of these parameters, first decrease them on the master, and then change the values on the replicas.

Changes to these parameters are written to WAL. If, while reading received WAL, the `startup` replica process detects that the value on the master has become greater than in the configuration of its instance, then if the replica is open for reading (the `hot_standby=on` parameter), a warning will be written to the cluster log and the overlay of log records will be suspended. If the replica does not allow connections (`hot_standby=off`), then the replica instance will stop and stop accepting log records, which can lead to a problem with synchronous replication.

List of parameters:

1) `max_connections`, `max_prepared_transactions`, `max_locks_per_transaction` these parameters limit the maximum number of object locks

2) `max_wal_senders`

3) `max_worker_processes`

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/hot-standby.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/hot-standby.html)



# Master-replica role reversal

- scheduled when the master is operational is called switchover
- If the master is not working, it is called failover
- make sure the master is stopped before signaling one of the replicas to become the master
- eliminate or minimize transaction loss
- `pg_last_wal_replay_lsn()` - the log record that was last restored
- `pg_last_wal_receive_lsn()` - last received LSN on a replica



## Master-replica role reversal

In physical replication, one of the clusters has the "role" of the master (leading, primary), the others have the roles of backup servers (replicas, slaves). You can change roles:

1) when the master is operational, such as for a scheduled shutdown of the master instance. This role change is called a switchover.

2) the master is inoperative. This role reversal is called failover.

Before the procedure, you need to:

1) eliminate or minimize transaction loss. To protect against loss, you can set up synchronous replication with transaction confirmation by replicas before the failure and switch to the replica that has the highest accepted and applied LSN. If synchronous replication was not used, then it is worth finding the master log files. You can determine the log file to which the master instance wrote before corruption by the master control file using the `pg_controldata` utility or other methods. This file and others, if they were not transferred to the replica, can be copied to the `PGDATA/pg_wal` directory of the replica and make sure that the log records from it are applied.

With synchronous replication, the setting can be such that transactions are confirmed by one of the replicas. If the master is corrupted, it may happen that only one replica received the latest log record, while the others did not. If you make a replica that did not receive the latest log record the master, then transactions may be lost. You can find out which of the replicas received the last log record to the master using the functions:

`pg_last_wal_receive_lsn()` - the last received LSN on the replica

`pg_last_wal_replay_lsn()` - the log record that was restored last. If `pg_is_in_recovery()` returns true, then this is the last applied log record. On the master, the function returns the LSN on which the master instance was opened after restoration, and if it was correctly closed, then it returns NULL. The replica with the higher LSN should be designated as the master.

2) There should only be one master at a time. If clients have two masters available and they accept changes ("split brain") from clients, then it will be difficult to parse transactions. To avoid having two masters available, you need to stop the master instance before signaling one of the replicas to become a master.

# Promoting a replica to master

- `pg_ctl promote` utility
- call the `pg_promote()` function
- promotion to master increases the timeline by one
- when switching to a new timeline, a text file `0000000N.history` is created in the `PGDATA/pg_wal` cluster directory



## Promoting a replica to master

There are two ways to make a replica a master (promote, advance or raise):

1) execute `pg_ctl promote`

2) call the `pg_promote(boolean, integer)` function . The first parameter is whether to wait for the operation to complete (default is true), the second parameter is the maximum number of seconds to wait (default is 60). Returns true if the promotion operation was successfully completed.

If you delete the `standby.signal` file and restart the replica instance, there will be no transition to the new timeline. In this case, the `pg_rewind` utility cannot be used in practice and the former master will have to be recreated. Deleting the `standby.signal` file can only be considered as a way to change roles with a correct master stop before promoting the replica.

After the new master appears, you can change the values of the `primary_conninfo` parameter of other replicas and the former master. Create replication slots on the new master. Make a replica out of the former master by creating a `standby.signal` file. If the former master instance was stopped correctly and the cluster files are not damaged, then it is enough to start the cluster instance without forgetting to create the `standby.signal` file . If the former master was stopped incorrectly, then you will most likely want to restore it. You can restore it by recreating the cluster: making a backup using the `pg_basebackup` utility `-R` . You can also use the `pg_rewind` utility if the new master promotion was performed with a transition to a new timeline.

# Timeline History Files

- They are located in the PGDATA/pg\_wal directory.
- They have the name 0000000N.history
- There is no need to delete these files.
- When a replica is promoted to master, a new timeline and a new timeline history file are created
- used by utilities and processes for backup and recovery purposes
- are requested and transmitted via the replication protocol along with log files
- are backed up along with log files



## Timeline History Files

Each time a new timeline is created, a timeline history file is created that stores tags of which timeline the new timeline branched off from and when.

A new timeline is created when a replica is promoted to master; when restoring from a backup to a point in time in the past, which can be specified by one of the parameters: `recovery_target`, `recovery_target_lsn`, `recovery_target_name`, `recovery_target_time`, `recovery_target_xid`.

History files are needed so that utilities and instance processes can find the name of the log file that contains the log entry with the desired timeline.

The timeline history file is a small text file in the PGDATA/pg\_wal directory named `0000000N.history`. You can add comments to the history file about how and why that particular timeline was created.

When a new file is created, the contents of the previous history file of the timeline on the basis of which the new timeline was created are saved into it.

Example of the contents of the file `00000003.history`

```
1 116/E30150E8 no recovery target specified
2 116/E30161E8 no recovery target specified
```

There is no point in deleting these files. Example of errors related to missing files:

```
pg_basebackup : could not send replication command "TIMELINE_HISTORY" : ERROR: could not
open file "pg_wal/00000002.history": No such file or directory
```

```
pg_rewind -D /var/lib/postgresql/tantor-se-17-replica/data1 --source-
server='user=postgres port=5432'
```

```
pg_rewind: connected to server
```

```
pg_rewind: error: could not open file "/var/lib/postgresql/tantor-se-17-
replica/data1/pg_wal/00000004.history" for reading: No such file or directory
```

Запуск экземпляра после перехода на новую линию:

```
pg_ctl start -D /var/lib/postgresql/tantor-se-17-replica/data1
```

```
...
```

```
LOG: unexpected timeline ID 2 in WAL segment 0000000400000116000000E3, LSN 116/E3016000,
offset 90112
```

```
LOG: invalid checkpoint record
```

```
PANIC: could not locate a valid checkpoint record
```

```
LOG: startup process (PID 7638) was terminated by signal 6: Aborted
```

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/continuous-archiving.html#BACKUP-TIMELINES](https://docs.tantorlabs.ru/tdb/en/17_5/se/continuous-archiving.html#BACKUP-TIMELINES)

# pg\_rewind utility

- synchronizes the PGDATA directory and tablespaces with the source cluster, including replacing parameter files
- used to bring a former master back into work after an unplanned role change
- the role change must be accomplished with the transition to a new timeline
- requires `full_page_writes` , checksum calculation (or `wal_log_hints` ) to be enabled
- creates a `backup_label` file that specifies parameters for restoring consistency



## pg\_rewind utility

`pg_rewind` utility synchronizes a cluster directory ( `PGDATA` and tablespace directories) with the directory of another cluster from which the synchronized directory was created.

The presence of timeline branching is essential for the utility to work. The utility searches for the `0000000N.history` file (contains the history of timeline creation) of both clusters in order to find the point at which the timelines of the two clusters diverged. Then it reads the log files in `PGDATA/pg_wal` starting from the last checkpoint before the moment when the timeline history diverged and up to the current log file of the cluster whose directory will be synchronized (the "target" cluster). From the log records it determines all the blocks to which changes were made. Then it copies these blocks from the other cluster.

Next, the utility copies all files located in `PGDATA` (and tablespaces), including new data files, log files, `pg_xact`, parameter files, arbitrary files. The directories `pg_dynshmem`, `pg_notify`, `pg_replslot`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp`, `pg_subtrans`, `pgsql_tmp`, files `backup_label`, `tablespace_map`, `pg_internal.init`, `postmaster.opts` and `postmaster.pid` are not copied. The utility creates a **backup\_label file** to switch to rewinding the log from the checkpoint to the point of divergence and sets in the `pg_control` file LSN of the beginning of the consistent state. The utility copies all parameter files that are in `PGDATA`. If the contents of the parameter files are important, it is worth saving them before running the utility in order to restore them after its operation.

Typically, the utility is used to bring a former master back into operation after an unplanned role change (failover). If the former master fails to send at least one log record to the replica that becomes the master, the former master cannot work as a replica. The `pg_rewind` utility is used to avoid completely recreating the former master .

It is essential that a new timeline is created when the replica is promoted. If this does not happen, `pg_rewind` will look for the most recent timeline, which may have been created a long time ago and the log files are no longer there.

If the utility cannot write to any file, it stops working. If the utility does not complete successfully and repeated attempts to start it do not result in a correct completion, the directory of the synchronized cluster cannot be used.

Using the `-R --source-server='address'` parameters simplifies configuration: a `standby.signal` file is created and the `primary_conninfo` parameter with connection parameters is added to the end of `postgresql.auto.conf` .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/app-pgrewind.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/app-pgrewind.html)

# Replica instance processes

- **postgres** main process, starts other processes, accepts connections
- **checkpointer** performs restart points
- **background writer** writes dirty pages to disk
- **startup** rolls log records from files to PGDATA/pg\_wal
- **walreceiver** receives a stream of log records and writes them to files in PGDATA/pg\_wal



## Replica instance processes

The following processes are present on the replica instance:

- 1) **postgres** - main process. listens on sockets, starts processes
- 2) **checkpointer** . Checkpoints are initiated only on the master. On the replica, a "restart point" is performed when a checkpoint log entry is received. If a failure occurs during recovery, the replica can continue from the last restart point.
- 3) **background writer** writes dirty pages from the buffer cache to disk
- 4) **startup** - rolls up journal entries
- 5) **walreceiver** , which receives log data from the master's **walsender process**
- 6) There may be extension processes, such as **stats collector** , as well as server processes that service sessions created with the replica.

Promoting a replica to master occurs quickly, since shared memory is allocated and some processes are running.

# Delayed replication

- You can set a delay for a replica in applying log records
- by default there is no delay
- `recovery_min_apply_delay` parameter
- the parameter is set at the cluster level, when changing it is enough to reread the parameter files



## Delayed replication

By default, a replica applies received log records immediately and at maximum speed. The `recovery_min_apply_delay` parameter sets the minimum delay that replica sessions should see data. The parameter is set on the replica and affects only it, not other replicas. The delay is calculated as the difference between the timestamp written to the log record on the master and the current time on the replica. If the time on the master and replica hosts is not synchronized and differs, the delay is not calculated accurately - taking this difference into account.

If the replica has just been created (using the `pg_basebackup` utility) and the replica files are not yet consistent, the log records for reconciling the files are applied immediately. The delay begins when the replica is synchronized and does not occur further, since the replica files remain synchronized.

When log records are received by a replica (the `walreceiver` process). The log files will be stored in `PGDATA/pg_wal` on the replica until they are applied by the `startup` process. The longer the delay, the larger the volume of WAL files that must be accumulated and the more disk space is required for the `PGDATA/pg_wal` directory on the replica.

It is important to remember that when using feedback (the `hot_standby_feedback` parameter), the master will not be able to clean up old row versions for at least the set delay (plus the duration of queries on the replica). You should use feedback with caution when using delay.

Also, if `synchronous_commit=remote_apply` on the master and the replica is the only one in the configuration or should be used to commit transactions, then all transactions will hang for the time of the set delay.

The delay is applied to log records containing `COMMIT`, other log records are rolled forward without delay if possible. However, log records cannot be rolled forward in any order due to dependencies between transactions. Therefore, you should not assume that by removing the delay, the replica will quickly roll forward log records.

Delayed replication is controlled by functions. For example, `pg_wal_replay_pause()` allows you to pause recovery. It is used if unwanted changes have occurred on the master and you need to decide what to do - unload data from the replica or roll up log records to the desired moment to make the replica the master.

# Recovering damaged data blocks from a replica

- The `page_repair` extension contains the `pg_repair_page` function for restoring damaged blocks.
- one block is restored per procedure call
- you can restore blocks of layers `main`, `vm`, `fsm`
- During the recovery process, an exclusive lock is set on the relation
- the block is requested from the replica via the connection
- connection parameters are passed to the function



## Recovering damaged data blocks from a replica

Tantor Postgres has a `page_repair` extension .

```
postgres=# select * from pg_available_extensions where name like '%repair%';
 name | default_ver | installed_ver | comment
-----+-----+-----+-----
 page_repair | 1.0 | | Individual page repairing
```

```
postgres=# load 'page_repair';
```

When a damaged data page appears on the master, it is possible to take the page image from the replica if it is not damaged on this replica. An extension must be installed in the master database.

Example command:

```
CREATE EXTENSION page_repair;
```

The extension contains two functions:

1) `pg_repair_page(table regclass, block_number bigint, connstr text)` Function  
parameters: table table name, block\_number damaged block number

`connstr` - connection string to the backup server. An example of a connection string can be taken from the `primary_conninfo` configuration parameter on any of the replicas. On the master, this parameter can be set in advance in case of a role change.

2) `pg_repair_page(table regclass, block_number bigint, connstr text, fork text)`  
`fork` - the name of the fork in which the block needs to be restored: 'main', 'fsm', 'vm' .

`pg_repair_page` function requests an exclusive `ACCESS EXCLUSIVE` lock on the object in which the block will be restored and waits until the replica overwrites the master's log records, eliminating the lag from the master. If you plan to restore several pages, you can obtain the lock in advance with the `LOCK TABLE` command .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/ be /page\\_repair.html](https://docs.tantorlabs.ru/tdb/en/17_5/ be /page_repair.html)

# Demonstration

- Creating a replica and running its instance



## Demonstration

Creating a replica and running its instance



# Practice

1. Creating a replica
2. Replication slots
3. Changing the cluster name
4. Creating a second replica
5. Choosing a replica for the role of the master
6. Preparing to switch to replica
7. Switch to replica
8. Enabling feedback
9. pg\_rewind utility



## Practice

Creating a replica  
Replication slots  
Changing the cluster name  
Creating a second replica  
Choosing a replica for the role of the master  
Preparing to switch to replica  
Switch to replica  
Enabling feedback  
pg\_rewind utility

# Practice

1. Using the pg\_dump utility
2. Custom format and pg\_restore utility
3. Directory format
4. Compression and backup speed
5. COPY command



## Practice

Using the pg\_dump utility  
Custom format and pg\_restore utility  
Directory format  
Compression and backup speed  
COPY command



8b

## Logical replication



# Logical replication

- supports replication of changes in rows of regular and partitioned tables
- Logical and physical replication can operate simultaneously.
- Two types of objects are created: publications and subscriptions
- The publication includes a set of tables from one database.
- The set of tables can be changed without stopping replication
- Changes are replicated, not commands.
- The same table can be a source and a receiver of changes



## Logical replication

Replication involves capturing, transferring, and applying changes to table rows. With physical replication, changes are tracked and applied at the physical level - the file and page level. With logical replication, changes are tracked at the level of tables and their rows, i.e. logical objects. In logical replication, changes are applied by SQL commands, for rows - row by row.

When you set up logical replication, you define sets of "source" tables whose changes you want to replicate. These sets of tables are included in a "publication" database object. You can add or remove tables from a "publication" without recreating it. A publication is a local database object and can only include tables that are in its database.

What is captured is not the SQL commands that made the changes, but their consequences: for each row affected by the command, the row identifier, the type of action with the row (delete, insert, change) and the values of the fields affected by the command in this row are captured. This logic is called "row-based replication". There are "statement-based replication" architectures, but this type of replication is not used for commands that process table rows, as it has side effects.

Logical and physical replication can run simultaneously.

Logical replication uses a "publish" (source) and "subscribe" (target) architecture. When replication is set up, objects of the same name are created in the databases.

Logical replication is evolving and new possibilities are emerging.

New features in version 15:

[https://docs.tantorlabs.ru/tdb/ru/15\\_12/se/release-15.html](https://docs.tantorlabs.ru/tdb/ru/15_12/se/release-15.html)

New features in version 16:

[https://docs.tantorlabs.ru/tdb/ru/16\\_8/se/release-16.html](https://docs.tantorlabs.ru/tdb/ru/16_8/se/release-16.html)

New features in version 17:

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/release-17.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/release-17.html)

# Using Logical Replication

Used for:

- transferring data to archival or analytical databases
- data consolidation: transferring data from regional databases to the central one
- organizing a backup database in case of loss of the main one
- reducing the load on the database by transferring some of the requests to backup databases



## Using Logical Replication

Examples of use other than those listed on the slide:

- propagation of changes in tables used by the application as reference data storage from the central database to regional ones.
  - event actions table - triggers to insert rows on the subscriber side when a row is inserted into a table in the publication table.
  - a table into which rows with the current time (timestamp) are inserted at a certain frequency and based on the publication database to track the lag and the status of the application on the subscriber side (heartbeat table).
  - replication between different major versions, builds, forks of PostgreSQL for the purpose of migration to them
  - distributing data to client applications that should not have access to the main database
- [https://docs.tantorlabs.ru/tdb/en/17\\_5/se/logical-replication.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/logical-replication.html)

# Physical and logical replication

Logical replication capabilities:

- you can replicate sets of tables, not the entire cluster
- insensitivity to major versions, platforms, builds
- log records are transferred not for the entire cluster, but only for replicated tables
- there is bidirectional replication
- log data **can** be transferred from a physical replica



## Physical and logical replication

Advantages of logical replication over physical replication:

- 1) you can replicate sets of tables, not the entire cluster
- 2) insensitivity to major versions, platforms, builds of PostgreSQL software physical damage does not apply
- 3) no extra traffic: log records are transmitted not for the entire cluster, but for the tables included in the publication
- 4) the structure of subscriber tables may differ from the structure of publication tables
- 5) flexibility: one table can be included in several different publications and subscriptions
- 6) there is bidirectional replication

Disadvantages compared to physical:

- 1) Only the results of executing the `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` commands on specific tables are replicated. Replication of other types of objects (external tables, views, etc.) is not supported. If the table on which the `TRUNCATE` command is executed is linked by a foreign key to tables not included in the subscription, the command will return an error on the Subscriber and replication will be suspended. Support for replication of the state of sequences is planned to be implemented in version 17. The absence of "support" means that the current value of the sequence on the Subscriber (if it exists) does not change. Data in auto-incremental columns (serial, bigserial) and generated columns ( `GENERATED .. AS IDENTITY` ) are replicated by values.
- 2) "conflicts" may occur, which will result in changes to subscription tables being suspended
- 3) requires setting the `wal_level` parameter to `replica` at the level of the entire cluster where the publication resides. This results in a significant increase in the volume of redo log data, especially if many tables have the `REPLICA IDENTITY FULL` property and rows are frequently deleted or changed in such tables.
- 4) `lo` replication is not supported . There is no way around this limitation except by storing data in regular tables, such as columns with the `bytea` data type .

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/logical-replication-restrictions.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/logical-replication-restrictions.html)

# Identifying Strings

- for replication of only row insertions (INSERT) and TRUNCATE, row identifiers are not needed
- to update and delete rows in the table there must be:
- primary key
- or a unique index and NOT NULL integrity constraints on each of the unique index columns
- or use all columns to identify a row



## Identifying Strings

Logical replication replicates not the text of SQL commands executed on tables included in the publication, but changes in table rows. For `INSERT`, identification is not required and `REPLICA IDENTITY` can have any value. For `UPDATE` and `DELETE` (and `MERGE` if at least one row is changed or deleted), it is necessary to identify the rows to which changes will be made. For identification, it is necessary to capture and transfer the values in the columns even if these columns were not mentioned in the command on the source. Sometimes this is called capturing field values before the change (before image). However, before image is a broader concept - they can be used for conflict resolution procedures and for this purpose, before image could include not only the columns identifying the row, but any others. In the current version, there is no automatic conflict resolution functionality and before images are used to identify the row.

To replicate `UPDATE` and `DELETE` s , which are replicated row by row, the publication tables must be configured with a "replication identifier" to identify the rows to modify or delete on the Subscriber side.

The simplest way to identify rows in tables is to use primary key values and this is the default value.

Instead of a primary key (including a composite key), you can assign any of the unique indexes on the table as a replication identifier. A primary key differs from a unique key in that the primary key has a `NOT NULL` constraint on all columns of the key . When using unique indexes, you must add this constraint to the columns that are used in this constraint. It makes sense to use unique indexes only if the table does not have a primary key.

Without primary keys and unique indexes, you can replicate `UPDATE` and `DELETE` , but you must specify a replication ID for all columns of the table. If you add a table to a publication that replicates `UPDATE` and `DELETE` operations without specifying `REPLICA IDENTITY` , `UPDATE` and `DELETE` transactions on the source (not on the subscribers) will fail.

# Methods of string identification

- If the table does not have a primary key, you must set the row identification method with the `ALTER TABLE` command name `REPLICA IDENTITY`
  - > `DEFAULT`; use primary key
  - > `USING INDEX name`; unique, non-partial, non-deferred index, set `NOT NULL` on all indexed columns
  - > `FULL`; pass values of all columns
  - > `NOTHING`; set for system catalog tables, meaningless for user tables



## Methods of string identification

`INSERT` commands will be able to execute without errors, they do not need an identifier and can be any. One of the possible values is `REPLICA IDENTITY NOTHING`. The documentation describes it as "Records no information about the old row", that is, in terms of "before image", and means that the values of columns other than those specified in the command are not captured, but the `UPDATE` and `DELETE` commands are blocked on the source. An example of an error on the source:

```
ALTER TABLE t REPLICA IDENTITY NOTHING;
UPDATE t SET t='b' WHERE id=2;
ERROR: cannot update table "t" because it does not have a replica identity and publishes updates
```

HINT: To enable updating the table, set `REPLICA IDENTITY` using `ALTER TABLE NOTHING` value is the default for system catalog tables (those in the `pg_catalog` schema).

There is no need to set `NOTHING` for regular tables, it does not provide any advantages, including for initial synchronization, since there is no need to identify rows during synchronization and inserts.

There are no requirements for indexes on subscription tables; indexes are created there to increase performance.

List of tables in the database that cannot replicate `UPDATE` and `DELETE` until a primary key is created or an identity method is specified:

```
SELECT relnamespace::regnamespace||'.'||relname "table"
FROM pg_class
WHERE relreplident IN ('d','n') -- d primary key, n none
AND relkind IN ('r','p') -- r is a table, p is partitioned
AND oid NOT IN (SELECT indrelid FROM pg_index WHERE indisprimary)
AND relnamespace <> 'pg_catalog'::regnamespace
AND relnamespace <> 'information_schema'::regnamespace
ORDER BY 1;
```



# Steps to create logical replication

- Set the value of the `wal_level=logical` parameter on the master and the physical replica to which the subscription will be connected (changing the parameter requires restarting the instance)
- select tables for replication
- check for primary keys or choose a method for identifying rows
- create tables in the recipient databases
- create publications based on the source database
- create subscriptions on the receiving databases and select subscription parameters



## Steps to create logical replication

To set up logical replication you need to:

- 1)check that the parameter `wal_level=logical` on the master and the source cluster
- 2)select tables to be included in the publication. Tables must have a primary key. If there is no primary key, it is advisable to check for the presence of a unique, non-partial, non-deferred index and the presence of the `NOT NULL` integrity constraint for the columns of this index and issue the `ALTER TABLE name REPLICA IDENTITY USING INDEX index_name command;` If these conditions are not met, you can issue the `ALTER TABLE name REPLICA IDENTITY FULL command ;` but in this case, when changing or deleting rows, old **values of all table fields will be written to the log** , and the size of the rows may be large.
- 3)Create tables on a database in this or another cluster that will accept changes. The table creation script can be obtained using the `pg_dump` utility with the `--schema-only` parameter . Logical replication functionality does not have the ability to copy table definitions. DDL commands are not replicated, and the definitions and set of tables are not synchronized. The initial set of tables in a publication can be copied using the `pg_dump` utility with the `--schema-only` parameter . Subsequent changes to the set of tables and their definitions will need to be synchronized manually. Schemas and tables do not have to be absolutely identical in the publication and subscriber databases. If the table definitions in the publication database do not change, logical replication works reliably. If a table definition changes in the publishing database, and the command on the subscriber cannot be applied, an error is returned, replication across the entire subscription is suspended, and you can manually change the table definition and replication will be restored without data loss. In many cases, you can change table definitions on the subscriber first, and then on the published table, and replication will not be interrupted.
- 4)In the database where the tables selected for replication are located, create a publication or publications using the `CREATE PUBLICATION command` . A publication can include tables only from its own database.
- 5)On the database with tables to which changes will be replicated, create a subscription or subscriptions using the `CREATE SUBSCRIPTION command` .

# Creating a publication

- select tables to be included in one publication
- You can specify which commands will be replicated: inserts, changes, deletions of rows, truncation of tables
- The publication may include:
  - › all database tables
  - › all tables in diagrams
  - › list of tables
- When specifying a list of tables, you can specify a set of columns and a filter for rows



## Creating a publication

After identifying the tables that should be included in the same publication because these tables are used in transactions at the same time, or are related by foreign keys, or logically must have time-consistent data (different publications may have different time lags), you can issue commands to create the publications.

Why not create one publication? The average size of rows in tables may vary. If there are tables with large rows, then changes in these rows will be transmitted to subscribers, and the network bandwidth may be limited and processing and applying to the subscription tables of all transmitted changes for all tables with a large volume of changes can create a significant replication lag in some time ranges. If you split the tables, for example, into two publications, and include tables with small rows in one, then the replication lag for them may be smaller and fluctuate within small limits. There are no universal means to determine how to split tables into several publications; you need to know how the application works with tables.

A publication name must be unique within its database. Creating a publication does not initiate replication. It only defines grouping and filtering logic for future subscribers. All tables added to a publication that publishes UPDATE and/or DELETE operations must have a REPLICA IDENTITY defined. Otherwise, these operations will be prohibited for those tables. For the MERGE and , the publication will publish an INSERT, UPDATE, or DELETE for each row inserted, updated, or deleted. COPY commands are published as INSERT operations. For the MERGE and INSERT.. ON CONFLICT commands, the publication will publish the actual operation on each row.

In the CREATE PUBLICATION command you can specify:

- 1) FOR ALL TABLES - replicates changes to all tables in the database, including tables created in the future
- 2) FOR TABLES IN SCHEMA - replicates changes to all tables in the specified list of schemas, including tables created in the future
- 3) FOR TABLE - a list of tables. If the word ONLY is specified before the table name , then only this table is added to the publication. If the word ONLY is not specified, then the table and all its descendants are added to the publication. After the table name, you can specify the names of the columns, then only the values of these columns (and the columns - row identifiers) will be replicated. By default, all columns are replicated, including those that will be added in the future. The WHERE option can be used to specify a filter to publish changes not in all rows, but only those changes that satisfy the specified condition. The list of tables can be empty. Tables can be added later using the ALTER PUBLICATION command.
- 4) In the WITH () option , you can specify values for two options. In the publish option, which row actions will be replicated: insert, update, delete, truncate . For partitioned tables, there is the publish\_via\_partition\_root option

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-createpublication.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createpublication.html)

# Create a subscription

- The subscription is created in the database where the change receiver tables are located.
- Each subscription must use its own logical replication slot in the primary database.
- The source and destination tables and their columns are matched by name and must be the same.
- The destination table may have additional columns. They can be populated with default values or triggers.



## Create a subscription

Once a publication has been created, subscriptions can be created on the databases containing the tables where changes will be replicated. Subscriptions are added with the `CREATE SUBSCRIPTION` command and can be suspended/resumed at any time with the `ALTER SUBSCRIPTION` command, and also removed with the `DROP SUBSCRIPTION` command.

Each subscription has its own logical replication slot in the publication database. When a subscription is created, an initial synchronization is performed by default, that is, existing rows in the source tables are copied to the subscription tables, using additional logical replication slots that are deleted after the synchronization is complete.

Publication tables are mapped to Subscriber tables by name. Replication to tables with different names on the Subscriber side is not supported. Table columns are also mapped by name. The order of columns in the Subscriber table may differ from the order of columns in the publication. The column types may also differ; the ability to convert the text representation of the data to the target type is sufficient. The target table may also contain additional columns that are not present in the published table. Such columns will be populated with default values specified in the target table definition or by triggers.

Each active subscription receives changes from its replication slot created on the publishing side. The subscription and the logical replication slot can be managed separately from each other. For example, you need to move the subscriber tables to another database (in the same cluster or another) and activate the subscription there. First, the `ALTER SUBSCRIPTION` command breaks the connection between the subscription and the slot. Then the subscription is deleted, the slot remains. Then the data is reloaded to another database and a subscription is created with the `create_slot=false` parameter, and is associated with the existing slot.

Just like physical slots, logical slots hold log files. If a slot is not going to be used, it must be deleted, both physical and logical.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/logical-replication-subscription.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/logical-replication-subscription.html)

# Create a subscription

- When creating a subscription, a connection string to the publication database is specified.
- You can specify several publications in one subscription if they are in the same database.
- When creating a subscription, rows from publication tables can be copied to destination tables.
- Logical replication only works through a slot
- The slot can be created in advance and specified when creating a subscription.
- The slot can be created when creating a subscription
- By default, the slot name is the same as the subscription name.



## Create a subscription

`CREATE SUBSCRIPTION` command creates a subscription. The subscription name is unique within the database where it is created. Subscription creation parameters:

1) `CONNECTION` 'string' connection to the publication database

2) `PUBLICATION` publication names separated by commas

3) `WITH` (parameter= value, ...). There are more than a dozen parameters, they are described in the documentation. The main parameters are:

`connect` is true by default. Whether to connect to the publication database. If set to false, the `create_slot`, `enabled`, `copy_data` parameters will also be false

`create_slot` = true by default. Whether to create a logical replication slot

`enabled` = true by default. Whether to start the subscription or leave it inactive

`copy_data` = true by default. Whether existing rows of tables to which subscription is made will be copied.

For large amounts of data, copying to one thread can take considerable time

`slot_name` is the same as the subscription name by default. It is worth setting up subscription naming rules so that their names are unique across all clusters. If you set the value to `NONE`, you need to set `enabled=false` and `create_slot=false`

`binary` by default false. This parameter allows to speed up initial synchronization and replication at the expense of less compatibility

`streaming` = off by default, data starts to be transferred to the subscription after the transaction is committed. When set to on, transaction data starts to be transferred immediately and is written to temporary files on the cluster with the subscription, and starts to be applied after the transaction is committed in the publishing database. When set to parallel, changes start to be applied immediately by the background parallel worker process. If there is no free process (their number is limited by the `max_logical_replication_workers` and `max_worker_processes` parameters), then the behavior is the same as for the on value. If transactions process large amounts of data, setting these values allows you to **reduce the replication lag**, since changes start to be transferred and applied without delay. The lag reduction that can be expected is 30-50%.

`synchronous_commit` = off by default. Overrides the value of the configuration parameter of the same name for transactions that apply changes to the subscription database. The off value is safe for logical replication, since if the subscriber loses transactions, they will be retransmitted.

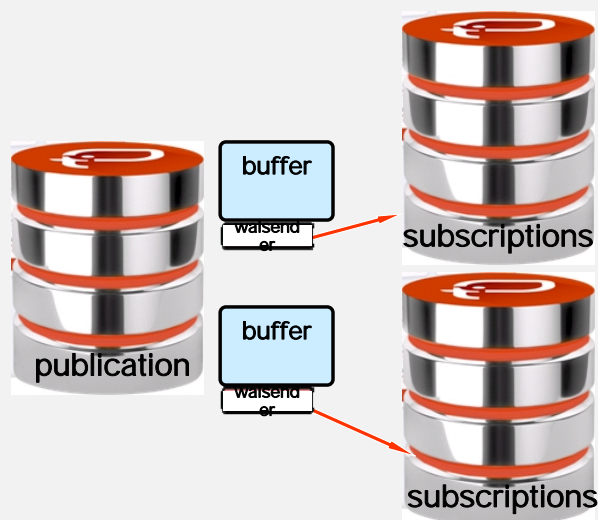
`disable_on_error` = false by default. If set to true, if an error is detected on the subscription side, the subscription is put into the disabled state. If true, periodic attempts are made to apply the change, in case the error disappears.

`origin`=any by default, publishing sends all changes. If bidirectional replication is used, then `origin=NONE` should be set to prevent loops ("ping pong", echo).

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/sql-createsubscription.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/sql-createsubscription.html)

# Load per instance

- For each subscription, a `walsender` process is started , which performs buffering and logical decoding.



## Load per instance

`walsender` process is started on the source cluster , one process **for each** subscription, which in turn uses a separate replication slot. The use of a logical replication slot is mandatory. Their number is limited by the `max_wal_senders` and `max_replication_slots` parameters . Changing the parameters requires restarting the instance. The `walsender` process reads the log files, but unlike physical replication, it does not simply transfer log records, but processes them. First, the `walsender` process accumulates changes made by each transaction in its local memory ( `reorderbuffer` ). By default, a subscription is created with the `streaming=off` parameter . This means that only committed transactions should be replicated, which is what the buffer is used for. If the volume of changes exceeds the `logical_decoding_work_mem` value (the conservative value of 64 MB by default ), the changes will be written to files in the `PGDATA/pg_replslot/slot_name` directory . Also, if the accumulated number of changes in one transaction exceeds 4096 , then the changes of this transaction will also start to be written to the file. The value is chosen large enough to cut off OLTP transactions from transactions with mass row changes.

The buffering logic can be changed using the `debug_logical_replication_streaming` configuration parameter .

The data accumulated in the buffer on committed transactions (or uncommitted ones if `streaming = on` or `parallel` ) are transferred to the output module `pgoutput` . The module is a separate process, and the code that executes is `walsender`. The operation of this module is affected by the subscriber's major software version number, the subscription's `binary` parameter (by default, `binary = off` and the conversion of changes to transactions in the form of text strings is used); `streaming` - if the value is `parallel` , additional information is transferred, origin (it is the module that filters transactions generated by logical replication processes) and other parameters that are set in the subscription properties.

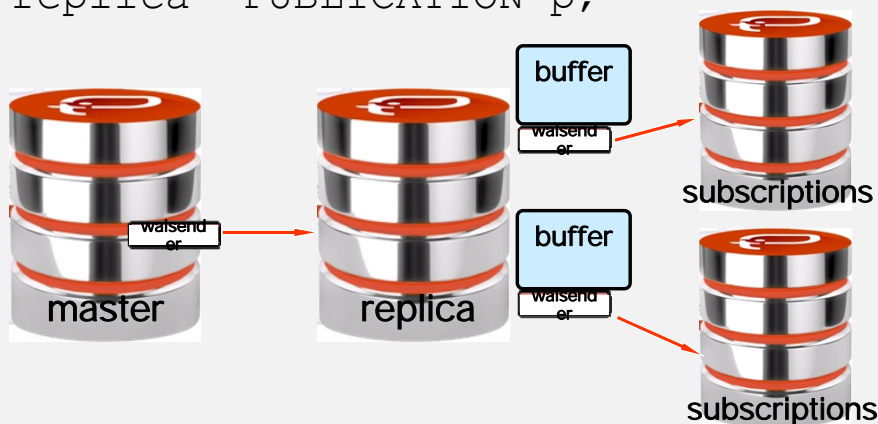
[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/protocol-logical-replication.html#PROTOCOL-LOGICAL-REPLICATION-PARAMS](https://docs.tantorlabs.ru/tdb/en/17_5/se/protocol-logical-replication.html#PROTOCOL-LOGICAL-REPLICATION-PARAMS)



# Getting log data from a replica

- When creating a subscription, it is enough to specify the address of the physical replica:

```
CREATE SUBSCRIPTION s CONNECTION 'dbname=name
host=replica port=replica' PUBLICATION p;
```



## Getting log data from a replica

Knowledge of the logical replication architecture allows you to understand the complexity of data processing on the source cluster, estimate the load on memory, processor (due to the large number of walsender processes) and disk I/O (reading log files by each walsender process and writing to files in the PGDATA/pg\_replslot directory). The load on the host where the walsender processes that service logical replication are running can be significant.

If you have physical replicas, it makes sense to move all the work performed by walsender processes to the physical replica side. In PostgreSQL's logical replication architecture, most of the change processing is performed by walsender, not on the receiver side.

To obtain data from a physical replica, you need to:

- 1) in the publication, specify the replica address in the connection parameter
- 2) the replica must be hot ( `hot_standby=on` )
- 3) enable feedback ( `hot_standby_feedback=on` ), otherwise autovacuum can clear the required subscription row versions in the system catalog tables and the slot will stop working, replication will stop
- 4) between the replica and the master you need to use a physical replication slot

the `CREATE SUBSCRIPTION` command does not return a prompt for a long time, you can execute the function on the master: `select pg_log_standby_snapshot()`. To create a logical replication slot, you need a snapshot (a list of all active transactions on the master). The replica does not have access to transactions on the master and is forced to wait until the `checkpointer` or `bgwriter` process on the master writes a snapshot to the log. If the prompt does not return after executing the function, it means that the initial synchronization of table rows is used ( `copy_data = true` ) and the data volume is large. The initial synchronization is performed through an additionally created logical replication slot, which will be deleted when the copying of rows is complete.

What happens if the master fails and the replica is promoted to master? Logical replication will continue to work without changes. Replication slots (logical and physical) in Tantor Postgres are preserved after role changes.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/logicaldecoding-explanation.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/logicaldecoding-explanation.html)

# Conflicts

- Conflict - if the **logical replication worker process** cannot make changes due to integrity constraint violation or other reason
- replication across the entire subscription is suspended
- There is no automatic conflict resolution.
- You can resolve the conflict by manually changing the data or skipping (not applying) the transaction in which the command that caused the error is executed.
- Information about the error appears in the cluster log.



## Conflicts

For each subscription, a logical replication worker process is started. This process connects to the walsender process via the replication protocol and receives a stream of changes decoded by the output module. Changes are made by `INSERT`, `UPDATE`, `DELETE` commands row by row: using the `REPLICA IDENTITY row identifier` . If the generated commands cannot make changes due to integrity constraint violation or for another reason (for example, a trigger fires and generates an unhandled exception, there are no privileges to execute the command), then replication in the entire subscription is suspended and will resume after the problem is fixed if the value of the subscription parameter `disable_on_error=false` . The occurrence of an error is called a "conflict".

If an `UPDATE` or `DELETE` command is executed and a row is missing (that is, zero rows were updated or deleted), then this is not an error and there is no conflict, the command is skipped and replication continues to work.

There is no functionality to create rules by which the conflict is resolved (automatic conflict resolution). Information about the error can be seen in the cluster log. The error indicates the LSN containing the `COMMIT` of the transaction to which the change that violates the constraint pertains.

You can resolve the conflict manually by changing the data or object definition: by changing the line with which the conflict arose, removing the integrity constraint, disabling the trigger, granting privileges. The second option: skip (do not apply) the transaction in which the command that caused the error is executed. This is done with the `ALTER SUBSCRIPTION command name SKIP (lsn = LSN)` . When the entire transaction is skipped (whose `LSN` with `COMMIT` is specified in the command), all changes made by the transaction are skipped, including those that do not violate any constraints.

the `streaming=parallel` subscription parameter , then the LSN of failed transactions can be written to the cluster log. In this case, you can change the value to on or off and resume replication.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/logical-replication-conflicts.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/logical-replication-conflicts.html)

# Bidirectional replication

- If you change the values of these parameters on the master, then on the replicas these values must match the values on the master.
- Changes to these parameters are recorded in the log.
- if the replica detects that the value on the master has become greater, then the overlay of log records will be suspended or the replica instance will be stopped
- List of parameters:
- `max_connections`, `max_prepared_transactions`, `max_locks_per_transaction` limit the maximum number of object-level locks
- `max_wal_senders`
- `max_worker_processes`



## Bidirectional replication

Bidirectional replication - two or more sets of tables are sources of changes and receivers for each other. The directions of replication are configured independently, but the settings are usually the same. For two sets of tables, two publications and two subscriptions are created. For three sets, three and three.

When setting up bidirectional replication, the greater the lag, the greater the likelihood of conflicts. To avoid conflicts, horizontal or vertical "partitioning" is used. With horizontal, at the application level, each node is assigned rows that can be changed or inserted into the table. For example, depending on the value in the table column. For example, two databases in two cities. Sessions with databases change and insert rows related mainly to their cities. There are no restrictions on the database side, and if an application in one city stops working, clients can be directed to an application in another city, and it will work with any rows. With vertical "partitioning", which is used less often, each node can make changes to its own set of columns.

The goal of bidirectional replication is not to improve performance, but to provide fault tolerance.

When using sequences to generate primary key values in tables involved in bidirectional replication for two nodes, configure the sequences so that on one node the sequence produces even numbers and on the other node it produces odd numbers.

the `origin=NONE` option on all subscribers .

Local commands (in local sessions) have `origin= NONE` . Setting this to `NONE` means that the publication will forward changes that do not have an origin, i.e. those made by local transactions, to the subscription, rather than by the logical replication worker. This avoids loops in bidirectional replication.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/replication-origins.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/replication-origins.html)



# Demonstration

- Unidirectional replication
- Bidirectional replication



## Demonstration

Unidirectional replication

Bidirectional replication

# Practice

1. Table replication
2. Replication without primary key
3. Adding a table to a publication
4. Bidirectional replication



## Practice

Table replication

Replication without primary key

Adding a table to a publication

Bidirectional replication



9

## Tantor Platform Review



# Use cases

## Classical

1. View the existing monitoring system, localize the **bottleneck** (CPU, RAM, Disk)
2. Connect to the server via ssh
3. **psql** terminal client
4. Analyze statistics based on system views ( **pg\_stat\_statements** )
5. Find query plans in DB log files ( **auto\_explain** )
6. Place the query plan of interest in the online visualizer
7. Identify problematic operations in a query
8. Make a decision

## Advanced

1. Open Tantor platform in browser
2. Select the monitored DB centrally
3. Localize the list of most resource-intensive queries
4. Open the DB log for the time interval of interest
5. Open visual query plan
6. Read the optimization recommendations
7. Make a decision



## Use cases

Let's consider what two approaches to diagnostics and optimization of PostgreSQL DBMS performance might look like.

### Classic scenario

You have a monitoring system that has registered a sharp drop in your database performance. The first thing you do is look at the CPU, RAM, and disk subsystem metrics to see where the problem lies. You determine that the bottleneck is in the processor.

Next, you connect to the database server via SSH. Once you open a terminal, you typically run the `psql` client and start analyzing system views such as `pg_stat_statements` to determine which queries are most expensive.

After analyzing the system views, you look through the database logs to find query plans that are created using the `auto_explain` extension. You copy the query plan you are interested in and paste it into the online visualizer for further analysis. There, you identify problematic operations in the query, such as Seq Scan instead of Index Scan.

Ultimately, based on the information collected, you make an optimization decision: for example, adding an index or rewriting a problematic SQL query.

### Advanced scenario using the Tantor platform

If you have access to the Tantor Platform, everything becomes even easier and faster. You open a web browser, go to the Platform and select the database you want to monitor. There you will quickly get a list of the most resource-intensive queries.

You view the database logs directly in the Platform interface for the time interval you are interested in. The next step is to analyze the visual query plan, which is also available in the interface.

One of the benefits of the Tantor Platform is the optimization recommendations. After reading them, you immediately understand what actions to take to solve a performance problem. By making a decision based on these recommendations, you save time and effort that would normally be spent on manual analysis and diagnostics.

In conclusion, although the classical approach requires more thorough and expensive analysis, it can be useful for understanding the nuances of the DBMS operation. However, specialized tools such as the Tantor platform offer a fast and convenient way of optimization that is ideal for complex and large systems.

# Monitoring tools

General-purpose monitoring systems not adapted for PostgreSQL



## Platform "Tantor"



## Monitoring tools

In today's world, DBMS performance monitoring is a critical component to ensure the reliability and efficiency of any application or complex infrastructure.

PostgreSQL is one of the most powerful and flexible DBMS, but its full potential can only be realized with the right monitoring tools.

There are many general-purpose monitoring systems, such as Zabbix, Grafana, OKMeter, New Relic, Munin, Cacti, and Datadog. They provide a wide range of features for monitoring various aspects of infrastructure and applications. However, precisely because of this "generality," they are often not adapted to work with the specific features and metrics of PostgreSQL. This means that they may not provide all the information you need in the most convenient form, which can be an obstacle to performance optimization.

On the other hand, there are specialized solutions, such as the Tantor Platform, which is designed specifically for PostgreSQL monitoring. This system knows which metrics are most important for the DBMS, how to collect them, and how to interpret them adequately. As a result, you get a detailed and complete picture of the state of your database: from query performance to disk load and memory usage.

Using a specialized solution like the Tantor Platform gives you the opportunity to make a qualitative leap in PostgreSQL performance tuning and monitoring. With this tool, you will be able to not only quickly identify bottlenecks and problems in your system, but also perform more precise tuning of DBMS parameters to achieve maximum efficiency.

In conclusion, choosing the right monitoring system is not just a matter of convenience, it is a matter that directly affects the efficiency of your application and, as a result, your business. Therefore, investing in specialized tools often pays off in the form of increased productivity and reduced troubleshooting time.

# Tantor Platform

- Functional software with a graphical user interface, created for convenient administration of PostgreSQL clusters
- Necessary for organizations that use multiple databases, each of which serves a specific information system or service
- Russian companies are switching to Russian analogues



## Tantor Platform

The platform is a functional software with a graphical user interface, installed, as a rule, within the customer's premises, it is created for convenient administration of PostgreSQL clusters.

With the help of the Tantor Platform, you can manage not only the Tantor cluster database, but also any other DBMS based on PostgreSQL, including the classic version.

The Tantor platform is necessary for organizations that use multiple databases, each of which serves a specific information system or service. Since each system has its own characteristics, different types of load and data - this makes the database a complex element of the corporate information system. Consequently, employees bear great responsibility for the normal functioning of the DBMS, and the Tantor Platform simplifies their daily work.

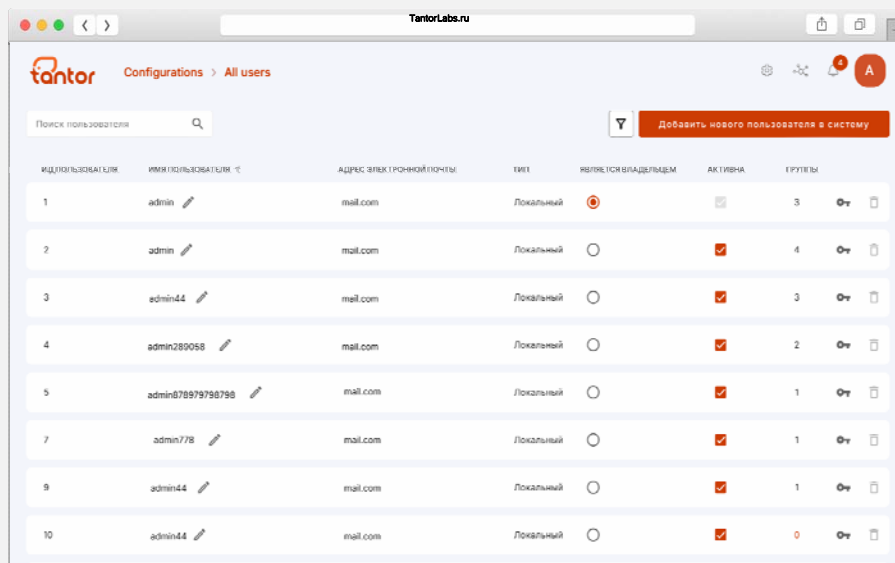
Due to the departure of international IT vendors from the Russian market, a large number of Russian companies, from small to large businesses, and all sectors of the economy, are switching to Russian analogues.

In all companies where there are IT services and DBMS are used, there is a need to administer a large number of database management systems.

<https://docs.tantorlabs.ru/tp>

# User settings

- Managing users and their roles



| ИД ПОЛЬЗОВАТЕЛЯ | ИМЯ ПОЛЬЗОВАТЕЛЯ  | АДРЕС ЭЛЕКТРОННОЙ ПОЧТЫ | ТИП       | ЯВЛЯЕТСЯ ВЛАДЕЛЬЦЕМ              | АКТИВЕН                             | ГРУППЫ |
|-----------------|-------------------|-------------------------|-----------|----------------------------------|-------------------------------------|--------|
| 1               | admin             | mail.com                | Локальный | <input checked="" type="radio"/> | <input checked="" type="checkbox"/> | 3      |
| 2               | admin             | mail.com                | Локальный | <input type="radio"/>            | <input checked="" type="checkbox"/> | 4      |
| 3               | admin44           | mail.com                | Локальный | <input type="radio"/>            | <input checked="" type="checkbox"/> | 3      |
| 4               | admin289058       | mail.com                | Локальный | <input type="radio"/>            | <input checked="" type="checkbox"/> | 2      |
| 5               | admin678979796796 | mail.com                | Локальный | <input type="radio"/>            | <input checked="" type="checkbox"/> | 1      |
| 7               | admin778          | mail.com                | Локальный | <input type="radio"/>            | <input checked="" type="checkbox"/> | 1      |
| 9               | admin44           | mail.com                | Локальный | <input type="radio"/>            | <input checked="" type="checkbox"/> | 1      |
| 10              | admin44           | mail.com                | Локальный | <input type="radio"/>            | <input checked="" type="checkbox"/> | 0      |

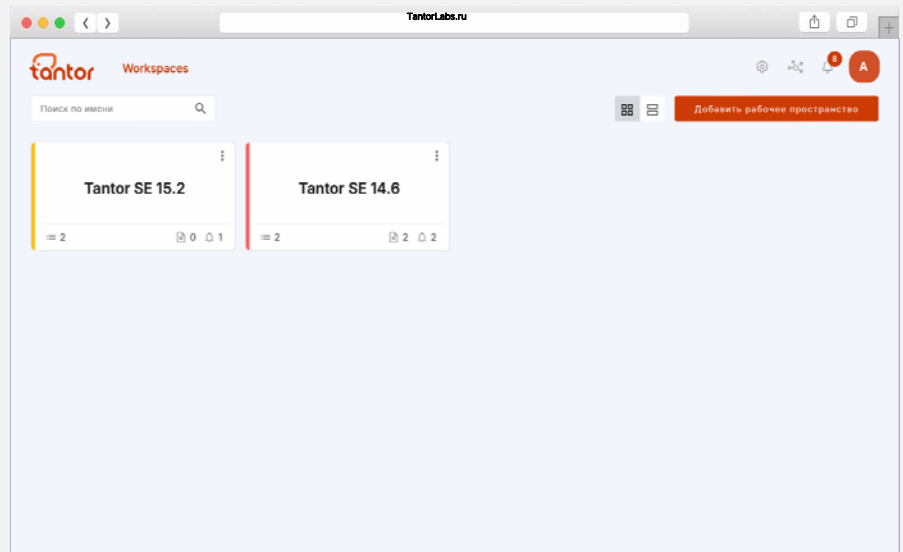


## User settings

In the Tanor Platform, user and group management is organized through the administration module, which allows various operations. The main functions include adding, activating, deactivating and deleting users. Options for managing user rights are also available, including transferring administrator rights. Additionally, it is possible to manage user groups by adding new groups and managing their membership, as well as integrate the system with Active Directory, ALD Pro and other LDAP-based directory services for deeper integration with corporate accounts.

# Workspaces

- Managing PostgreSQL instances and Patroni clusters



## Workspaces

A workspace in the Tantor Platform is a web interface designed to manage PostgreSQL server instances. It allows users to organize and manage different database instances that reside in one or more workspaces.

Within a workspace, you can create new instances, administer existing ones, and perform various management and monitoring operations. Each PostgreSQL server instance in a workspace is designed to manage one or more databases. Creating a new workspace requires a user role with administrative rights and is done through the Platform interface using a special input form.

The primary benefit of using Tantor Platform workspaces is the ability to centrally monitor and manage all of an enterprise's PostgreSQL database clusters. This enables unified management of all database resources, which is critical for large organizations looking to optimize and improve the efficiency of their IT operations.

Centralized management simplifies the process of configuring, monitoring, and maintaining multiple PostgreSQL instances, providing:

- Quick access to all the information you need about the status of each instance and cluster.

- Proactive management and monitoring that allows you to respond promptly to potential problems or changes in the operation of databases.

- Simplify the process of scaling and adding new instances and clusters without the need for significant system reconfiguration.

This makes workspaces an indispensable tool for ensuring the reliability, availability, and performance of databases in an enterprise environment.



# Instance review

- Consolidate all important information about a PostgreSQL instance and present it in a simplified form



## Instance review

On the instance overview page in the Platform, tiles provide a visual representation of key information about the database instance. Each tile provides an overview of a specific aspect of the instance, such as CPU load, memory usage, disk space, and active sessions and processes. Tiles can be configured to display data for different time intervals and include hover details, allowing you to quickly drill down to detailed information without having to navigate to other pages.

The Instance page opens a menu with modules for managing and monitoring the database instance. The menu includes the following modules:

Overview: General information and performance metrics of the instance.

Configuration: Instance settings.

Maintenance: Maintenance tools.

DB Inspector: Tools for analyzing database structure.

Query Profiler: Profiler for query analysis.

Current Activity: Displays the current activity.

Replication: Manage replication settings.

Tablespaces: Managing tablespaces.

Charts: Monitoring charts.

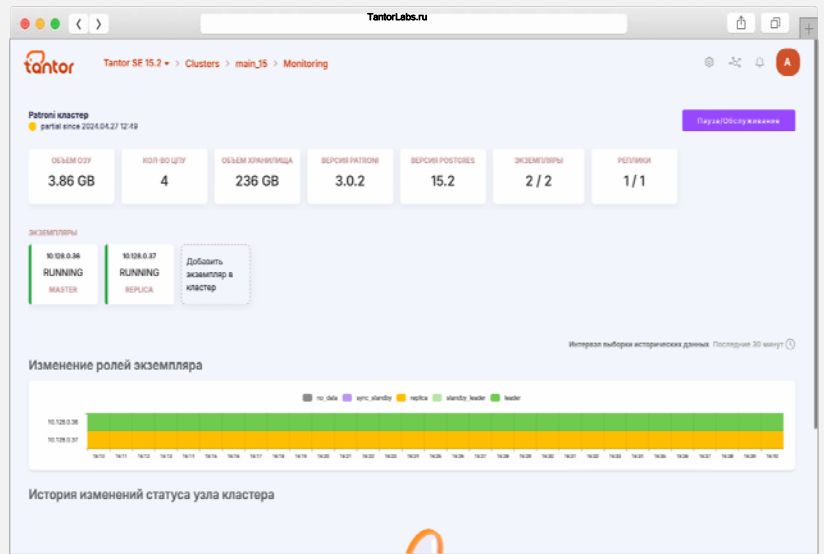
Monitoring Config: Monitoring system settings.

Advanced Analytics: Tools for advanced data analysis.

Tasks: Scheduler for running delayed tasks on DBMS instances

# Patroni cluster

- Visualization and intuitive management of Patroni clusters

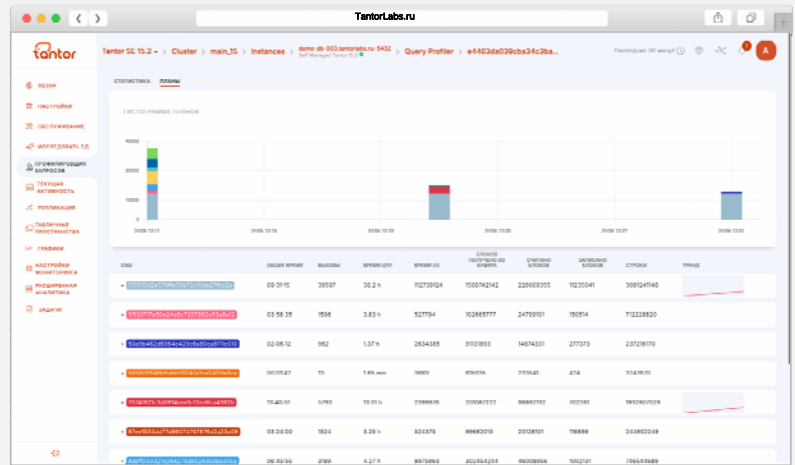


## Patroni cluster

Working with Patroni clusters on the Tanor Platform covers various aspects of management and monitoring. The Clusters tab allows you to see all clusters in the workspace, their status, Patroni version, and resources such as CPU and memory. You can drill down to detailed information about each cluster, monitor clusters, configure them, and manage maintenance. For each cluster, you can view, monitor, pause/maintain, and configure functions. Cluster information includes data about each instance, their roles, and statuses.

# Query Profiler

- Tracking query execution parameters and their plans for a selected period of time.
- Analysis and identification of problematic queries in the database



## Query Profiler

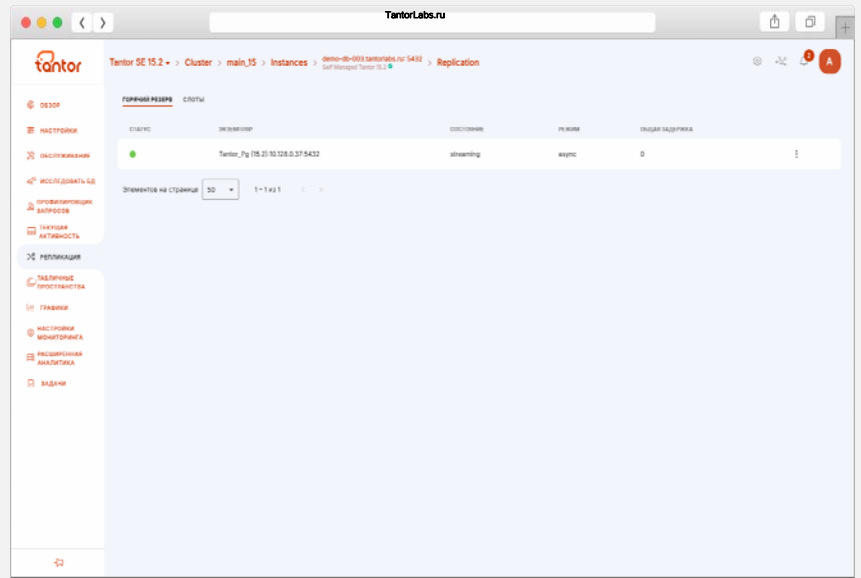
The Query Profiler on the Tantor Platform uses the `pg_stat_statements` extension to collect statistics on PostgreSQL queries. The main function of this tool is to identify and analyze slow queries. The profiler allows you to select different time intervals to display data, offers detailed metrics such as query execution time, number of calls, CPU time, and IO. It also includes visualizations such as graphs, and provides the ability to view query details, including text and execution plan.

The Tantor Platform's Query Profiler allows users to deeply analyze the performance of SQL queries. It helps in query optimization by providing data on execution time, memory usage, and the number and types of I/O operations. Using this information, developers and database administrators can identify and eliminate inefficient queries, which leads to improved overall system performance.

In addition, the Query Profiler has the ability to compare query performance over time, allowing you to track the impact of changes to your code or data structure on database performance. This feature becomes an invaluable tool when testing changes and evaluating their effectiveness.

# Replication

- Displaying the replication status on the Primary server and on the Standby server



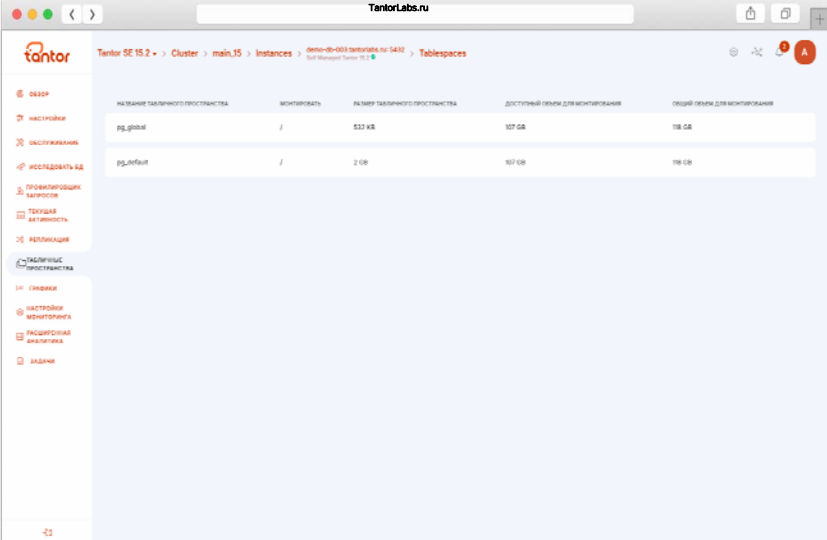
## Replication

The Replication module on the Tanfor Platform provides detailed monitoring of PostgreSQL replication. It includes two main tabs: STANDBY'S for managing replicas and SLOTS for managing replication slots. On the STANDBY'S tab, the user can see a list of replicas with their statuses and access to detailed information about each replica. The SLOTS tab displays a list of replication slots with their activity and status. These tools help optimize the replication process and ensure reliable data synchronization.

Additionally, the Replication module on the Platform allows you to monitor replication parameters in real time, which includes tracking replication delays and synchronization statuses. This is a key element for maintaining data integrity and minimizing the risk of data loss in the event of a primary server failure. Effective replication management helps ensure high availability and reliability of databases.

# Tablespaces

- Monitoring the space occupied by tablespaces in a database



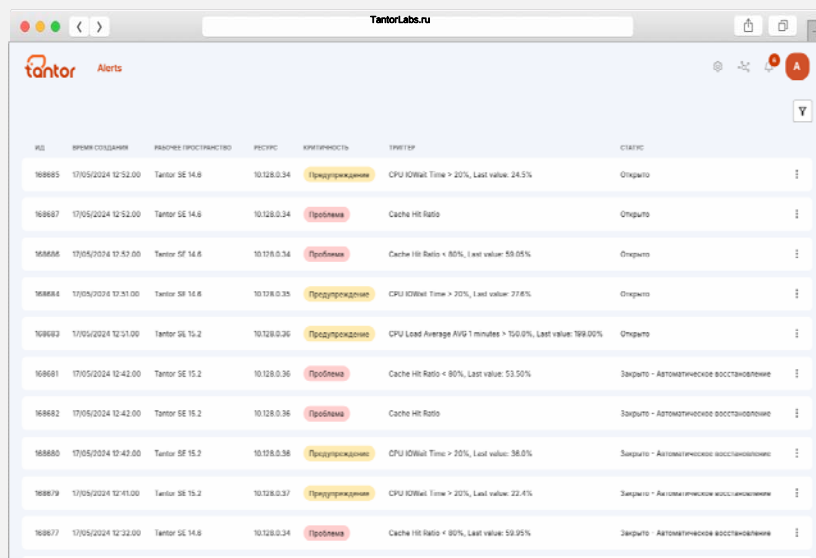
| НАЗВАНИЕ ТАБЛИЧНОГО ПРОСТРАНСТВА | МОНТИРОВАТЬ | РАЗМЕР ТАБЛИЧНОГО ПРОСТРАНСТВА | ДОСТУПНЫЙ ОБЪЕМ ДЛЯ МОНТИРОВАНИЯ | ОБЩАЯ ОБЪЕМ ДЛЯ МОНТИРОВАНИЯ |
|----------------------------------|-------------|--------------------------------|----------------------------------|------------------------------|
| pg_global                        | /           | 533 KB                         | 107 GB                           | 108 GB                       |
| pg_default                       | /           | 2 GB                           | 107 GB                           | 108 GB                       |

## Tablespaces

The Tablespaces module is designed to effectively monitor the space usage of tablespaces in a database. It compares the disk size with the size of each tablespace, which provides a more detailed view of the amount of space occupied by each part of the database.

# Notifications

- Generating notifications about critical situations with the database



The screenshot shows the Tantor Labs Alerts interface. It features a table with columns for ID, when column, previous IP address, IP, IP type, IP type, and status. The table lists several alerts, including CPU IDle Time, Cache Hit Ratio, and CPU Load Average. Each alert has a status indicator (e.g., 'Предупреждение', 'Проблема') and a description of the issue.

| ID     | when column         | previous IP address | IP          | IP type        | IP type                                                     | status                                  |
|--------|---------------------|---------------------|-------------|----------------|-------------------------------------------------------------|-----------------------------------------|
| 168865 | 17/05/2024 12:52:00 | Tantor SE 14.6      | 10.128.0.34 | Предупреждение | CPU IDle Time > 20%, Last value: 24.5%                      | Открыто                                 |
| 168867 | 17/05/2024 12:52:00 | Tantor SE 14.6      | 10.128.0.34 | Проблема       | Cache Hit Ratio                                             | Открыто                                 |
| 168868 | 17/05/2024 12:53:00 | Tantor SE 14.6      | 10.128.0.34 | Проблема       | Cache Hit Ratio < 80%, Last value: 58.05%                   | Открыто                                 |
| 168864 | 17/05/2024 12:51:00 | Tantor SE 14.6      | 10.128.0.35 | Предупреждение | CPU IDle Time > 20%, Last value: 27.6%                      | Открыто                                 |
| 168863 | 17/05/2024 12:51:00 | Tantor SE 15.2      | 10.128.0.36 | Предупреждение | CPU Load Average AVG 1 minute > 150.0%, Last value: 169.00% | Открыто                                 |
| 168861 | 17/05/2024 12:42:00 | Tantor SE 15.2      | 10.128.0.36 | Проблема       | Cache Hit Ratio < 80%, Last value: 53.50%                   | Закрыто - Автоматическое восстановление |
| 168862 | 17/05/2024 12:42:00 | Tantor SE 15.2      | 10.128.0.36 | Проблема       | Cache Hit Ratio                                             | Закрыто - Автоматическое восстановление |
| 168860 | 17/05/2024 12:43:00 | Tantor SE 15.2      | 10.128.0.36 | Предупреждение | CPU IDle Time > 20%, Last value: 36.0%                      | Закрыто - Автоматическое восстановление |
| 168879 | 17/05/2024 12:41:00 | Tantor SE 15.2      | 10.128.0.37 | Предупреждение | CPU IDle Time > 20%, Last value: 22.4%                      | Закрыто - Автоматическое восстановление |
| 168877 | 17/05/2024 12:32:00 | Tantor SE 14.6      | 10.128.0.34 | Проблема       | Cache Hit Ratio < 80%, Last value: 59.95%                   | Закрыто - Автоматическое восстановление |



## Notifications

The Alerts module is designed to create notifications about critical situations in the database. It provides the ability to monitor changes in the status of alerts, allowing you to quickly respond to important events in the database. This tool provides effective notification of possible problems, which allows you to quickly and accurately respond to changes in the operation of the database.

- Setting up lists of databases for monitoring.
- Setting up triggers for each database



In the Tantor Platform, the monitoring configuration module allows you to set up database monitoring, manage alert triggers, and configure alert conditions based on collected PostgreSQL metrics. This includes selecting databases to monitor, setting triggers for warnings, problems, and recovery, and saving changes to activate the configured monitoring parameters. The tool provides the ability to fine-tune monitoring parameters to effectively manage database performance and security.

- Collection, analysis and visualization of events from database logs

[illegible]

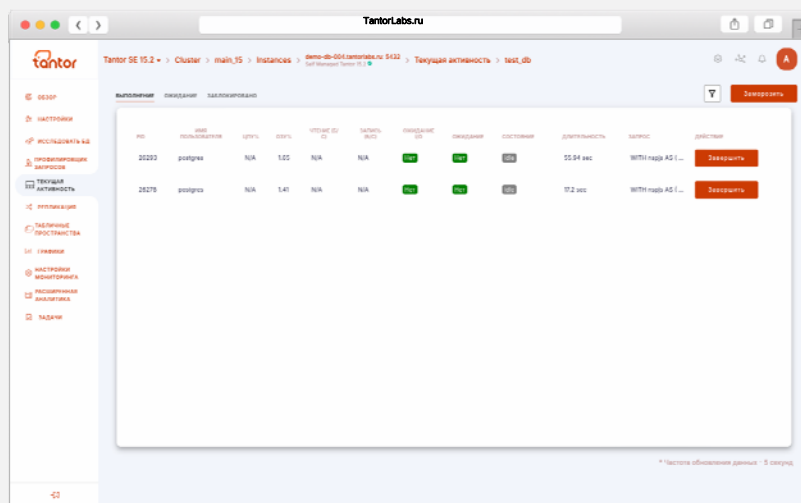
## Analytics

Advanced analytics on the Tantor Platform allows for detailed analysis of database server performance. It covers slow query analysis, blocking, errors, system actions, and logs. It includes sections for megaqueries and query problem diagnostics, allowing users to monitor performance and identify bottlenecks. The interface offers graphs and host summaries, making real-time monitoring and analytics easy.



# Background process activities

- Online monitoring of user sessions status in PostgreSQL database



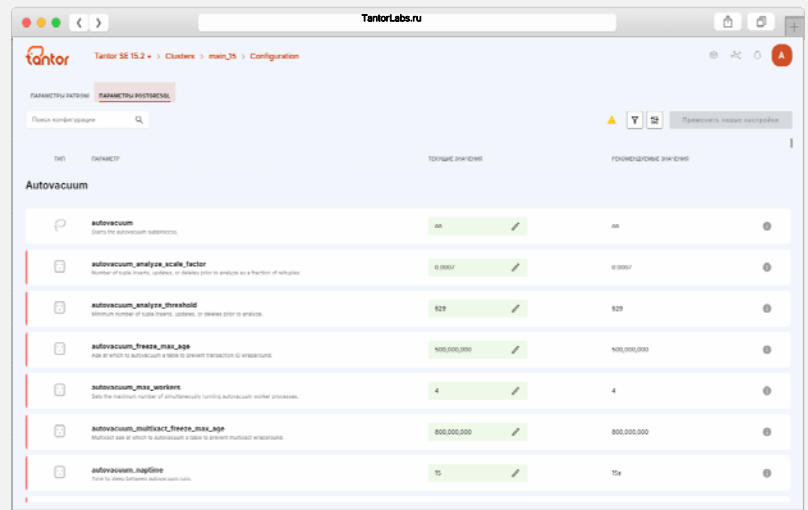
## Background process activities

The "Current Activity" module on the Tantor Platform displays detailed information about user and system database processes in real time. This includes monitoring of active, waiting and blocking sessions. For each process, parameters such as CPU usage, memory, read/write speed and process status are shown. The "TERMINATE" function allows you to terminate processes directly through the interface. All data is updated every five seconds and can be "frozen" for easy analysis.

The Current Activity module also offers convenient filters for sorting and searching processes by various parameters, such as user name, database, or session state. This makes it a powerful tool for quickly identifying and resolving database problems, especially in high-load environments where immediate intervention can prevent long downtimes or system failures.

# Settings

- View and modify the postgresql.conf configuration file and the Patroni cluster software configuration



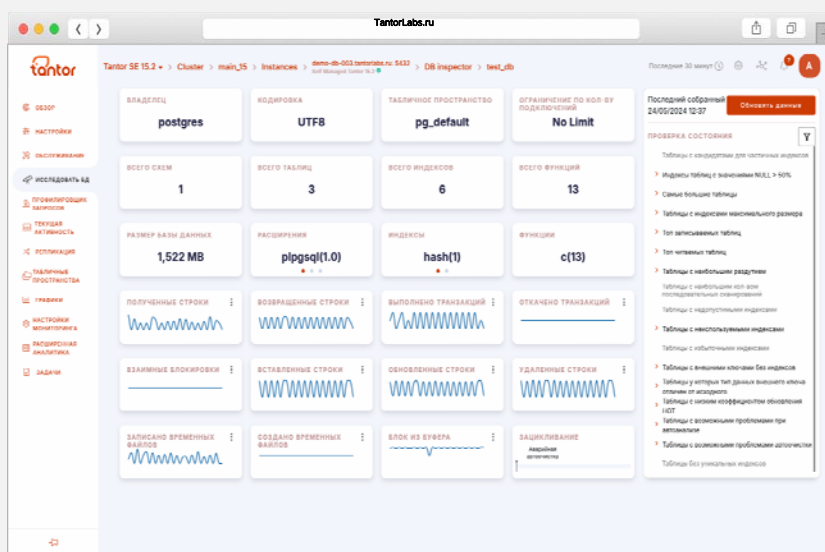
## Settings

The Settings module on the Tanor Platform provides a convenient interface for viewing and modifying the postgresql.conf configuration file, as well as the configuration of the Patroni cluster software. It automatically recommends optimal values for various parameters, allows you to apply these values, and requires a reboot or restart of the instance for the changes to take effect. The interface also provides color coding for easy identification of the status of parameters: default, changed, and requiring a reboot or restart. Users can filter parameters by status and category for ease of management.

In addition, the Settings module provides the ability to save and restore previous configurations, allowing you to easily roll back changes if necessary. This feature is especially valuable when testing new settings in a production environment, ensuring that you can safely experiment while minimizing the risk of failure.

# Data schema analysis

- PostgreSQL Database Schema Analysis



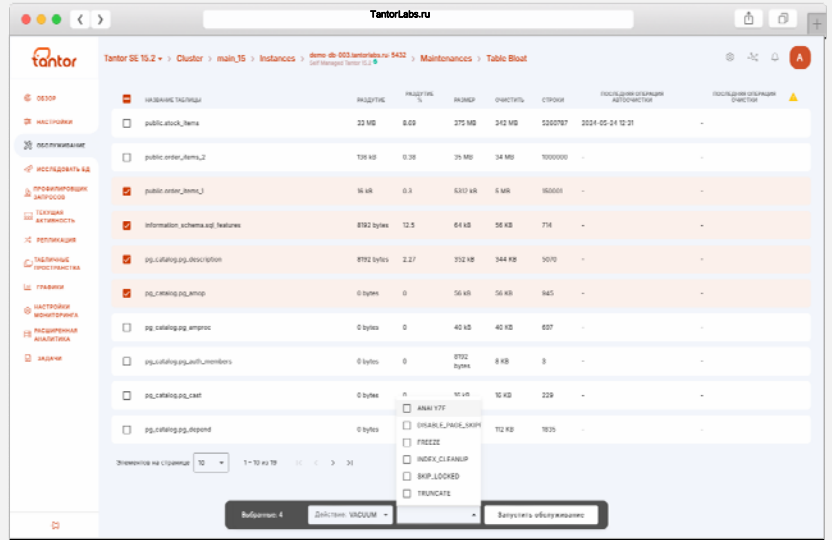
## Data schema analysis

The Database Inspector module on the Tantor Platform allows you to analyze the PostgreSQL database schema. Using HEALTHCHECKS, you can identify potential problems such as unused or redundant indexes, large tables, or non-optimal settings. The user has access to details of each problem, with the ability to apply recommended changes directly through the interface. This tool helps improve database performance and ensure its correct configuration.

In addition to the core functionality, the Platform's Database Inspector allows administrators to view detailed characteristics of each table and index, including information on the number of rows, space occupied, and read/write activity. This provides useful data for optimizing and reorganizing data, improving overall database performance.

## Routine maintenance

- Fix table and index bloat and eliminate transaction counter overflow



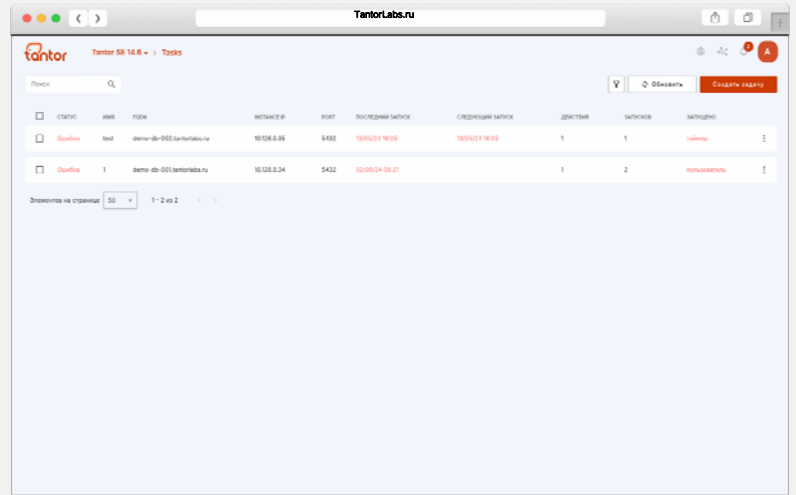
## Routine maintenance

The Tantor Platform Maintenance module manages PostgreSQL database maintenance tasks, such as fixing table and index bloat and eliminating transaction counter overflows. Users can select specific actions to fix problems and run VACUUM, REINDEX, or ANALYZE commands to optimize database performance. Maintenance history provides access to details and results of past operations.

The Maintenance page also allows you to set up automated scheduled tasks to improve the efficiency of your maintenance operations. This includes the ability to schedule tasks on a regular basis, which helps keep your database in an optimal state without the need for constant administrator intervention.

# Task Scheduler

- Automation and planning of various operations. Setting up the launch of actions according to a specified schedule, includes the execution of system commands or SQL scripts



## Task Scheduler

The Tasks module on the Platform allows users to automate and schedule various operations. It can be used to schedule actions, including running system commands or SQL scripts. This functionality is ideal for automating routine database management tasks such as updates, backups, or cleanup procedures, increasing efficiency and reducing the likelihood of errors through process automation.

# Tantor Platform Course

- The capabilities of the Tantor Platform are explored in the training course
  - › course duration 2 days
- Course topics:
  - › Introduction
  - › Preparing for work
  - › Monitoring
  - › Configuration and maintenance tasks
  - › Installing the Platform and Agent



## Tantor Platform Course

The capabilities of the Tantor Platform are explored in a 2-day training course.

Course topics:

1. Introduction
2. Preparation for work
3. Monitoring
4. Configuration and maintenance tasks
5. Installation

<https://tantorlabs.ru/educationcenter>

# tantor 10

10

## Tantor Postgres 17.5 New Features



# Tantor Postgres - PostgreSQL branch

Tantor Postgres DBMS includes:

- all the features of "vanilla" PostgreSQL
- features coming in future versions of PostgreSQL
- additional extensions and utilities
- changes in the PostgreSQL core that are needed for high-load DBMS and applications that generate complex queries (1C:ERP)
- own improvements



## Tantor Postgres - PostgreSQL branch

Tantor Postgres DBMS is a fork of PostgreSQL and:

- 1) includes all the features of "vanilla" (main branch) PostgreSQL
- 2) includes features that will appear in future versions of PostgreSQL. The process of accepting (committing) changes that add functionality (patches) to the main PostgreSQL branch is long and can take several years. Changes that are useful and have no drawbacks are added to Tantor Postgres before they appear in the main PostgreSQL branch. Example: the `pg_uuidv7` extension will appear in PostgreSQL version 18 appeared in Tantor Postgres version 16.8; parameters setting the sizes of SLRU buffers (`transaction_buffers`, `subtransaction_buffers`, etc.), timeouts (`transaction_timeout`), which appeared in the main branch of version 17 were added to Tantor Postgres version 15; extended use of SIMD processor instructions, which will appear in PostgreSQL version 18, appeared in Tantor Postgres version 17, and began to be implemented since Tantor Postgres version 15.
- 3) additional extensions. Extensions that are easy to port (rebase) to new major versions of PostgreSQL are added to the standard (contrib) extensions: they do not have compiled code, do not have a very large amount of code, do not have many interactions with the main code or have popular functionality. Many useful extensions and utilities are not included in the main branch, but are added to Tantor Postgres. For example, `pg_hint_plan` (optimizer hints), `pg_columnar` (columnar storage), `pg_ivm` (updatable materialized views), `pg_background` (use of background processes), `pgcopydb`, `pgcompacttable`, `pg_repack` utilities
- 4) changes in the PostgreSQL core that are needed for high-load DBMSs and are so complex that adding them to the main branch has been postponed for many years: a 64-bit transaction counter, autonomous transactions, improvements for compatibility with 1C:ERP and other programs that generate complex queries.
- 5) own modifications of PostgreSQL code, extensions, utilities. Modifications are offered in the form of patches to the community, are issued as free projects (<https://github.com/TantorLabs>) by their authors.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/differences.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/differences.html)



# Improvements in Tantor Postgres

- The modifications allow for improved performance and fault tolerance during industrial operation.
- Improvements are made so that Tantor Postgres is minimally different from the main branch of PostgreSQL
  - › the implementation that has the highest probability of appearing in the main branch is selected
  - › least changing PostgreSQL code and its default settings
- When administering Tantor Postgres, you can apply your PostgreSQL administration experience.
- Tantor Labs avoids modifications that could tie applications ("vendor lock") and make it difficult for applications to work in vanilla PostgreSQL



## Improvements in Tantor Postgres

The modifications allow for improved performance and fault tolerance during industrial operation.

Improvements are made so that Tantor Postgres is minimally different from the main PostgreSQL branch: the implementation is chosen that has the highest probability of appearing in the main branch or that least changes the PostgreSQL code and its default settings. For example, `pg_controlcluster` wrappers are not used, changes are made disabling (configuration parameters `enable_group_by_reordering`, `enable_temp_memory_catalog` and others). Tantor Postgres strives to be compatible with PostgreSQL and not differ in terms of operation.

Tantor Labs avoids modifications that could tie applications ("vendor lock") and make it difficult for applications to work in vanilla PostgreSQL.

When administering Tantor Postgres, you can apply your PostgreSQL administration experience. Your Tantor Postgres administration experience will be useful for working with PostgreSQL, including future versions.

# Additional configuration options

- Tantor Postgres 17.5 has 15 additional parameters that affect the creation and selection of query plans and functionality:

```
postgres=# \dconfig enable_*
 Parameter | Value
-----+-----
enable_convert_exists_as_lateral_join | on
enable_convert_in_values_to_any | on
enable_group_by_reordering | on
enable_index_path_selectivity | on
enable_join_pushdown | on
enable_self_join_removal | on
```

```
backtrace_on_internal_error | off
enable_delayed_temp_file | off
enable_large_allocations | off
enable_temp_memory_catalog | off
libpq_compression | off
wal_sender_stop_when_crc_failed | off
pg_stat_statements.sample_rate | 1
pg_stat_statements.mask_const_arrays | off
pg_stat_statements.mask_temp_tables | off
```

- Parameters introduced in PostgreSQL version 17:

```
allow_alter_system, commit_timestamp_buffers, huge_pages_status, io_combine_limit,
max_notify_queue_pages, ultixact_member_buffers, multixact_offset_buffers, notify_buffers,
restrict_nonsystem_relation_kind, serializable_buffers, subtransaction_buffers,
summarize_wal, sync_replication_slots, synchronized_standby_slots,
trace_connection_negotiation, transaction_buffers, transaction_timeout, wal_summary_keep_time
```



## Additional configuration options

Some improvements in the Tantor Postgres SE and SE 1C kernel have been made disableable by parameters.

Tantor Postgres SE parameters that affect the creation and selection of query execution plans:

```
postgres=# \dconfig enable_*
 Parameter | Value
-----+-----
enable_convert_exists_as_lateral_join | on
enable_convert_in_values_to_any | on
enable_group_by_reordering | on
enable_index_path_selectivity | on
enable_join_pushdown | on
enable_self_join_removal | on
```

Tantor Postgres SE parameters that affect functionality:

```
backtrace_on_internal_error | off
enable_delayed_temp_file | off
enable_large_allocations | off
enable_temp_memory_catalog | off
libpq_compression | off
wal_sender_stop_when_crc_failed | off
pg_stat_statements.sample_rate | 1
pg_stat_statements.mask_const_arrays | off
pg_stat_statements.mask_temp_tables | off
```

The parameters added in version 17.5 are highlighted in blue , and those added in version 16 are highlighted in green .

Parameters introduced in PostgreSQL version 17:

```
allow_alter_system, commit_timestamp_buffers, huge_pages_status, io_combine_limit,
max_notify_queue_pages, ultixact_member_buffers, multixact_offset_buffers,
notify_buffers, restrict_nonsystem_relation_kind, serializable_buffers,
subtransaction_buffers, summarize_wal, sync_replication_slots,
synchronized_standby_slots, trace_connection_negotiation, transaction_buffers,
transaction_timeout, wal_summary_keep_time.
```

Параметры, удаленные в 17 версии: db\_user\_namespace, old\_snapshot\_threshold, trace\_recovery\_messages

# Tantor Postgres SE and SE 1C Extensions

- Ядра Tantor Postgres SE и SE 1C унифицированы
  - › все возможности и расширения SE 1C есть в SE
- Tantor Postgres SE и SE 1C включают:
  - › расширения `credcheck`, `cube`, `fasttrun`, `fulleq`, `hypopg`, `mchar`, `page_repair`, `pg_cron`, `pg_hint_plan`, `pg_repack`, `pg_stat_kcache`, `pg_store_plans`, `pg_trace`, `pg_uuidv7`, `pg_wait_sampling`, `pgaudit`, `pgaudittofile`, `transp_anon`
  - › библиотеки `dbcopies_decoding`, `oauth_base_validator`, `online_analyze`, `pg_query_id`, `pg_stat_advisor`, `plantuner`, `wal2json`
  - › утилиты `pgcompacttable`, `pgcopydb`, `pg_diag`, `pg_repack`
- The following programs are supplied in separate packages: `pg_anon`, `wal-g`, `pg_configurator`, `pg_cluster`, `pg_diag_setup` , `pg_sec_check`
- Tantor Postgres SE additionally includes extensions:
  - › `http`, `orafce`, `pgl_ddl_deploy` , `pgq`, `vector` , `pg_archive` , `pg_columnar`, `pg_ivm` , `pg_partman`, `pg_qualstats`, `pg_tde` , `pg_throttle` , `pg_variables`, `pg_background`



## Расширения Tantor Postgres SE и SE 1C

Tantor Postgres SE and SE 1C kernels are unified. All features and extensions of the Tantor Postgres SE 1C build are in the Tantor Postgres SE build. In particular, a 64-bit transaction counter, autonomous transactions, optimized `pglz` data compression algorithm , expanded use of SIMD instructions of central processors.

Some of the changes in the kernel are made by adding options to SQL commands: `ALTER TABLE t ALTER COLUMN c SET STATMULTIPLIER 100;` in addition to `SET STATISTICS` .

In addition to the standard vanilla PostgreSQL extensions, the Tantor Postgres SE and SE 1C distribution package includes:

расширения: `credcheck`, `cube`, `fasttrun`, `fulleq`, `hypopg`, `mchar`, `page_repair`, `pg_cron`, `pg_hint_plan`, `pg_repack`, `pg_stat_kcache`, `pg_store_plans`, `pg_trace`, `pg_uuidv7`, `pg_wait_sampling`, `pgaudit`, `pgaudittofile`, `transp_anon`

библиотеки: `dbcopies_decoding`, `oauth_base_validator`, `online_analyze`, `pg_query_id`, `pg_stat_advisor`, `plantuner`, `wal2json`

utilities: `pgcompacttable`, `pgcopydb` , `pg_diag` , `pg_repack` .

The standard delivery includes the following programs in separate packages: `pg_anon`, `wal-g`, `pg_configurator`, `pg_cluster`, `pg_diag_setup` , `pg_sec_check` .

Tantor Labs releases and supports applications, utilities, extensions that are not included in the standard delivery of the Tantor Postgres DBMS (for example, PostGIS, pgRouting) under a separate agreement ("extension support certificates") , since porting extensions to the required DBMS version, assembling for the required Linux operating system , testing extensions, and technical support are complex. If the extension does not require modification and porting, Tantor Labs provides instructions for self-assembly.

The following extensions have been added to the Tantor Postgres SE distribution package:

`http`, `orafce`, `pgl_ddl_deploy` , `pgq`, `vector` , `pg_archive` (addition to `pg_columnar` ), `pg_columnar`, `pg_ivm` , `pg_partman`, `pg_qualstats`, `pg_tde` , `pg_throttle` (improved for cgroup use in linux), `pg_variables`, `pg_background` .

The parameters highlighted in blue are those that appeared in version 17.5 , and those highlighted in green are those that appeared in version 16 of Tantor Postgres .

The improvements available in Tantor Postgres **BE** are listed in the documentation:

[https://docs.tantorlabs.ru/tdb/en/17\\_5/ be /differences.html](https://docs.tantorlabs.ru/tdb/en/17_5/ be /differences.html)

# Query Optimizer Options

- Tantor Postgres version 17 introduces 6 parameters that can be used to enable query planning optimizations
- Optimizations allow to significantly reduce the execution time of queries encountered in practice
- Parameters have been added to provide flexibility in configuring the query optimizer.
- Optimizations are enabled by default.

```
postgres=# \dconfig enable_*
Parameter | Value
-----+-----
enable_convert_exists_as_lateral_join | on
enable_convert_in_values_to_any | on
enable_group_by_reordering | on
enable_index_path_selectivity | on
enable_join_pushdown | on
enable_self_join_removal | on
```



## Query Optimizer Options

Tantor Postgres version 17 introduces parameters that can be used to enable additional query planner optimizations. These optimizations were created by Tantor Labs to eliminate performance issues that arise in real applications, mainly 1C:ERP. When investigating the problem, queries with non-optimal plans were identified. With optimal plans, query execution time was reduced by orders of magnitude. By default, optimizations are enabled. The parameters were added to provide flexibility in configuring the planner and the ability to quickly check the effectiveness of optimizations.

`enable_convert_exists_as_lateral_join` allows the planner to convert subqueries with EXISTS to lateral SEMI JOINS when possible. This conversion can improve performance in correlated subqueries.

`enable_convert_in_values_to_any` enables a planner optimization that converts IN VALUES value lists to ANY constructs. This can simplify query plans and provide more efficient execution paths.

`enable_group_by_reordering` Whether the query planner will create a plan that will provide GROUP BY on columns sorted in column order corresponding to the columns on which the child node of the plan returns sorted data. For example, during an index scan. When optimization is disabled, the planner considers sort order only to service ORDER BY (if present).

`enable_index_path_selectivity` allows the planner to apply additional selectivity when evaluating join paths using indexes. By default, the planner chooses a composite index created on fewer columns because it is smaller in size and does not take into account that index entries point to a large number of rows that do not match the join condition. This parameter allows you to choose a more appropriate index.

`enable_join_pushdown` allows the planner to move inner joins into subqueries when doing so will not change the result. This transformation may allow more efficient join paths to be used.

`enable_self_join_removal` replaces table joins with equivalent constructs that allow the table to be scanned in a single pass. Affects regular (heap) tables only.

# Библиотека pg\_stat\_advisor

- automatically detects queries where the planner **underestimates** or **overestimates** the number of rows returned (actual rows differ from planned rows)
- If the discrepancy ratio exceeds the specified threshold, creates extended statistics on the table

```
set pg_stat_advisor.suggest_statistics_threshold = 0.33;
create table t (i int, j int);
insert into t select i/10, i/100 from generate_series(1, 1000000) i;
analyze t;
explain (analyze, buffers, timing off) select * from t where i = 100 and j = 10;
select pg_sleep(1);
\dx
```

| Schema | Name       | Definition  | Ndistinct | Dependencies | MCV     |
|--------|------------|-------------|-----------|--------------|---------|
| public | t_i_j_stat | i, j FROM t | defined   | defined      | defined |

```
\i cat $PGDATA/log/postgresql-*.log | grep pg_stat_advisor
LOG: pg_stat_advisor: successfully created extended statistics from public.t
```



## pg\_stat\_advisor library

**pg\_stat\_advisor** - the library automatically detects queries where the planner **underestimates** or **overestimates** the number of rows returned (actual rows differ from planned rows, which are compared). If **actual/planned** or **planned/actual**  $\geq$  `pg_stat_advisor.suggest_statistics_threshold`, it automatically generates and executes the `CREATE STATISTICS ON` command on the columns, and then executes the `ANALYZE` command to update statistics. The type of statistics is not specified, so all types of statistics are created (`mcv`, `ndistinct`, `dependencies`). The commands for creating statistics and updating statistics are run **asynchronously** background worker process.

the `shared_preload_libraries` parameter.

Working conditions:

1. INSERT, UPDATE, DELETE are not supported, only SELECT and WITH
2. Node is **not** NestedLoop, MergeJoin, HashJoin
3. No data is created from the temporary table
4. WHERE specifies **from 2** to 8 columns (**inclusive**) from one table
5. The table **has been parsed** and at least one column has `ndistinct`  $\neq$  1
6. Columns are not covered by a composite index (there is another optimization for this,

`enable_index_path_selectivity`)

```
set pg_stat_advisor.suggest_statistics_threshold = 0.33 ;
drop table if exists t;
create table t(i int, j int);
insert into t select i/10, i/100 from generate_series(1, 1000000) i;
analyze t;
explain (analyze, buffers, timing off) select * from t where i = 100 and j = 10;
-> Parallel Seq Scan on t (cost=0.00..10675.00 rows=1) (actual rows=3 loops=3)
select pg_sleep(1);
```

```
\dx
```

| Schema | Name       | Definition  | Ndistinct | Dependencies | MCV     |
|--------|------------|-------------|-----------|--------------|---------|
| public | t_i_j_stat | i, j FROM t | defined   | defined      | defined |

```
\i cat $PGDATA/log/postgresql-*.log | grep pg_stat_advisor
LOG: pg_stat_advisor: successfully created extended statistics from public.t
```

The patch has been submitted to the community <https://www.postgresql.org/message-id/aa034271-821c-42f3-92a1-b4112111c9c2%40tantorlabs.com>

# Options `enable_temp_memory_catalog` and `enable_delayed_temp_file`

- `enable_temp_memory_catalog` allows you to save metadata of temporary objects in the local memory of the process that operates on them and not make changes to the system catalog tables
  - › the speed of access to metadata of an already created temporary table is higher
  - › the way data is stored in temporary objects and the way data is accessed does not change
- `enable_delayed_temp_file` speeds up work with temporary tables by allowing temporary table files not to be created until the local buffer memory of the server process is sufficient.



## Configuration parameters `enable_temp_memory_catalog` and `enable_delayed_temp_file`

When creating and deleting temporary tables in PostgreSQL, changes are made to the system catalog tables, despite the fact that temporary tables are only accessible to the process in whose session they were created. This can lead to bloating of the system catalog tables and additional load on the instance from autovacuum processes. The most bloated are `pg_attribute`, `pg_class`, `pg_depend`, `pg_type`.

**`enable_temp_memory_catalog`** parameter allows you to save metadata of temporary objects in the local memory of the process that works with them and not make changes to the system catalog tables. The **`enable_temp_memory_catalog parameter`** eliminates changes to any system catalog tables when working with temporary tables. The parameter can be enabled at different levels, including the session level.

When using the parameter, the local memory of the server process is used to store metadata instead of system catalog table blocks. Using the parameter does not require setting memory allocation parameters (`work_mem`, `maintenance_work_mem`). The speed of access to the metadata of an already created temporary table is higher, since the metadata is stored in the local memory of the server process and there is no need to use locks to access the tables and indexes of the system catalog, which reduces contention. The parameter changes the method of storing metadata, the method of storing data in temporary objects and the method of accessing data do not change, except for the case when only temporary tables are affected in the transaction. If the transaction does not affect permanent storage objects, then the transaction is committed faster.

**`enable_delayed_temp_file`** parameter speeds up work with temporary tables (~15%), allowing you to avoid creating temporary table files while there is enough memory in the local buffer of the server process.



# enable\_large\_allocations parameter

- 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

**Increases the size of the 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 the StringBuffer. If it does not fit, then any client with which the server process works will receive an error, including the `pg_dump` and `pg_dumpall` utilities. The size of a table row field of all types cannot exceed 1 GB, but there can be several columns in the table and the size of the row can exceed both a gigabyte and several gigabytes.

`pg_dump` utility may refuse to dump such rows because it does not use the `WITH BINARY` option of the `COPY` command. For text fields, a non-printable character occupying one byte will be replaced by a sequence of printable characters occupying two bytes (for example, `\n`), and the text field may 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. The StringBuffer is allocated dynamically during the processing of each line, not when the server process starts. If there are no such lines, the parameter does not affect the operation of the server process.

This problem occurs with the row of the config table of the 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;
```

# pglz compression algorithm

- Tantor Postgres optimizes pglz data compression algorithm
- The optimization removes potentially redundant operations, increasing compression speed by 1.4x.
- Compression is used only for variable-width data types and is the default for most data types that can use compression.
- **pglz** compression algorithm is used by default for TOAST compression.

```
postgres=# \dconfig *compress*
List of configuration parameters
Parameter | Value
-----+-----
default_toast_compression | pglz
libpq_compression | off
wal_compression | off
```



## pglz compression algorithm

**pglz** data compression algorithm has been optimized in the Tantor Postgres DBMS . The optimization removes potentially redundant operations, increasing the compression speed by 1.4 times.

**pglz** compression algorithm is used by default for TOAST compression.

```
postgres=# \dconfig *compress*
List of configuration parameters
Parameter | Value
-----+-----
default_toast_compression | pglz
libpq_compression | off
wal_compression | off
(3 lines)
```

Compression is used only for variable-width data types (e.g. int is fixed-length and uncompressed, text is variable-length and compressed) and is used only when the column storage mode is MAIN or EXTENDED. EXTENDED is the default for most data types that support storage other than PLAIN. The storage mode can be set with the command:

```
ALTER TABLE name ALTER COLUMN column SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
MAIN };
```

The compression algorithm can be changed at the column level:

```
ALTER TABLE name ALTER COLUMN column SET COMPRESSION {DEFAULT | pglz | lz4};
```

Technical details of optimizations of the pglz algorithm code in the Tantor Postgres DBMS:

- 1) A more compact hash table is used with uint16 indexes instead of pointers.
- 2) The prev pointer in the hash table is ignored.
- 3) More efficient 4-byte comparison operations are used instead of 1-byte ones.

Also macro functions are replaced with regular functions (does not affect performance).

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-connection.html#GUC-LIBPQ-COMPRESSION](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-connection.html#GUC-LIBPQ-COMPRESSION)



# libpq\_compression parameter

- default value `off`
- defines the list of supported network traffic compression algorithms
- valid values: `off`, `on`, `lz4`, `zlib`
- You can set the compression level, for example:  

```
alter system set libpq_compression =
'lz4:1,zlib:2';
```



## libpq\_compression parameter

`libpq_compression` configuration option enables compression support in the `libpq` library, implemented by a new `libpq_compression` configuration option . The functionality can be used by client applications and drivers written in C or other languages that support API calls to C.

`libpq_compression` parameter can take the following values: `off`, `on`, `lz4`, `zlib` . By default, `libpq_compression = off` .

Compression is especially useful for importing/exporting data using the `COPY` command and for replication operations (both physical and logical). Compression can also improve response times for queries that return large amounts of data (e.g. JSON, BLOB, text, etc.)

This parameter controls the available compression methods for traffic between the client and the server. It allows you to reject compression requests even if the server supports this feature (for example, due to security or CPU consumption reasons). For more precise control, you can specify a list of allowed compression methods. For example, to allow only the `lz4` and `zlib` methods, you can set the parameter value to `lz4, zlib`. You can also specify the maximum compression level for each method, for example, by setting the parameter value to `lz4:1, zlib:2`, the maximum compression level for the `lz4` method will be set to 1, and for the `zlib` method to 2. If the client requests compression with a higher compression level, the maximum allowed level will be set. By default, the maximum possible compression level for each algorithm is 1.

Appeared starting with version 15.4 of Tantor Postgres, not present in vanilla PostgreSQL 17.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/runtime-config-connection.html#GUC-LIBPQ-COMPRESSION](https://docs.tantorlabs.ru/tdb/en/17_5/se/runtime-config-connection.html#GUC-LIBPQ-COMPRESSION)

# Parameter `wal_sender_stop_when_crc_failed`

- Reduces the likelihood of corrupted log records being transferred to replicas
- disabled by default
- When enabled, `walsender` processes will check the checksums of log records before transmitting them to clients. If the checksum does not match, the processes will try to read the record from the log buffer (WAL buffer). If there is no record in the log buffer or the checksum does not match, `walsender` will stop



## Parameter `wal_sender_stop_when_crc_failed`

The `wal_sender_stop_when_crc_failed` configuration parameter enables checksum verification of redo log records before they are transmitted to clients via the replication protocol. The `walsender` process is used to transmit redo log records to replicas and other clients (`pg_receivewal`), and reads WAL segments from the file system. Redo log records are protected by checksums, but by default `walsender` does not verify the checksums.

`wal_sender_stop_when_crc_failed` configuration parameter is set to `true`, `walsender` processes will check the checksums of log records before sending them to clients. If the checksum does not match, the processes will try to read the record from the WAL buffer. If there is no record in the log buffer or the checksum does not match, `walsender` will stop. This will prevent bad pages from propagating to replicas and WAL archives.

# backtrace\_on\_internal\_error parameter

- The parameter belongs to the Developer Options group, i.e. it is not used in industrial operation.
- If this option is enabled and an error with code XX000 (internal\_error) occurs, a stack trace is written to the diagnostic log along with the error message.
  - This is useful for debugging internal errors that don't normally occur in production.
- Disabled by default.



## backtrace\_on\_internal\_error parameter

This option is in the Developer Options group, meaning it is not used in production. If this option is enabled and an error with the XX000 code ( internal\_error ) occurs, the stack trace is written to the diagnostic log along with the error message. This is useful for debugging internal errors that do not typically occur in a production environment. Disabled by default.

# uuid\_v7 extension

- extension adds **high-speed** generation of uuid\_v7
- uuid\_v7 generates incremental values, **allowing** for btree index insertion optimization
  - › with very frequent insertions in the session, high speed, not slower than with bigserial
  - › the index structure remains optimal and compact
- uuid takes 16 bytes, bigint 8 bytes
- example of use:

```
create extension if not exists " uuid-oss" ;
create extension if not exists pg_uuidv7 ;
create table tt1 (id uuid default uuidv7() primary key, data bigint);
create table tt2 (id bigint generated by default as identity primary key, data bigint);
select count(*), pg_indexes_size('tt1') from tt1;
```



## uuid\_v7 extension

PostgreSQL has an optimization for inserting into a btree index, which allows not to descend from the root of the index tree. The server 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 optimization is used for the number of levels in the index starting from the second (macro in the PostgreSQL core code BTREE\_FASTPATH\_MIN\_LEVEL).

When using the uuid type as a unique key, `uuidv7()` generates increasing values and the optimization works. When using v4 (and others), there will be no optimization of fast insert, since random values are inserted, not increasing ones. Moreover, inserting into different leaf blocks of the index leads to an increase in the volume of the log due to writing more full page images (FPI) to the log.

Example test:

```
pgbench -i
echo "insert into tt1(data) values(1);" > txn.sql
create extension if not exists "uuid-oss";
create extension if not exists pg_uuidv7;
create table tt1 (id uuid default uuidv7() primary key, data bigint);
vacuum analyze tt1;
pgbench -T 30 -c 16 -f txn.sql
select count(*), pg_indexes_size('tt1') from tt1;
drop table if exists tt1;
create table tt1 (id bigint generated by default as identity primary key, data
bigint);
```

Скорость вставки сравнима: для `uuidv7()` tps = 1734, для `bigint` tps = 1707.

The sizes of indexes on a uuid column are larger than the size of an index on a bigint column because the size of a uuid field (16 bytes) is twice as large as the size of a bigint field (8 bytes). For `uuidv7`, the number of rows in the test example is 97172, and the index size is 3088384 bytes, for `bigint`, the number of rows is 99050, and the index size is 2236416 bytes.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pg\\_uuidv7.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pg_uuidv7.html)

# pg\_tde (Transparent Data Encryption) Extension

- Implements "transparent" data encryption (Transparent Data Encryption)
- Transparency means that the client receives and transmits unencrypted data.
- Does not encrypt data in memory (in the buffer cache) and when transmitted over the network
- Encryption is performed by protocols: AES, Magma, Kuznyechik, ChaCha20
- Physical and logical replication are supported
- System catalog tables are not encrypted.
- pg\_rewind does not work with encrypted WAL yet
- WALs are encrypted completely



## pg\_tde (Transparent Data Encryption) Extension

Implements "transparent" data encryption (Transparent Data Encryption). Transparency means that the client receives and transmits unencrypted data. The purpose of this option is to prevent access to encrypted data when stealing cluster data files and log (WAL) files without stealing files (devices) with keys. pg\_tde does not encrypt data in memory (in the buffer cache) and when transmitting over the network. On Astra Linux, the lib **gost**-astra package configures OpenSSL automatically and encryption is performed by protocols with a symmetric key: AES, Magma, Kuznyechik, ChaCha20.

You can encrypt existing tables:

```
ALTER TABLE t SET ACCESS METHOD tde_heap;
```

A configuration parameter can be set to encrypt the tables created:

```
ALTER SYSTEM SET default_table_access_method = tde_heap;
```

```
SET default_table_access_method = tde_heap;
```

The tde\_heap access method works on top of the heap access method. The buffer cache stores data in unencrypted form.

Only rotation of the main key is implemented. Each file is encrypted block by block (8Kb) with its own key. To rotate the keys that encrypt the files, it would be necessary to re-encrypt the files.

Peculiarities:

- 1) Physical and logical replication are supported.
- 2) System catalog tables are not encrypted.
- 3) pg\_rewind does not work with encrypted WAL yet, this will be implemented in future versions.
- 4) WAL-G does not support sending WAL deltas if the WAL is encrypted.
- 5) WAL are encrypted completely. Tables (including temporary ones) are encrypted with dependent objects: TOAST, indexes.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pg\\_tde.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pg_tde.html)

# Validator oauth\_base\_validator

- Tantor Postgres 17 has an **oauth** (OAuth 2.0) authentication method, which will appear in PostgreSQL version 18
- This method uses an external service for authentication.
- **oauth** method is inserted into the fourth field of the line in the `pg_hba.conf` file

```
#TYPE DATABASE USER ADDRESS METHOD
local all all oauth issuer="http://1.1.1.1:80/realms/a" scope="openid" map="o1"
```

- To use the oauth authentication method, you need to write a "validator" in C
- In Tantor Postgres, the library with the validator is supplied
- Example of authentication in psql:

```
psql "user=alice dbname=postgres oauth_issuer=http://1.1.1.1:80/realms/a oauth_client_id=user1
oauth_client_secret=AbCdEf123GhIjKl"
Visit http://1.1.1.1:80/realms/a/device and enter the code: XYZX-XYZO
postgres=>
```



## Validator oauth\_base\_validator

Tantor Postgres 17 has an **oauth** (OAuth 2.0) authentication method that will appear in PostgreSQL version 18. This method, similar to **radius**, uses an external service for authentication. The **oauth method** is inserted into the fourth field of the line in the `pg_hba.conf` file. Example:

```
#TYPE DB USER ADDR METHOD
local all all oauth issuer="http://1.1.1.1:80/realms/a" scope="openid" map="o1"
Names can be matched via pg_ident.conf :
MAP SYSTEM-USERNAME PG-USERNAME
o1 "0fc72b6f-6221-4ed8-a916-069e7a081d14" "alice"
```

It is possible to map via the validator code, if it is implemented in the validator. In this case, instead of `map="o1"` in the line `pg_hba.conf` you need to insert the option `delegate_ident_mapping=1`

To use the **oauth authentication method**, you need to write a "validator" in C. The library name is specified in the configuration parameter:

```
alter system set oauth_validator_libraries = 'oauth_base_validator';
```

Tantor Postgres comes with a library with a validator.

To be able to use the http protocol, you need to use an environment variable:

```
export PGOAUTHDEBUG="UNSAFE"
```

and run the client:

```
psql "user=alice dbname=postgres oauth_issuer=http://1.1.1.1:80/realms/a
oauth_client_id=user1 oauth_client_secret=AbCdEf123GhIjKl"
```

A message will appear telling you where to go and what code to enter.

Visit `http://1.1.1.1:80/realms/a/device` and enter the code: XYZX-XYZO

After entering the code at the external service address, the connection will be established and **psql** will prompt you:

```
postgres=>
```

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/oauth-base-validator.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/oauth-base-validator.html)

# credcheck library

- Uses a library that can be loaded at the cluster level ( `shared_preload_libraries` parameter ) and for a single session (using the `LOAD credcheck` command )
- When loading, it registers 30 configuration parameters that can be used to set password complexity checks, protection against password guessing, password reuse parameters, a list of roles for which checks do not apply, etc.
- The extension has 8 functions and 2 views.
- The extension is triggered when a role is created, renamed, password is changed, authentication is performed.

```
postgres=# \dconfig credcheck.*
 Parameter | Value
-----+-----
 credcheck.auth_delay_ms | 0
 credcheck.encrypted_password_allowed | off
 credcheck.max_auth_failure | 0
 ...
 credcheck.whitelist |
(30 rows)
```



## credcheck library

Uses a library that can be loaded at the cluster level ( `shared_preload_libraries` parameter ) and for a single session (using the `LOAD credcheck` command ). When loading, it registers 30 configuration parameters that can be used to set password complexity checks, password guessing protection, password reuse parameters, a list of roles that are not affected by checks, etc.

```
postgres=# LOAD 'credcheck';
postgres=# CREATE EXTENSION credcheck;
postgres=# \dconfig credcheck.*
 Parameter | Value
-----+-----
 credcheck.auth_delay_ms | 0
 credcheck.encrypted_password_allowed | off
 credcheck.max_auth_failure | 0
 credcheck.no_password_logging | on
 credcheck.password_contain |
 ...
 credcheck.username_min_upper | 0
 credcheck.username_not_contain |
 credcheck.whitelist |
(30 rows)
```

You can install an extension that has 8 functions and 2 views.

The extension is triggered when a role is created, renamed, password is changed, authentication occurs.

The `credcheck.max_auth_failure` parameter: the number of unsuccessful authentication attempts allowed for a user before blocking. The `credcheck.auth_delay_ms` parameter allows you to enter a delay in case of unsuccessful password entry, which protects against password guessing. To protect against password guessing, you can use the standard `auth_delay` extension , but this method of protection against guessing **aggravates** DDOS attacks, since server processes hold resources for the duration of the delay, unlike blocking accounts.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/credcheck.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/credcheck.html)

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/auth-delay.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/auth-delay.html)



# fasttrun and online\_analyze extensions

- Truncate temporary tables updates pg\_class table
- The fasttrun extension contains the **fasttruncate function**
  - › the files name does not change
  - › system catalog tables do not change
  - › can be used instead of the TRUNCATE command
  - › works only with temporary tables
- After truncation or other commands, statistics can be collected
- To automatically **collect statistics**, you can use the online\_analyze extension
  - › The extension is configurable via configuration parameters

```
select fasttruncate ('tt');
INFO: analyzing "pg_temp_5.tt"
INFO: "tt": scanned 0 of 0 pages, containing 0 live rows and 0 dead rows; 0 rows in sample, 0 estimated
total rows
INFO: analyze "tt" took 0.00 seconds
fasttruncate

(1 row)
```



## fasttrun and online\_analyze extensions

TRUNCATE temporary table results in deletion and creation of files with new names, row in pg\_class is updated. Old row versions may not be purged if the database horizon is long and pg\_class and indexes become bloated.

The fasttrun extension consists of one function `fasttruncate('name')` . When using the function, the temporary table is truncated, the file names do not change. 1C applications use this function call instead of the TRUNCATE command. The function works only with temporary tables:

```
select fasttruncate('t');
ERROR: Relation isn't a temporary table
```

To use the extension, you need to download the library and install the extension:

```
alter system set shared_preload_libraries = fasttrun, fulleq, mchar;
create extension fasttrun;
```

After inserting or changing rows in temporary tables, it may be useful to re-compile statistics for the scheduler. 1C Enterprise, starting with version 8.3.13, executes the ANALYZE command after inserting rows into a temporary table. For other applications that do not do this, you can use the `online_analyze` extension . You should not load it for all sessions, because if statistics are collected by a separate command, the automatic collection does not know about it and repeats the same action, which leads to unnecessary resource consumption. Moreover, statistics are collected synchronously, which slows down the execution of commands that trigger the extension. An example of using the extension at the session level:

```
load 'online_analyze';
set online_analyze.enable = on;
set "online_analyze.verbose" = on;
set online_analyze.table_type = 'temporary';
```

The double quotes around the second parameter are necessary because verbose is a reserved word. This parameter executes the ANALYZE VERBOSE command. After the command that results in the analysis is executed, INFO-level notifications are sent to the caller of the command.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/fasttrun.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/fasttrun.html)

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/online\\_analyze.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/online_analyze.html)



# mchar extension

- Adds support for mchar, mvvarchar types for compatibility with Microsoft SQL Server
- For mchar and mvvarchar types, defines functions and operators:
  - › length()
  - › substr(str, pos[, length])
  - › || concatenation with different types (mchar || mvvarchar)
  - › < <= = >= > case insensitive comparison (ICU)
  - › &< &<= &= &>= &> case-sensitive comparison (ICU)
  - › LIKE
  - › SIMILAR TO
  - › ~ (regular expressions)
- Add implicit typecasting from mchar to mvvarchar and back
- support for b-tree and hash index types
- Using Indexes to Perform the LIKE Operator



## mchar extension

Adds support for mchar, mvvarchar types for compatibility with Microsoft SQL Server. The following functions and operators are supported for mchar and mvvarchar types:

length()  
substr(str, pos[, length])  
|| concatenation with different types (mchar || mvvarchar)  
< <= = >= > case insensitive comparison (ICU)  
&< &<= &= &>= &> case-sensitive comparison (ICU)  
LIKE  
SIMILAR TO  
~ (regular expressions)

Implicit casting of mchar to mvvarchar and back

Support for b-tree and hash index types

Using Indexes to Perform the LIKE Operator

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se1c/mchar.html](https://docs.tantorlabs.ru/tdb/en/17_5/se1c/mchar.html)

# fulleq extension

- When using the "=" operator to compare values, if at least one of the operands is NULL, the result is NULL.
- In 1C applications, the "==" operator is often used, which returns true when the operands are equal or both have the NULL value, which is convenient when working with databases, especially with 1C, where the operators and semantics of working with NULL differ from the SQL standard.
- The "==" operator from the fulleq extension allows you to perform highly efficient value comparisons using the desired logic.



## fulleq extension

When using the "=" operator to compare values, if at least one of the operands is NULL, the result is NULL. In 1C applications, the "==" operator is often used, which returns true when the operands are equal or both are NULL. This is convenient when working with databases, especially with 1C, where the operators and semantics for working with NULL differ from the SQL standard.

The "==" operator allows you to perform highly efficient value comparisons using the desired logic.

The "==" operator, when applied to two operands, returns true if they are equal or both are NULL.

The "==" operator, when applied to two operands, returns false if they are not equal or if one of them is NULL.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se1c/fulleq.html](https://docs.tantorlabs.ru/tdb/en/17_5/se1c/fulleq.html)

# orafce extension

- An extension that provides features that simplify the migration of application code from Oracle Database
- orafce emulates some of the Oracle Database functionality and packages
- Creates a large number of functions and other objects and data types that work the same way as their Oracle Database counterparts
- The extension simplifies the migration of application code to Tantor Postgres from Oracle Database
- The extension creates 15 schemas in which the extension objects reside. The names of the eight schemas correspond to the names of packages in Oracle Database.



## orafce extension

The orafce extension is present in the Tantor Postgres SE build.

The extension contains functions and data types that are similar to those in Oracle Database.

The orafce functions and operators emulate some of the functions found in commonly used Oracle Database procedure packages.

Using orafce reduces migration time and reduces the labor intensity of migrating application code.

When migrating from Oracle Database to Tantor Postgres, commands and program code may use functions, procedures, and data types that are available in Oracle Database and not in PostgreSQL or the SQL standard. Rewriting code can be quite labor-intensive, especially if there are many commands.

The orafce extension creates a large number of functions that work similarly to the functions and procedures of the same name in Oracle Database.

These are the most common routines that are most often used in application code that works with Oracle Database. The extension does not cover the entire set of functions, also the syntax for calling some functions may differ, and you should not assume that SQL commands that were executed in Oracle Database will be executed in postgres.

The purpose of the extension is to simplify code migration, enable code execution without significant changes, and gradually rewrite and optimize the execution of SQL commands.

Also, the functions from this extension can be useful on their own.

In Oracle Database, program units (functions and procedures) are contained in "packages".

Postgres has a "schema" object that has similar functionality, so the extension creates quite a large number of schemas whose names correspond to the names of packages in Oracle Database.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/orafce.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/orafce.html)

# http extension

- The http extension provides the ability to execute HTTP and HTTPS requests directly from SQL
- Installed into the database using the `create extension http` command;
- you can create a trigger that calls a web service, passes data and gets a result that can be used in the trigger logic
- pgsql-http can be useful in:
  - Integrations with external APIs
  - Interactive applications
  - Real-time data processing



## http extension

The http extension is available in Tantor Postgres SE.

Installed into the database using the `create extension http` command;

The http extension provides the ability to execute HTTP and HTTPS requests directly from SQL.

For example, you can create a trigger that calls a web service, send data, and get a result that can be used in the trigger logic. Using the HTTP protocol requires caution. In particular, you should avoid creating situations where the server process is blocked due to a long wait for a response to an HTTP request.

The pgsql-http functionality can be useful in the following tasks:

1) Integration with external APIs: in some cases it is more convenient to work via the REST protocol directly from the database, especially when the data received from the web service needs to be used in SQL commands. The pgsql-http extension allows you to do this by supporting all the main HTTP protocol methods, including GET, POST, PUT, DELETE, and the relatively new PATCH method.

2) Interactive applications: In some use cases, PostgreSQL can be part of an interactive web application where the database interacts with the user via HTTP. http can be used to send requests to the application server and receive responses to them.

3) Real-time data processing: allows access to data that is constantly updated and available to clients via the HTTP protocol. Using http, you can request this data directly from the database server.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pgsql-http.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pgsql-http.html)

# pg\_store\_plans extension

- Extension for tracking SQL query execution statistics
- saves full query plans, not just their statistics or text
- Using pg\_store\_plans may increase the load on the system due to collecting and storing additional data.



## pg\_store\_plans extension

All versions of Tantor Postgres include the pg\_store\_plans extension.

The extension provides the means to track the execution plan statistics of all SQL queries.

Used by the Tantor platform to collect query plan statistics.

Unlike other tools such as auto\_explain, pg\_stat\_statements, or pg\_stat\_plans, pg\_store\_plans is capable of collecting and storing full query plans, not just statistics or query text.

Allows you to analyze how requests are executed in the system.

Using pg\_store\_plans may increase the load on your system due to the additional collection and storage of query plan information.

pg\_store\_plans:

- 1) Automatically saves query execution plans, allowing you to explore how queries execute in your database.
- 2) Stores query plans over time, allowing you to analyze historical data and determine how changes in application code or database affect query performance.
- 3) You can identify slow queries and determine which operations in a query plan take the most time. This can help you optimize your queries and improve database performance.
- 4) Compatible with other extensions. pg\_store\_plans can be used together with other extensions such as pg\_stat\_statements and pg\_qualstats.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pg\\_store\\_plans.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pg_store_plans.html)

# pg\_variables extension

- An extension that allows you to create and use variables in SQL queries
- are not transactional by default, but can be transactional
- are stored in the memory of the server process
- exist only in the current user session
- can be used on physical replicas, temporary tables cannot
- the speed of work is comparable to temporary tables
- no overhead (doesn't require a real transaction number, doesn't create files)



## pg\_variables extension

Tantor Postgres SE has a pg\_variables extension.

The pg\_variables extension allows you to define and use variables inside SQL queries on a PostgreSQL server.

Variables can be used to store temporary values, exchange data between functions, store intermediate results, etc.

Provides a means to track execution plan statistics for all SQL queries executed by the Tantor server.

Provides functions for working with variables of various types. Created variables exist only in the current user session.

By default, created variables are not transactional (i.e. they are not affected by BEGIN, COMMIT, ROLLBACK commands).

The extension allows storing in the server process memory the values of variables of various types, including: numeric, text, date-time, logical, jsonb, arrays, composite types. Variables are available within the session.

Variables can be used as an alternative to temporary tables. You can work with sets of values using the pgv\_select and pgv\_insert functions. The speed of work can be higher than when working with data using temporary tables. Variables **can** be used on physical replicas, temporary tables cannot. Variables can have a composite type, including row image.

There are no overhead costs: no real transaction number is required, no files are used, the contents of system catalog tables are not changed, and the operating system cache is not used. There is no performance degradation when actively changing variable values, which is typical when actively changing rows in temporary tables. Using the pgv\_stats function, you can see how much memory is used.

The functionality is similar to the variables of packages and application contexts in Oracle Database, so the extension can be used when migrating applications to the Tantor Postgres DBMS.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pg\\_variables.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pg_variables.html)

# Performance when using pg\_variables

- When comparing the speed of access to tables in memory, temporary, normal, the results are not obvious
- The pg\_variables extension is not standard and is not popular due to its awkward syntax and poor performance.

```
select * from pgv_select('bookings', 'tickets', '0005432020304' ::char(13)) as (ticket_no character(13),
book_ref character(6), passenger_id character varying(20), passenger_name text, contact_data jsonb) ;
Time: 0.281 ms
select * from tickets where ticket_no='0005432020304';
Time: 0.266 ms
select book_ref from tickets where passenger_name like '%G IVANOV' limit 1;
Time: 0.463 ms
select book_ref from tickets1 where passenger_name like '%G IVANOV' limit 1;
Time: 1.169 ms
select book_ref from pgv_select('bookings', 'tickets', '0005432020304'::char(13)) as (ticket_no
character(13), book_ref character(6), passenger_id character varying(20), passenger_name text, contact_data
jsonb) where passenger_name like '%G IVANOV' limit 1;
Time: 0.185 ms
```



## Benefits of the pg\_variables extension

Using the pg\_variables extension functions, you can store both scalar variables and composite types (row images). Row searches can be performed by full scanning or by hash. Structures are stored in the local memory of the process, and there is no point in using other methods such as btree.

```
wget https://edu.postgrespro.com/demo-medium-en.zip
zcat demo-medium.zip | psql
psql -d demo
create extension pg_variables;
demo=# \o t.tmp
\timing on\\
select pgv_insert('bookings', 'tickets', tickets) from tickets;
Time: 1634.973 ms (00:01.635)
demo=# create temp table tickets1 as select * from tickets;
Time: 557.808 ms
select * from tickets1 where ticket_no='0005432020304';
Time: 269.005 ms
select * from tickets where ticket_no='0005432020304';
Time: 0.266 ms
select * from pgv_select('bookings', 'tickets', '0005432020304'::char(13)) as (ticket_no character(13), book_ref
character(6), passenger_id character varying(20), passenger_name text, contact_data jsonb);
 ticket_no | book_ref | passenger_id | passenger_name | contact_data
-----+-----+-----+-----+-----
0005432020304 | F5C81C | 7257 672943 | OLEG IVANOV | {"email": "oleg-ivanov_1984@postgrespro.ru", "phone":
"+70632852802"}
(1 row)
Time: 0.281 ms
```

The speed of selection from the in-memory table is slightly slower than selection from a regular table by index. No index was created on the temporary table. If you create a btree index:

```
create index on tickets1(ticket_no);
Time: 5615.559 ms (00:05.616)
select * from tickets1 where ticket_no='0005432020304';
Time: 0.302 ms
```

then the speed of index access is wide and does not differ from a regular table.

```
select book_ref from tickets where passenger_name like '%G IVANOV' limit 1;
Time: 0.463 ms
select book_ref from tickets1 where passenger_name like '%G IVANOV' limit 1;
Time: 1.169 ms
select book_ref from pgv_select('bookings', 'tickets', '0005432020304'::char(13)) as (ticket_no character(13),
book_ref character(6), passenger_id character varying(20), passenger_name text, contact_data jsonb) where
passenger_name like '%G IVANOV' limit 1;
Time: 0.185 ms
https://pgconf.ru/media//2019/02/08/zakirov-pg-variables-pgconf-ru-2019.pdf
```

# Benefits of the pg\_variables extension

- Временные таблицы нельзя использовать на репликах
- pg\_variables extension functions can be used on replicas just like on the master
- pg\_variables stores data only in the local memory of the server process and does not use temporary files.
- Variables can be transactional or not.
- The saved objects are subject to a string buffer limit of 1GB.

```
select pgv_insert('bookings','t2', pgbench_branches) from pgbench_branches;
select * from pgv_select('bookings','t2',1) as (bid int, bbalance int, filler character(88));
bid | bbalance | filler
-----+-----+-----
1 | 0 |
select pgv_select('bookings','t2',1);
pgv_select

(1,0,)
```



## Benefits of the pg\_variables extension

**Temporary tables cannot be used on replicas** . The main advantage of the pg\_variables extension is that its temporary data storage capabilities can be used on replicas in the same way as on the master. This allows complex analytics that require storing intermediate data to be implemented on replicas and transferred to replicas.

It is important to remember that pg\_variables stores data **only in the local memory** of the server process and does not use temporary files. The saved objects are limited by the 1GB string buffer. This is not a problem, since similar functionality in other DBMSs has similar memory limitations. The disadvantage of pg\_variables is its inconvenience (unusualness) of use, which can be bypassed. For example, the function produces strings instead of the number of inserted strings, which generates network traffic if the function is called from the client:

```
select pgv_insert('bookings','t2', pgbench_branches) from pgbench_branches;
pgv_insert

```

(1 row)

When selecting composite types, you have to specify the structure details:

```
select * from pgv_select('bookings','t2',1) as (bid int, bbalance int, filler
character(88)) ;
```

```
bid | bbalance | filler
-----+-----+-----
1 | 0 |
```

```
select pgv_select('bookings','t2',1);
```

```
pgv_select

```

(1,0,)

One of the advantages of the extension is that it is possible to create transactional variables, i.e. changes to values can be changed atomically upon transaction commit, rolled back. By default, non-transactional variables are created.

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pg\\_variables.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pg_variables.html)



# pg\_stat\_kcache extension

- complements and depends on pg\_stat\_statements
- collects linux statistics by executing getrusage system call after each command execution
- 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 |         |            |



## pg\_stat\_kcache extension

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. The `shared_blks_read` statistics do not distinguish whether the pages (4K in size) that make up a block (8K in size) were in the Linux page cache or were read from disk. The extension allows this distinction, it collects Linux statistics by executing the `getrusage` system call after each command. The statistics collected by the extension can be useful for determining the effectiveness of caching and possible bottlenecks. The data collected by the system call is written to shared memory.

`getrusage` call is also used by the `log_executor_stats=on` configuration parameter (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 command-level detail. 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        |
| <b>pg_stat_kcache</b>      | 150351488 | 992  | 1024           | <b>1024</b>    |
| <b>pg_stat_kcache hash</b> | 150352512 | 2896 | 2944           | <b>1373056</b> |

The extension has the following parameters:

**\dconfig \*kcache\***

`pg_stat_kcache.linux_hz` (default -1) is automatically set to the value of the `linux CONFIG_HZ` parameter and is used to compensate for sampling errors. No need to change.

`pg_stat_kcache.track=top` parameter - analog of `pg_stat_statements.track`

`pg_stat_kcache.track_planning=off` analogue of `pg_stat_statements.track_planning`

# Statistics collected by pg\_stat\_kcache

Statistics in the `pg_stat_kcache` and `pg_stat_kcache_detail` views :

- `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
- `msgrcv` 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 by 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 precision. 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_msgsends IPC messages sent
exec_msgrcv IPC messages received
exec_nsignals signals received
exec_nvcsws voluntary context switches
exec_nivcsws involuntary context switches
```

# pg\_wait\_sampling extension

- included in all Tantor Postgres builds
- returns statistics on wait events of all processes in the instance
- 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 ;
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 queryids.
- 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
```



## pg\_wait\_sampling extension

The extension is included in all Tantor Postgres builds. 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` so that `pg_wait_sampling` does not overwrite the queryids 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 samples at 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 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
```

# Waiting Event History

- The extension uses "sampling" with a frequency of 1 millisecond (default is 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
  - › memory for 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
```



## Waiting Event History

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 data 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 | 2035-11-11 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
```

The largest part 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 processes in the instance. 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). The memory for queryid is equal to the maximum number of PIDs multiplied by 8 bytes (the size of the bigint type used by queryid ).

# pg\_background extension

- The extension allows asynchronously (in the background) to execute an arbitrary command and implement arbitrary tasks that need to be performed by the application or administrator. The tasks will be executed by the background processes of the instance.
- The extension contains the following functions:
  - > `pg_background_launch()`
  - > `pg_background_result()`
  - > `pg_background_detach()`



## Расширение pg\_background

The `pg_background` extension is available in Tantor SE and Tantor BE.

The extension allows arbitrary operations to be performed asynchronously (in the background). Using the extension, arbitrary tasks can be manually implemented that need to be performed in the background by an application or an administrator. The tasks will be executed by the instance's background processes. The extension provides a programming interface for launching and interacting with background processes, eliminating the need to use a low-level process interaction interface that requires C programming.

The extension contains the following functions:

`pg_background_launch` - Accepts the SQL command the user wants to execute and the size of the queue buffer. This function returns the ID of the background worker process.

`pg_background_result` - Takes a process ID as an input parameter and returns the result of the executed command through a background worker process.

`pg_background_detach` - Takes a process ID and detaches a background process that is waiting for the user to read its results.

# pgaudit and pgaudittofile extensions

- `pgaudit` and `pgauditlogtofile` extensions can be used to direct messages about session creation and duration to a separate audit file or files
- For extensions to work, you need to download their libraries:

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile;
```

- Extensions operate independently and in parallel with the cluster log and are controlled by their own parameters, which are prefixed with "`pgaudit.`"
- `pgaudit.log_connections` and `pgaudit.log_disconnections` parameters are similar to the PostgreSQL parameters of the same name and can create similar entries in a separate audit file



## pgaudit and pgaudittofile extensions

When using the `log_connections` and `log_disconnections` parameters, messages are written to the cluster log. During production use, many other messages are written to this log. Logging connections is not needed for routine analysis and clutters the general log, making it difficult to read more important messages. It is desirable that logging of connections, ddl commands, and other commands be performed not in the cluster log, but in a separate file or files.

Tantor DBMS has `pgaudit` and `pgauditlogtofile` extensions, which can be used to direct messages about session creation and duration to a separate audit file or files. `pgauditlogtofile` extension redirects the records created by the `pgaudit` extension to a separate file or files. Without it, the records go to the cluster log. The `pgauditlogtofile` extension depends on the `pgaudit` extension and does not work without it. To use the extensions, you only need to load **two libraries**:

```
alter system set shared_preload_libraries = pgaudit, pgauditlogtofile ;
```

Extension libraries register configuration parameters in the instance, which can be used to customize what is logged and where. Extensions operate independently and in parallel with the cluster log and are controlled by their own parameters, which are prefixed with "`pgaudit.`"

The extension has 18 parameters. 7 parameters are related to the `pgauditlogtofile` library, including the `pgaudit.log_connections` and `pgaudit.log_disconnections` parameters. These parameters are similar to the PostgreSQL parameters of the same name and can create similar records, but only **in a separate audit file, not in the cluster log, which is a big advantage of these parameters**. The advantage outweighs the disadvantages in the form of the need to load two libraries and the inconvenience of their use. Library parameters are set only at the cluster level, specifying these parameters in the environment variable leads to an error and the inability to connect, unlike the standard parameters: `export PGOPTIONS="-c pgaudit.log_connections=off"`

**psql**

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: parameter "pgaudit.log_connections" cannot be changed now
```

`pgaudit.log_disconnections` parameter, unlike the `log_disconnections` parameter, cannot be set when creating a session.



# Configuring pgaudit and pgaudittofile extensions

- **seven** parameters are related to the `pgauditlogtofile` library
- To create an audit log, you need to set the `pgaudit.log` parameter to at least 'misc'
  - > 'none' - no audit log file is created
  - > 'role' and 'ddl' `pgaudit.log_connections` and `pgaudit.log_disconnections` have no effect

```
postgres=# \dconfig pgaudit*
 List of configuration parameters
 Parameter | Value
-----+-----
pgaudit.log | none
pgaudit.log_autoclose_minutes | 0
pgaudit.log_catalog | on
pgaudit.log_client | off
pgaudit.log_connections | off
pgaudit.log_directory | log
pgaudit.log_disconnections | off
pgaudit.log_filename | audit-%F.log
pgaudit.log_level | log
pgaudit.log_parameter | off
pgaudit.log_parameter_max_size | 0
pgaudit.log_relation | off
pgaudit.log_rotation_age | 1d
pgaudit.log_rotation_size | 0
pgaudit.log_rows | off
pgaudit.log_statement | on
pgaudit.log_statement_once | off
pgaudit.role |
(18 rows)
```



## Configuring pgaudit and pgaudittofile extensions

The disadvantage of using extension parameters is that you need to set the `pgaudit.log` parameter to at least 'misc' to create an audit log. But the value 'misc' forces the logging of DISCARD, FETCH, CHECKPOINT, VACUUM, SET commands and bloats the audit log. **With the default value 'none', no log file is created. When set to 'role' and 'ddl', the parameters `pgaudit.log_connections` and `pgaudit.log_disconnections` have no effect.**

Installing the `pgauditlogtofile` extension with the command is useless because there are no objects in the extension:

```
create extension pgauditlogtofile;
\dx+ pgauditlogtofile
Objects in extension "pgauditlogtofile"

(0 rows)
```

`pgaudit` extension includes two triggers and two trigger functions:

```
event trigger pgaudit_ddl_command_end
event trigger pgaudit_sql_drop
function pgaudit_ddl_command_end()
function pgaudit_sql_drop()
```

The substitution variable '`%F`' (or its equivalent `%Y-%m-%d`) in the audit log and cluster log names is more convenient than the default value (`%Y%m%d_%H%M`) in that it does not create a separate file when the instance is restarted. A new file is created once per day. Example of setting values:

```
alter system set pgaudit.log_filename = 'audit- %F .log';
alter system set log_filename = 'postgresql- %F .log';
```

# pgcopydb utility

- The utility automates copying a database to another cluster
- A typical use case for pgcopydb is migrating to a new major version of PostgreSQL while minimizing downtime.
- The utility implements the task of parallelization with streaming data transfer according to the logic "pg\_dump -jN | pg\_restore -jN" between two working clusters, conducting these utilities
- Supports parallel index creation, change tracking and application, resuming interrupted overload, object filtering
- Open source project



## pgcopydb utility

The utility automates copying a database to another cluster. A typical use case for pgcopydb is migration to a new major version of PostgreSQL with minimal downtime. The utility implements the task of parallelization with streaming data transfer according to the logic of "pg\_dump -jN | pg\_restore -jN" between two running clusters, conducting these utilities. Supports parallel index creation, change tracking and application, resuming interrupted overload, filtering objects.

pgcopydb - open source project <https://github.com/dimitri/pgcopydb>  
[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pgcopydb.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pgcopydb.html)



# pg\_anon utility

A program written in python performs:

- search for confidential data in the database
- creating a dictionary based on search results
- saving and restoring using dictionary
- synchronizing the contents or structure of specified tables between the source and target databases
- data unloading with masking (depersonalization, anonymization) according to specified templates



## pg\_anon utility

pg\_anon is an application written in python.

The application performs:

`anon_funcs` schema in databases , which contains a set of functions for masking (depersonalization, anonymization) of data.

Search for sensitive data in a dictionary-based database.

Creating a dictionary based on search results (reconnaissance).

Saving and restoring using a dictionary. Separate dictionary files can be provided for different databases.

Synchronize the contents or structure of specified tables between the source and target databases.

The application is downloaded and installed separately from the package.

[https://github.com/TantorLabs/pg\\_anon](https://github.com/TantorLabs/pg_anon)

There is also an extension **transp\_anon** - transparent anonymization of values on the fly of query results from clients

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/transp\\_anon.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/transp_anon.html)

# pg\_configurator utility

- It is a python script `pg_configurator`, installed in `/usr/bin`
- suggests recommended configuration parameters based on hardware resource characteristics such as available memory, number of processors, disk space, etc. This allows for optimal use of available resources and increases instance performance



## pg\_configurator utility

The `pg_configurator` application is available for all Tantor Postgres DBMS builds. The application is supplied separately, for example as a package.

It is a python script `pg_configurator`, installed at `/usr/bin`.

`pg_configurator` suggests recommended configuration parameters based on hardware resource characteristics such as available memory, number of processors, disk space, etc. This allows you to make optimal use of available resources and improve instance performance.

Project page [https://github.com/TantorLabs/pg\\_configurator](https://github.com/TantorLabs/pg_configurator)

Web version of the configurator: <https://tantorlabs.ru/pgconfigurator>

# pg\_diag\_setup.py utility

- The task performed by the utility is to call a utility on the database cluster host that will install and configure extension parameters according to a template and reserve the parameter values so that changes can be restored.
- Checks for extension availability via `pg_available_extensions`
- Updates the value of `shared_preload_libraries` without overwriting existing libraries
- Adds new parameters to the end of `postgresql.conf`
- Creates a backup text file with the values of the configurable parameters with a timestamp
- Allows you to roll back changes to any backup



## The pg\_diag\_setup.py utility

It is a script written in python. The task solved by the utility is to call the utility on the database cluster host, which will install and configure extension parameters according to the template and reserve the parameter values so that the changes can be restored. It is assumed that the parameters of diagnostic extensions such as `pg_store_plans`, `pg_stat_statements`, `pg_stat_kcache`, `auto_explain`, `pg_buffercache`, `pg_trace`, `pg_wait_sampling` will be configured.

The utility does not restart the instance after making changes.

When running the utility:

- 1) reads configuration parameter files, taking into account `include* parameters`
- 2) creates a list of parameters specifying the source file
- 3) reads its `default.yaml` settings file , which specifies **the extensions to be configured** and **their configuration settings** . Example file contents for an extension

```
pg_stat_statements :
shared_preload_lib: pg_stat_statements
create_cmd: CREATE EXTENSION pg_stat_statements
params :
pg_stat_statements.max: 10000
pg_stat_statements.track:all
pg_stat_statements.track_utility: "on"
pg_stat_statements.track_planning: "off"
pg_stat_statements.save: "on"
```

- 4) Checks for extension availability via `pg_available_extensions` by connecting to the instance via a Unix socket
- 5) Updates the value of `shared_preload_libraries` with the `ALTER SYSTEM` command without overwriting existing libraries, installs extensions (if possible)
- 6) Adds new parameters to the end of `postgresql.conf` , marking the added parameters with the comment "Added by pg\_diag\_setup"
- 8) Creates a backup text file with the values of the configured parameters with a timestamp
- 9) Allows you to roll back changes to any backup created by the utility

# pg\_sec\_check utility

- allows you to automate the process of checking security parameters: from operating system settings to PostgreSQL configuration parameters
- creates reports on identified problems and recommendations for their elimination
- checks are performed by `.sql` and `.sh` scripts
- The results of the checks are validated by scripts in the **Lua language**, and they also generate reports and recommendations
- checks the integrity of its files with checksums
- The utility has 68 checks
- you can create your own checks by writing `.sql` `.sh` `.lua` scripts



## pg\_sec\_check utility

Postgres Security Check is a utility written in Rust and designed to audit the security of PostgreSQL database configurations. The utility allows you to automate the process of checking security parameters, from operating system settings to PostgreSQL configuration parameters. Based on the results of the checks, it creates reports on the problems identified and recommendations for their elimination.

Possibility to bind checks to PostgreSQL versions (minimum and maximum supported versions). Generates reports in HTML, JSON formats in Russian and English. Monitors the integrity of its files using checksums.

Checks are executed by: `.sql` and `.sh` scripts

The results of the checks are validated by scripts in the Lua language, and they also generate reports and recommendations.

The utility configuration file is also text, in `.json` format.

The utility has 68 checks that allow you to identify typical errors. The checks are described in the form of scripts that can be edited.

Using the example of the scripts supplied with the utility, you can create your own checks; for new checks, you need to write `.sql` `.sh` `.lua` scripts

[https://docs.tantorlabs.ru/tdb/en/17\\_5/se/pg\\_sec\\_check.html](https://docs.tantorlabs.ru/tdb/en/17_5/se/pg_sec_check.html)

# WAL-G (Write-Ahead Log Guard) utility

- utility for creating encrypted backup copies of a database cluster (full and incremental) and archiving WAL segments, their highly efficient sending/receiving via the S3 protocol "from" and "to" storage (cloud storage in the enterprise network or external) directly without creating intermediate files in the file system.
- Using the WAL-G utility you can:
  - › create backup copies of the cluster and WAL segments
  - › restore a cluster to a selected point in time in the past
  - › Manage S3 backups: delete unnecessary backups and associated log files



## WAL-G (Write-Ahead Log Guard) utility

WAL-G (Write-Ahead Log Guard) is a command line utility for creating encrypted backups of a database cluster and archiving WAL files, their efficient sending/receiving in several streams (with maximum speed and minimum load on the processor and memory) via the S3 protocol "from" and "to" storage (cloud in the enterprise network or external) directly without creating intermediate files in the host file system. WAL-G is designed to efficiently reserve the use of WAL segments, but is also capable of creating PGDATA backups of the cluster.

The utility is supplied in deb or rpm packages. The package contains a single executable file WAL-G, which is copied to the standard directory with executable files `/opt/tantor/usr/bin`.

Example of setting configuration parameters for WAL segment backup:

```
ALTER SYSTEM SET archive_command='wal-g wal-push "%p" >> ~/archive-command.log 2>&1';
```

```
ALTER SYSTEM SET restore_command='wal-g wal-fetch "%f" "%p" >> ~/restore-command.log 2>&1';
```

```
ALTER SYSTEM SET archive_mode=on;
```

Example of PGDATA backup command:

```
wal-g backup-push $PGDATA >> ~/backup-push.log 2>&1
```

Example of a command to restore from a backup (instance must be stopped):

```
wal-g backup-fetch $PGDATA LATEST
```

```
touch $PGDATA/recovery.signal
```

WAL-G can:

1) create backup copies of the cluster and WAL segments in "push" mode. The current WAL segment is not backed up and the utility **cannot be used as the only solution for ensuring high availability**.

2) restore the cluster to a selected point in time in the past. It is possible to restore WAL segments from the storage except for the current one (to which the instance processes wrote at the time the cluster was stopped). **Full recovery (without losing transactions) is possible only if the current WAL segment was not lost**

3) manage backups via S3 protocol: delete backups and associated log files

4) Encrypt files before transferring them to storage

# Other extensions

- **dbcopies\_decoding** - 1C library, provides logical replication slots when copying 1C databases
- **vector** - full support for large vector data type
- **pg\_partman** - automation of partitioned table support
- **pg\_qualstats** - maintains statistics on predicates found in WHERE clauses and in JOIN clauses
- **pg\_hint\_plan** - hints to the optimizer in queries
- **plantuner** hides indexes from the scheduler
- **pg\_cron** - intra-instance scheduler
- **pg\_throttle** - limit the amount of rows read in queries, allowing to reduce contention for input/output
- **pg\_trace** - traces of running queries, can be useful for analyzing queries in 1C
- **pg\_ddl\_deploy** - extension for logical replication
- **pgq** - queue in database



## Other extensions

For reference, here is a short description of the extensions that were not considered:

**dbcopies\_decoding** - 1C library, provides logical replication slots when copying 1C databases

**vector** - full support for the high-dimensional vector data type: functions, operators, index support.

Freely distributed project <https://github.com/pgvector/pgvector>

**pg\_partman** - automation of partitioned table support [https://github.com/pgpartman/pg\\_partman](https://github.com/pgpartman/pg_partman)

**pg\_qualstats** - keeps statistics on predicates found in WHERE statements and JOIN clauses

[https://github.com/powa-team/pg\\_qualstats](https://github.com/powa-team/pg_qualstats)

**pg\_hint\_plan** - hints to the optimizer in queries [https://github.com/ossc-db/pg\\_hint\\_plan](https://github.com/ossc-db/pg_hint_plan)

**plant** hides indexes from the scheduler <https://github.com/postgrespro/plantuner>

**pg\_cron** - intra-instance scheduler

**pg\_throttle** - limit the amount of rows read in queries, allowing to reduce contention for input/output

**pg\_trace** - tracing of running SQL queries. To obtain tracing, you need a client that will connect to the background process port and receive debug information in json format. An example of use for analyzing queries in 1C and an example of a client

<https://habr.com/ru/companies/tantor/articles/915256/>

**pg\_ddl\_deploy** - extension for logical replication , implementing the capture of DDL commands by triggers and replication of DDL commands [https://github.com/enova/pgl\\_ddl\\_deploy](https://github.com/enova/pgl_ddl_deploy)

**pgq** - a database queue from Skype. Message handlers (consumers) can be written in python and java <https://github.com/pgq/pgq>

# Practice

1. orafce extension
2. pg\_variables extension
3. page\_repair extension
  1. Preparing a replica
  2. Preparing the table
  3. Restoring a page with page\_repair
  4. Page Reset
4. Debugging subroutines
  1. Installing an extension from source code using pldebugger as an example
  2. Debugging a function in pgAdmin
  3. Debugging Subroutines in DBeaver
5. Handling Large Strings with StringBuffer
6. Finding orphaned files



## Practice

- 1.orafce extension
- 2.pg\_variables extension
- 3.page\_repair extension
- 4.Debugging subroutines
  - 1.Installing an extension from source code using pldebugger as an example
- 5.Handling Large Strings with StringBuffer
- 6.Finding orphaned files

The practical assignments for this chapter are optional and can be completed if time remains.