ADPG

Arenadata Postgres

Arenadata Postgres (ADPG) is a commercial distribution of the PostgreSQL database with additional Enterprise functionality, enterprise support and documentation.

Version CURRENT Language: EN

Contents

1. Concepts
   - Use cases
   - ADPG architecture
   - Load balancing
   - Monitoring
   - Connection pooling

2. Planning guide
   - Hardware requirements
   - Network requirements
   - Software requirements

3. Get started
   - Installation
     - Online installation
       - Install ADCM
       - Prepare hosts
       - Install ADPG cluster
         - Create a cluster
         - Add services
         - Add hosts to a cluster
         - Add components
         - Configure services
         - Configure a cluster
         - Install a cluster
       - Install monitoring
     - Offline installation
       - Install ADCM
       - Prepare hosts
       - Install Enterprise Tools cluster
         - Create a cluster
         - Add services
         - Add hosts to a cluster
         - Add components
         - Install a cluster
       - Install ADPG cluster
         - Create a cluster
         - Add services
         - Add hosts to a cluster
         - Add components
         - Configure services
         - Configure a cluster
         - Import ET settings
         - Install a cluster
       - Install monitoring
   - Connect to ADPG
     - psql
     - Third-party client tools
       - pgAdmin
       - DBeaver

4. Access management
   - Authentication
     - PG_HBA configuration
     - Password authentication
     - Kerberos
     - LDAP
     - Enable SSL encryption
   - Database roles and privileges
   - Column and row level security
     - Row security policies
     - Column-level security

5. Database maintenance
   - Cluster management via ADCM
     - Cluster actions
       - Cluster upgrade
     - Service actions
       - ADPG
       - ADPG Control Agents
       - Balancer
       - Chrony
       - Etcd
       - Monitoring
   - Monitoring metrics
   - Configure HA
     - Overview
     - Logical replication
     - Configure load balancing
     - Configure PgBouncer
     - Switchover from leader to replica
   - Backup and restore
     - SQL dump
     - File system level backup
     - PITR
     - Backup and restore using pgBackRest
   - Data optimization
     - Vacuuming
     - Obtain statistics via ANALYZE
   - Logging
   - Performance tuning
   - Enable huge pages in Linux

6. Data management
   - Data directory structure
   - Overview of database objects
     - Databases
     - Schemas
     - Tables
       - Tables overview
       - Inheritance
       - Partitioning
       - Foreign tables
       - TOAST
     - Tablespaces
     - Views and materialized views
     - Sequences
     - Indexes
     - Triggers
     - User-defined functions and procedures
   - Concurrency control
     - Transaction isolation levels
     - Locks
   - Data querying
     - Example of working with tables via psql
     - Use tables and column aliases
     - Query types
       - JOIN
       - Subqueries
       - Combine queries
       - Common table expressions
       - Aggregate functions
       - Window functions
     - Use complex data types
       - JSON/JSONB
       - JSONPath
       - XML
       - hstore
       - bytea
     - Full text search
     - Analyze queries
   - Work with extensions
     - timescaledb
   - Server programming
     - Use PL/pgSQL
     - Use PL/Python
     - Use C language functions
     - Use PL/Perl
     - Use PL/Tcl
   - Encryption using pgcrypto

7. References
   - Configuration parameters
   - Supported extensions

8. Release notes
   - ADPG releases
   - Known issues

Use cases

Eugenia Kuzina

To Table of Contents

ADPG is a powerful relational database management system (RDBMS). ADPG is based on the object-relational Open Source DBMS PostgreSQL. ADPG can be used in business critical and non-critical applications in various industries.

ADPG is a good solution in the following cases:

- **Financial industry**. ADPG is fully ACID compliant and suitable for OLTP (Online Transaction Processing) workloads. It is also an analytical database and can be integrated with math software.
- **Industrial manufactures**. ADPG is a reliable long-term data warehouse that offers reliable storage at a low cost. ADPG is the good choice because it can be configured for automatic failover, full redundancy, and almost zero downtime upgrades.
- **GIS data**. ADPG includes a powerful GIS extension — PostGIS, which provides hundreds of functions to process geographic data in various formats.
- **Web technologies and NoSQL workloads**. Websites might require thousands or hundreds of thousands of requests per second to serve customers. ADPG meets this requirement and can be used as a storage backend in scalable applications. ADPG works with modern web frameworks including Django (Python), node.js (JavaScript), Hibernate (Java), Ruby on rails, PHP, and others.
  Due to replication capabilities, websites can be scaled out to as many database servers as you need.

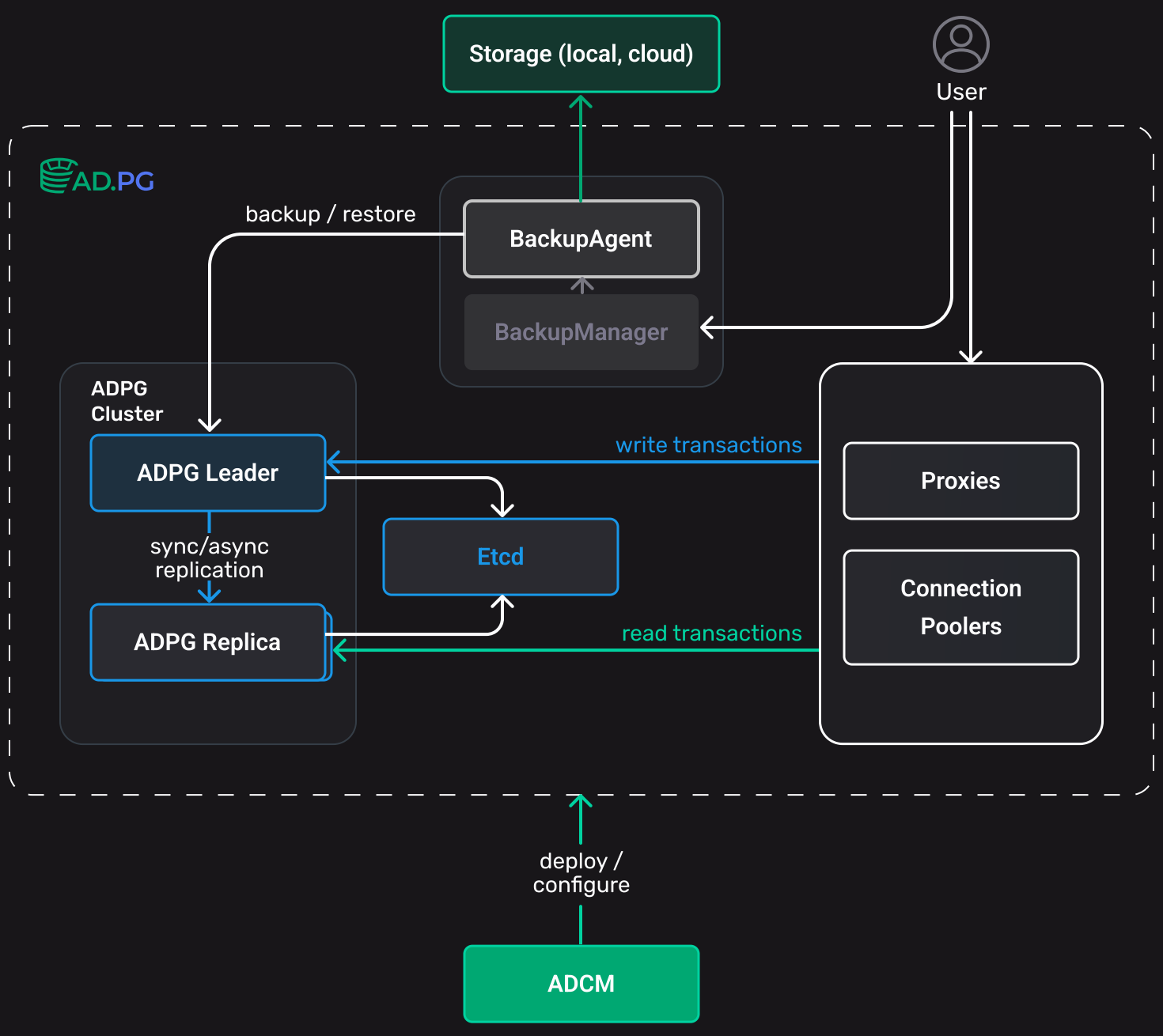
  ADPG can also be used as a NoSQL-style data store.
- **Scientific data**. Research and scientific projects can generate terabytes of data. ADPG has analytical capabilities and offers a powerful SQL engine that can easily process large amounts of data. ADPG can also be extended. You can write aggregation functions and implement your own business logic in the database.
- **Metric storage**. You can install the TimescaleDB extension and use ADPG to store time series data. For example, currency rates, vehicle movement telemetry, statistics of server access or CPU load that change over time.
- **Storage for Arenadata products**. Some Arenadata products use ADPG as their internal storage. For example, ADH saves meta-information in an ADPG cluster.

ADPG architecture

Eugenia Kuzina

To Table of Contents

Arenadata Postgres (ADPG) is a DBMS for efficient work with loads of different profiles (primarily OLTP — Online Transaction Processing). ADPG allows you to work with various amounts of data and supports a wide range of data types (including JSON and custom types) with the ability to create data processing mechanisms.



ADPG architecture

The main components of the ADPG cluster in the diagram:

- **BackupAgent** — creates a backup and sends it to the repository. BackupAgent is based on the pgBackRest utility.
- **BackupManager** — manages the creation and storage of backups.
- **ADPG leader** — the main host that has the right to perform write transactions.
- **ADPG replica** — additional hosts that are replicas of the leading ADPG host.
- **etcd** — distributed configuration storage.
- **Proxies** — distribute read/write transactions between leader host and replica hosts (HAProxy).
- **Connection Poolers** — the PgBouncer instances installed on each ADPG node that manage the number of connections and reuse them.

ADPG uses Patroni to build the cluster based on the PostgreSQL streaming replication and implement High Availability (HA). An ADPG cluster that supports HA should contain several ADPG nodes. Each node consists of a PostgreSQL server and a Patroni agent that serves the local PostgreSQL instance. One ADPG node is the leader, others are replica instances. The leader serves read-write transactions (unless otherwise specified in load balancing settings), and replicas process only read-only requests. In case of the leader failure, Patroni elects a new leader from replicas and the ADPG cluster continues to operate.

Patroni saves the cluster configuration in the Distributed Configuration Storage (DCS). ADPG requires an etcd cluster to use it as Patroni DCS.

To implement load balancing, ADPG uses HAProxy (High Availability Proxy). HAProxy is a software TCP/HTTP load balancer. It listens on ports in pairs: connections to one port from the pair are transferred to the leader (directly or throw PgBouncer), requests to another port are distributed across ADPG nodes (directly or throw PgBouncer). For more information, see Load balancing.

PgBouncer is used as a connection pooler. An application can connect to PgBouncer as an ADPG server, and PgBouncer creates a connection to the ADPG server, or reuses one of its existing connections. PgBouncer is designed to reduce the performance impact of opening new connections. To avoid compromising transaction semantics, PgBouncer supports multiple types of pooling when rotating connections:

- Session pooling — when a client connects, a server connection is assigned to it for the entire time the client is connected. When the client disconnects, the server connection returns to the pool. This is the default method.
- Transaction pooling — a server connection is assigned to a client only during a transaction. When PgBouncer notices that the transaction is complete, the server connection is put back into the pool.
- Statement pooling — the server connection is returned to the pool immediately after a query completes. Multi-statement transactions are prohibited in this mode, because they may be disrupted.

Another ADPG part is **Storage** that is a backup storage host. You can use any type of storage supported by the pgBackRest utility.

ADPG cluster management is performed via ADCM. The following actions are available in the ADCM UI:

- Cluster actions that allow you to manage the entire cluster with all services.
- Service actions that manage a single service.

Load balancing

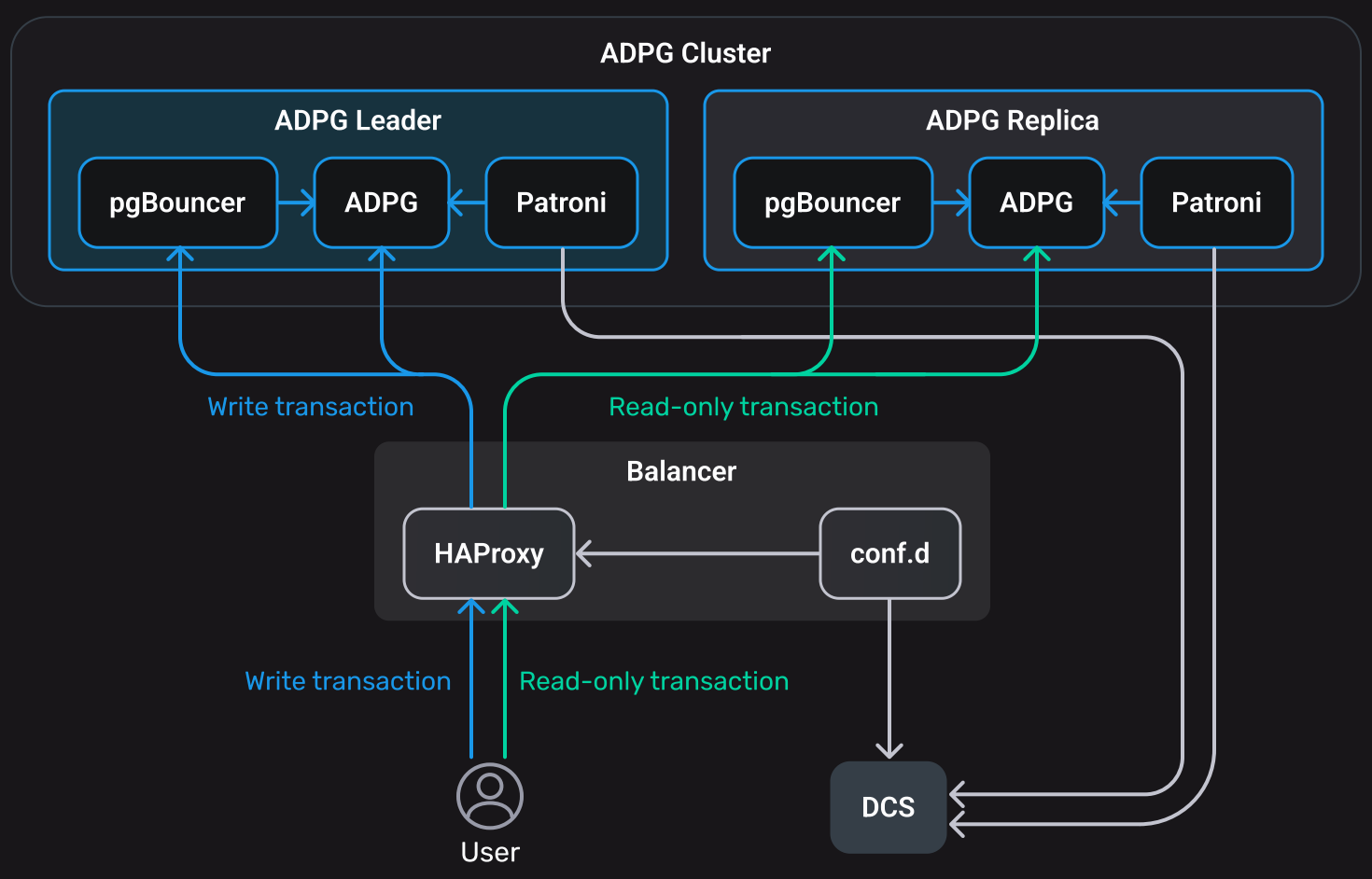
Eugenia Kuzina

To Table of Contents

NOTE

The load balancing feature is available for ADPG Enterprise Edition.

The ADPG cluster implements load balancing, which distributes client requests across multiple database replicas. This technique allows ADPG to process more requests simultaneously.



Load balancing in ADPG

ADPG uses HAProxy (High Availability Proxy) to implement load balancing. HAProxy is a software TCP/HTTP load balancer. It listens on two ports: connections to one of them are transferred to the leader, requests to another are distributed across ADPG nodes. Write transactions should be sent to the first port, read only transactions to the second one. If you send a write transaction to the port for read only requests, ADPG refuses it.

NOTE

HAProxy does not check whether a current request is a write or read-only transaction. Client applications must choose the correct port for connections.

HAProxy chooses which node should process a client request according to specified rules. Only the leader can process write transactions. Read only transactions are passed to replicas. You can also configure HAProxy to allow the leader to process read only transactions. See Configure load balancing for details.

If PgBouncer is enabled, write transactions will be sent to a PgBouncer instance of an ADPG leader node and read only transactions — to PgBouncer instances of replica nodes. See Configure PgBouncer.

The ADPG uses the Etcd cluster as DCS (Distributed Configuration Store). The Patroni service that implements streaming replication stores information about ADPG cluster configuration in etcd. For more information about ADPG replication, see ADPG high availability overview.

The conf.d service updates HAProxy configuration files when the cluster configuration changes. For example, a new node is added, or a leader is changed. The conf.d service is installed on the host with HAProxy. It subscribes on ADPG cluster configuration updates stored in Etcd. When ADPG changes the configuration, the conf.d service generates new configuration files for HAProxy and reloads HAProxy to apply the changes.

HAProxy also uses the Patroni REST API for additional verification which node is the leader and which is the replica.

TIP

To improve HAProxy high availability, you can use technologies that are not related to ADPG. For example, HAProxy instances may be added as A/AAAA records with a single FQDN to DNS server. Also, the keepalived tool can be used to manage multiple HAProxy hosts.

Monitoring

Eugenia Kuzina

Contents

To Table of Contents

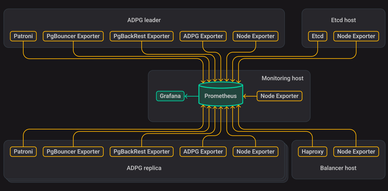
Overview

Monitoring service settings

Overview

The Monitoring service implements monitoring in the ADPG cluster. It deploys the Prometheus server, which collects and stores metrics. The Monitoring service also supports Grafana for data visualization.

Prometheus retrieves metrics through HTTP calls to specific endpoints. Metrics are transferred to these endpoints by exporters, which should be installed on the cluster hosts.



Monitoring in ADPG

The table below describes the components of the Monitoring service and on which hosts they should be installed.

Component	Description	Host
Prometheus	The Prometheus server that collects and stores metrics	A separate host for monitoring
Grafana	The Grafana web interface for creating dashboards to visualize metrics	A host for Grafana. It is recommended to use a host with a Prometheus server
ADPG Exporter	Sends PostgreSQL metrics	All hosts with the ADPG service
Pgbouncer Exporter	Sends Pgbouncer metrics	All hosts with the ADPG service
PgBackRest Exporter	Sends backup statistics	All hosts with the ADPG service
Node Exporter	Sends system metrics	All hosts from which system metrics need to be collected

Patroni and Etcd exposes the `/metrics` endpoint by default. There is no need to configure a separate exporter for them to support Prometheus scraping. The Balancer (HAProxy) service does not require installation of a separate component to export metrics, but you can configure the parameters of the built-in HAProxy exporter in the Monitoring service settings.

Monitoring service settings

To specify Monitoring service parameters, open the ***Services*** tab of the ADPG cluster and select the ***Monitoring*** service. The parameters are available on the ***Primary configuration*** tab.

HAProxy Exporter settings

Parameter	Description	Default value
Listen port	Port on the host with the Balancer service to retrieve HAProxy metrics in the Prometheus format	8405
Metrics endpoint	Endpoint on the host with the Balancer service to retrieve HAProxy metrics in the Prometheus format	/metrics

Pgbouncer Exporter settings

Parameter	Description	Default value
Listen port	Port on the host with the ADPG service to retrieve connection pooler metrics in the Prometheus format	9188

PgBackRest Exporter settings

Parameter	Description	Default value
Listen port	Port on the host with the ADPG service to retrieve PgBackRest metrics in the Prometheus format	9854
Metrics endpoint	Endpoint on the host with the ADPG service to retrieve PgBackRest metrics in the Prometheus format	/metrics

ADPG Exporter settings

Parameter	Description	Default value
Listen port	Port on the host with the ADPG service to retrieve PostgreSQL metrics in the Prometheus format	9187

Prometheus settings

Group	Parameter	Description	Default value
—	scrape_interval	Specifies how frequently to scrape targets	1m
—	scrape_timeout	Specifies how long to wait until a scrape request times out	10s
—	Password for Grafana connection	A password of a Grafana user (`adsprom_grafana`) that is used to connect to Prometheus. The password should contain a combination of uppercase [A-Z] and lowercase [a-z] English letters, digits [0-9], special characters and have a minimum length of 8	—
—	Prometheus users to login/logout to Prometheus	User credentials for logging into the Prometheus web interface	—
Service parameters	config.file	Path to the Prometheus server configuration file	/etc/admprom/prometheus/prometheus.yml
	storage.tsdb.path	Path to the Prometheus server database	/var/lib/admprom/prometheus
	web.console.libraries	Path to console management libraries	/usr/share/admprom/prometheus/console_libraries
	web.console.templates	Path to Prometheus server console templates	/usr/share/admprom/prometheus/consoles
	web.config.file	Specifies which web configuration file to load. The file should have the YAML format	/etc/admprom/prometheus/prometheus-auth.yml
	storage.tsdb.retention.time	Specifies how long to retain samples in the storage. Supported units: `y`, `w`, `d`, `h`, `m`, `s`, `ms`	15d
	web.listen-address	Address to access the Prometheus web interface	0.0.0.0:11200

Grafana settings

Parameter	Description	Default value
Grafana administrator's password	Password of the Grafana `admin` user. The password should include uppercase [A-Z] and lowercase [a-z] English letters, digits [0-9], and special characters	—
Grafana listen port	Port to access the Grafana web interface	11210

Node Exporter settings

Parameter	Description	Default value
Listen port	Port on the cluster's host to access host system metrics in the Prometheus format	11203
Metrics endpoint	Endpoint on the cluster's host to retrieve system metrics in the Prometheus format	/metrics

The ***Prometheus settings → Password for Grafana connection*** and ***Grafana settings → Grafana administrator's password*** parameters are mandatory and should be specified during the monitoring installation.

For more information on the Monitoring service installation, refer to Install monitoring.

The Monitoring service metrics are described in ADPG monitoring metrics.

Connection pooling

Eugenia Kuzina

Contents

To Table of Contents

How ADPG/PostgreSQL establishes connections

How PgBouncer works

PgBouncer modes

- Session
- Transaction
- Statement

PostgreSQL has a resource-intensive connection processing architecture. Every incoming connection creates a new backend process. This is not efficient when dealing with large numbers of connections in a production environment. When a client sends a request to a PostgreSQL server, it creates a connection with a database. Until the server responds back to the client, the connection remains open. The memory is allocated for each of these connections. If connections are opened and closed at a high frequency, this can cause problems. This is why PgBouncer is needed. PgBouncer is a PostgreSQL connection pooler that allows you to increase the number of connections without performance degradation.

NOTE

The PgBouncer feature is available for ADPG Enterprise Edition. See Configure PgBouncer.

How ADPG/PostgreSQL establishes connections

ADPG/PostgreSQL implements a "process per user" client/server model. In this model, every client process connects to exactly one backend process. Since it is not determined how many connections will be made, the PostgreSQL server has to use a supervisor process that spawns a new backend process every time a connection is requested. This supervisor process is called **postmaster** and listens at a specified TCP/IP port for incoming connections. Whenever it detects a request for a connection, it spawns a new backend process. Those backend processes communicate with each other and with other processes of the server instance using semaphores and shared memory to ensure data integrity throughout concurrent data access. Each of these backend processes may grow depending on the data it is accessing.

Despite the fact that memory is allocated for each new backend process, the available memory is not the main limiting factor. A more subtle and important point is that postmaster and its internal processes share memory for communication, and some parts of this shared space are global bottlenecks. For example, here is a structure that keeps track of every current process and transaction. Operations that happen in any backend require walking the entire list of processes or transactions stored in this structure. An exclusive lock is used to add a new process to the structure. The cumulative effect is that the performance of any given backend is inversely proportional to the number of all active backends in the system.

The `max_connections` ADPG/PostgreSQL configuration parameter determines the maximum number of concurrent connections to the server. The default value is `100`. It is highly likely that this number of connections may not be enough in a production environment.

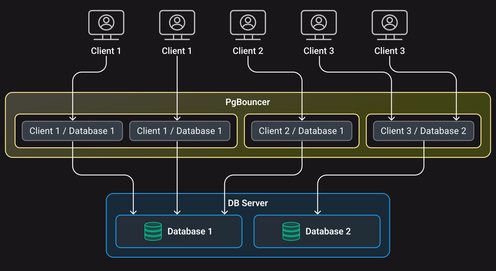
How PgBouncer works

PgBouncer is a PostgreSQL connection pooler. The client application can connect to PgBouncer as an ADPG/PostgreSQL server. The important point is PgBouncer caches the connections.

When PgBouncer receives a client connection, it performs authentication on behalf of the PostgreSQL server. PgBouncer supports all the authentication mechanisms that PostgreSQL provides.

If authentication is successful, PgBouncer checks for a cached connection with the same username and database combination. If a cached connection is found, it returns the connection to the client. Otherwise, PgBouncer creates a new connection. This operation must not violate the restrictions set in the following parameters: `default_pool_size`, `max_client_conn`, `max_db_connections`, and `max_user_connections`. All of these values can be defined in the PgBouncer settings. If creating a new connection would violate any limits specified in these settings, PgBouncer queues the connection until a new one can be created, unless the `max_client_conn` is exceeded. Violating the `max_client_conn` constraint aborts the connection.

In the `transaction` or `statement` pooling mode, the post-authentication operations are executed only when the client starts executing a transaction or statement, respectively. The pooling modes are described below.



How PgBouncer works

Before returning a connection to the pool, PgBouncer performs a reset query to remove all session information — this makes it safe to share connections between clients.

PgBouncer can help you optimize:

- network traffic;
- overhead of starting a new backend process and authentication.

PgBouncer does not support automated load balancing or high availability, but ADPG Enterprise Edition provides this functionality. See Load balancing.

PgBouncer has the following advantages:

- It does not require client-side code changes.
- It provides pass-through authentication. PgBouncer can securely authenticate users without access to their passwords. This makes PgBouncer more secure and easier to maintain — there is no need to update PgBouncer every time a user updates the password.
- It is a single process, and all client commands and server responses pass through it without any processing. Therefore, it maintains a very small memory footprint.
- It can significantly improve the number of transactions per second a server processes, and it scales very well to many clients.

PgBouncer modes

PgBouncer modes allow users to decide when a connection should be returned to the pool. The following modes are available:

- session
- transaction
- statement

You can set the required mode on the ***Primary configuration*** tab of the ***ADPG*** service. See Configure PgBouncer.

Session

In the `session` pooling mode, a connection is returned to the pool when a client closes a session. This is the default mode. This mode reduces the time and resources required to establish connections and authenticate users.

Transaction

In the `transaction` pooling mode, a connection is returned to the pool when a client completes a transaction. There is no guarantee that two transactions run on the same client PgBouncer connection will run on the same PgBouncer server connection. A server connection is assigned to the client connection whenever client starts a new transaction, and is released to the pool not when client disconnects, but rather — when it finishes transaction, for example, using `ROLLBACK` or `COMMIT`. Note that if you do not start a transaction explicitly, each query in PostgreSQL is executed in a single transaction. See Transactions.

The transaction mode provides better performance and allows you to reduce the number of idle transactions, in addition to reducing time spent on establishing connections and user authentication.

You need to be careful when using sequences, advisory locks, and `%_SHARED` in PL/Perl functions. Also, query plans created by the PREPARE statements are cashed and exist outside a transaction. Their use in the transaction mode may lead to errors or unexpected results.

This is the full list of the session-based PostgreSQL features that are not supported in the transaction mode:

- SET / RESET;
- LISTEN;
- `WITH HOLD CURSOR` option of the DECLARE command;
- PREPARE / DEALLOCATE;
- `ON COMMIT PRESERVE ROWS` and `ON COMMIT DELETE ROWS` options of the CREATE TABLE command for temporary tables;
- LOAD;
- Session-level advisory locks.

For more information, see PgBouncer features.

Statement

In the `statement` pooling mode, a connection is returned to the pool as soon as a statement is executed. It is the most aggressive mode. This is a transaction pool with one limitation: multi-statement transactions are not allowed. This is intended to provide an autocommit mode on the client, primarily targeted at PL/Proxy.

Hardware requirements

Eugenia Kuzina

Contents

To Table of Contents

ADPG CE

ADPG EE

Arenadata Enterprise Tools

Arenadata Monitoring

IMPORTANT

The following system requirements are **minimal**. The target sizing should be calculated based on the customer requirements.

ADPG CE

ADPG service (PostgreSQL instance)

- CPU: 2 cores
- RAM: 4 GB
- Disk storage: 30 GB

Monitoring service

- CPU: 2 cores
- RAM: 2 GB
- Disk storage: 20 GB

ADPG EE

ADPG service (PostgreSQL with Patroni and PgBouncer)

- CPU: 4 cores
- RAM: 8 GB
- Disk storage: 30 GB

Monitoring service

- CPU: 2 cores
- RAM: 2 GB
- Disk storage: 20 GB

Balancer service

- CPU: 1 core
- RAM: 1 GB
- Disk storage: 5 GB

Etcd service

- CPU: 2 cores
- RAM: 2 GB
- Disk storage: 10 GB

Arenadata Enterprise Tools

If you use ET along with your product, take into account the following requirements:

- Virtual server.
- CPU: 4 cores or more.
- RAM: 8 GB or more.
- Disk: 50 GB or more.

Arenadata Monitoring

If you use ADM along with your product, take into account the following requirements:

- Virtual server.
- CPU: 8 cores or more.
- RAM: 16 GB or more.
- Disk: 10 GB per monitored host or more.

Network requirements

Eugenia Kuzina

To Table of Contents

- One or more dedicated and separate peer-to-peer Interconnect networks are required for internal cluster communication.
- All cluster servers should be connected to Interconnect networks.
- All servers should have fully qualified domain names (FQDN) and the ability to communicate with each other at required speeds.
- The IP address of each server should be known.
- Ping access to any server (ICMP) should be provided.
- For online ADPG installation, access to Arenadata Cluster Manager deployed on a dedicated server is required. See the Install ADCM article for details on how to install Arenadata Cluster Manager.
- For offline ADPG installation, access to Arenadata Enterprise Tools deployed on a dedicated server is required. The Install Enterprise Tools cluster section contains instructions on how to install Arenadata Enterprise Tools.
- ADPG hosts should resolve short host names to IP addresses to connect to each other. If your network does not include a DNS service to resolve host names, you can add FQDN and a short name for each IP address to the hosts file on all ADPG hosts.
- The ports listed below should be open on hosts where you install the specified service.

ADPG

Service	Port	Purpose
Common	11203	Port on cluster hosts to access system metrics in the Prometheus format
ADPG	5432	The default port that the ADPG service (PostgreSQL) listens on
	15432	The default port that PgBouncer listens on
	8008	Patroni REST API port — the port on a host with the ADPG service to retrieve Patroni metrics in the Prometheus format
	9187	Port on a host with the ADPG service to retrieve PostgreSQL metrics in the Prometheus format
	9188	Port on a host with the ADPG service to retrieve connection pooler (PgBouncer) metrics in the Prometheus format
	9854	Port on a host with the ADPG service to retrieve PgBackRest metrics in the Prometheus format
Balancer	6432	Port on a host with the HAProxy component for write transactions
	6433	Port on a host with the HAProxy component for read only transactions
	7000	Port on which a web page with a HAProxy statistics report is available
	16432	Port, all connections to which are transferred to a PgBouncer instance of an ADPG leader node
	16433	Port, all connections to which are transferred to PgBouncer instances of ADPG replica nodes
	8405	Port on a host with the Balancer service to retrieve HAProxy metrics in the Prometheus format
Etcd	2380	Port for peer communication
	2379	Port for client requests — port on a host with the Etcd service to retrieve PgBackRest metrics in the Prometheus format
Monitoring	11200	Port to access the Prometheus web interface
	11210	Port to access the Grafana web interface

Enterprise Tools

Port	Purpose
443	TCP port used by Arenadata Enterprise Tools (Docker Registry service)
81	TCP port used by Arenadata Enterprise Tools (access to the official repositories for offline installation)

ADCM

Port	Purpose
8000	TCP port to send statuses of cluster components to ADCM

Software requirements

Eugenia Kuzina

Contents

To Table of Contents

Operating system

ADCM

SELinux

Antivirus

Default network route

This page provides information about requirements and recommendations before ADPG installation.

Operating system

IMPORTANT

- **CentOS 7.9**. Since the life cycle of CentOS 7 ended on June 30, 2024, it is strongly recommended to use Ubuntu as a free open-source software. In ADPG, CentOS 7 support is retained for compatibility with the ADPG service used in other Arenadata products and will be deprecated in releases after January 1, 2026. ADPG v16.3.5 is the last release that supports CentOS 7, and ADPG v16.3.5 on CentOS 7 is not compatible with ADPG Control because ADPG Control cannot be installed on CentOS 7.
- **Ubuntu 22.04** will be replaced with Ubuntu 24.04 in releases after April 1, 2026. Consider ADPG cluster backup/restore along with an OS upgrade, if planned.
- **Alt Linux 8.4 SP** will be replaced with Alt Linux 10 SP in releases after January 1, 2026. Consider ADPG cluster backup/restore along with an OS upgrade, if planned.

ADPG supports only x86_64 architecture and the following operating systems:

- CentOS 7.9 (x86_64)
- RHEL 7.9 (x86_64)
- Ubuntu 22.04 (x86_64)
- Alt Linux 8.4 SP (x86_64) in the ADPG Enterprise Edition
- Astra Linux 1.7 SE "Orel", "Voronezh" (x86_64) in the ADPG Enterprise Edition
- RED OS 7.3 Certified edition in the ADPG Enterprise Edition

You need to prepare all dependent repositories together with the chosen operating system.

ADCM

It is required to use ADCM 2.6.0 or higher. When trying to load an ADPG bundle into ADCM of an unsupported version, the following error occurs: `Map key "flag_autogeneration" is not allowed here (rule "cluster_object")`.

SELinux

SELinux must be turned off. The full guide about this operation you can find in the Install ADCM section.

Antivirus

It is recommended to switch off any antivirus software before ADPG installation and during the ADPG operation process. In case of using antivirus, please contact your antivirus software vendor to find out the settings required to use ADPG.

Default network route

Make sure that routing tables on each host contain the default route — `0.0.0.0/0`. For more information, see Prepare hosts.

Installation

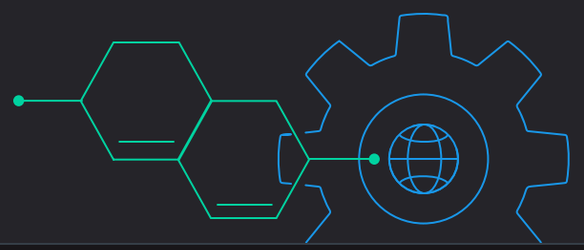
To Table of Contents

This tutorial guides you through the process of installing an Arenadata Postgres (ADPG) cluster. The easiest way to install it is to use Arenadata Cluster Manager (ADCM) — a separate Arenadata product designed for simple, convenient, and fast software deployment and exploitation.

You can install ADPG via ADCM in two ways.

Online installation

on hosts with access to Internet



Offline installation

on hosts without access to Internet

Online installation

To Table of Contents

This topic describes **online** installation of ADPG. It can be applied to the hosts with access to Internet.

The process of online installation via ADCM includes the following steps.

AD.CM

Step 1.

Install ADCM

AD.CM

Step 2.

Prepare hosts

AD.CM

ARENADATA POSTGRES

Step 3.

Install ADPG cluster

AD.CM

Step 4.

Install monitoring

AD.CM

ARENADATA ADPG CONTROL

Step 5.

Install ADPG Control

Install AGCM

Contents

- [Step 1. Check prerequisites](#)
  - [Software and hardware requirements](#)
  - [Check the hardware requirements](#)
  - [Check the software requirements](#)
  - [Check the DNS resolution](#)
  - [Check the NTP synchronization](#)
- [Step 2. Install AGCM](#)
  - [Check the Docker customer status](#)
  - [Check the AGCM user availability](#)
  - [Check the AGCM web interface](#)
  - [Set the AGCM URL](#)
- [Step 3. Update AGCM](#)

Step 1. Check prerequisites

Software and hardware requirements

Check the software and hardware requirements for your AGCM deployment.

- Linux x64
- Docker
- Git
- Curl
- Wget
- Nano
- Net-tools
- Iproute2
- Systemd-resolved

Check the software requirements for your AGCM deployment.

Check the hardware requirements

Check the hardware requirements for your AGCM deployment.

- 4 CPU
- 8 GB RAM
- 100 GB HDD

Check the software requirements

Check the software requirements for your AGCM deployment.

Install Docker, Git, Curl, Wget, Nano, Net-tools, Iproute2, Systemd-resolved, Docker Compose, Docker Swarm, and Linux kernel 5.x.

```
sudo apt-get update
sudo apt-get install -y docker.io git curl wget nano net-tools iproute2 systemd-resolved
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Install Docker, Git, Curl, Wget, Nano, Net-tools, Iproute2, Systemd-resolved, Docker Compose, Docker Swarm, and Linux kernel 5.x.

```
sudo yum update -y
sudo yum install -y docker git curl wget nano net-tools iproute2 systemd-resolved
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Check the DNS resolution

Check the DNS resolution for your AGCM deployment.

Install Systemd-resolved, Net-tools, and Iproute2 packages. Configure the DNS resolution. Restart the systemd-resolved service.

```
sudo apt-get update
sudo apt-get install -y systemd-resolved net-tools iproute2
sudo nano /etc/systemd/resolved.conf
DNS=8.8.8.8 8.8.4.4
FallbackDNS=1.1.1.1 1.0.0.1
sudo systemctl restart systemd-resolved
```

Install Systemd-resolved, Net-tools, and Iproute2 packages. Configure the DNS resolution. Restart the systemd-resolved service.

```
sudo yum update -y
sudo yum install -y systemd-resolved net-tools iproute2
sudo nano /etc/systemd/resolved.conf
DNS=8.8.8.8 8.8.4.4
FallbackDNS=1.1.1.1 1.0.0.1
sudo systemctl restart systemd-resolved
```

Check the NTP synchronization

Check the NTP synchronization for your AGCM deployment.

Install NTP. Configure the NTP synchronization. Restart the NTP service.

```
sudo apt-get update
sudo apt-get install -y ntp
sudo nano /etc/ntp.conf
server 0.pool.ntp.org iburst
server 1.pool.ntp.org iburst
server 2.pool.ntp.org iburst
server 3.pool.ntp.org iburst
sudo systemctl restart ntp
```

Install NTP. Configure the NTP synchronization. Restart the NTP service.

```
sudo yum update -y
sudo yum install -y ntp
sudo nano /etc/ntp.conf
server 0.pool.ntp.org iburst
server 1.pool.ntp.org iburst
server 2.pool.ntp.org iburst
server 3.pool.ntp.org iburst
sudo systemctl restart ntp
```

Disable SELinux

Disable SELinux.

To disable SELinux permanently, edit the /etc/selinux/config file and set the SELINUX parameter to disabled.

```
sudo nano /etc/selinux/config
SELINUX=disabled
```

Reboot the system.

To verify if the changes have been applied, run the following command:

```
sestatus
SELinux status:                 disabled
```

Stop Firewall

Stop Firewall.

To stop the Firewall service, run the following command:

```
sudo systemctl stop firewalld
```

To disable the Firewall service from starting at boot, run the following command:

```
sudo systemctl disable firewalld
```

To verify if the Firewall service is stopped, run the following command:

```
sudo systemctl status firewalld
● firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/lib/systemd/system/firewalld.service; disabled; vendor preset: enabled)
   Active: inactive (dead) since Mon 2021-08-09 10:00:00 UTC; 1min 0s ago
     Docs: man:firewalld(1)
  Process: 1234 ExecStart=/usr/sbin/firewalld --nofork --nopid $FIREWALLD_ARGS (code=exited, status=0/SUCCESS)
 Main PID: 1234 (code=exited, status=0/SUCCESS)
```

Allow access to the Docker CLI without root privileges

To allow access to the Docker CLI without root privileges, create the docker group and add your user to it.

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Log out and log back in so that your group membership is re-evaluated.

Step 2. Install AGCM

Check the Docker customer status

Check the Docker customer status.

To check the Docker customer status, run the following command:

```
sudo docker customer status
```

The output should look like this:

```
CUSTOMER_ID   NAME          EMAIL                 COMPANY       STATUS    PLAN
1234567890    John Doe      john.doe@example.com  Example Inc.  active    Business
```

Check the AGCM user availability

Check the AGCM user availability.

To check the AGCM user availability, run the following command:

```
sudo cat /etc/passwd | grep agcm
```

The output should look like this:

```
agcm:x:1001:1001::/home/agcm:/bin/bash
```

Check the AGCM web interface

To check the AGCM web interface, open https://<agcm-host>/ in a browser.

- Host: agcm
- Port: 443
- Protocol: https

Check the AGCM web interface

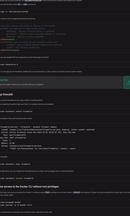
Check the AGCM web interface

Check the AGCM web interface

Set the AGCM URL

Set the AGCM URL.

1. In the Settings menu, click Server.
2. In the Server URL field, enter the AGCM URL.



Set the AGCM URL

3. Click Save.

Set the AGCM URL

Step 3. Update AGCM

Update AGCM.

To update AGCM, change your branch and then run the following commands:

```
cd /opt/agcm
git checkout <your-branch>
git pull origin <your-branch>
```

Then restart the AGCM service:

```
sudo systemctl restart agcm
```

Verify the update by checking the AGCM version:

```
sudo cat /opt/agcm/VERSION
```

Update AGCM.

To update AGCM, change your branch and then run the following commands:

```
cd /opt/agcm
git checkout <your-branch>
git pull origin <your-branch>
```

Then restart the AGCM service:

```
sudo systemctl restart agcm
```

Verify the update by checking the AGCM version:

```
sudo cat /opt/agcm/VERSION
```

Update AGCM.

To update AGCM, change your branch and then run the following commands:

```
cd /opt/agcm
git checkout <your-branch>
git pull origin <your-branch>
```

Then restart the AGCM service:

```
sudo systemctl restart agcm
```

Verify the update by checking the AGCM version:

```
sudo cat /opt/agcm/VERSION
```

Prepare hosts

To Table of Contents

To manage a cluster and its nodes via ADCM, it is necessary to prepare hosts for a cluster and link these hosts to ADCM. This is the function of ADCM **hostproviders** — special plugins for ADCM, which encapsulate the complex mechanism of interaction between ADCM and the hosts controlled by it.

ADCM supports several types of hostproviders. For more information on how to configure hostproviders and use them to create hosts, refer to the following documentation:

- SSH
- Yandex Cloud
- VCloud (VMware)
- Cloud.ru Advanced
- CROC Cloud

IMPORTANT

It is forbidden to use the reserved words `adpg`, `balancer`, `chrony`, `etcd`, and `prometheus_monitoring` as hostnames in an ADPG cluster.

Make sure that routing tables on each host contain the default route — `0.0.0.0/0`.

Use the `route` command to check this:

```
$ route
```

The output should be similar to:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         gateway         0.0.0.0         UG    100    0        0 eth0
10.92.40.0      0.0.0.0         255.255.252.0   U     100    0        0 eth0
```

If the `0.0.0.0/0` default route is missing in the output, add it using the following command:

```
$ sudo route add default gw 0.0.0.0 eth0
```

Create a cluster

Eugenia Kuzina, Daria Barysheva

Contents

To Table of Contents

Step 1. Download a cluster bundle

Step 2. Upload a cluster bundle to ADCM

Step 3. Create a cluster based on the uploaded bundle

In ADCM a **cluster** means a set of services running on a set of hosts.

Clusters distributions for ADCM are made in the form of so-called bundles. Generally a bundle is a regular archive that includes a description and software logic of cluster deployment and exploitation.

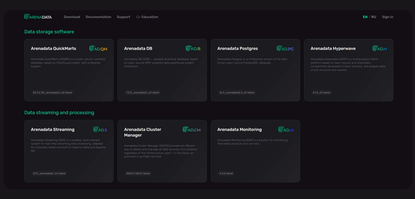
The steps for adding a cluster to ADCM are listed below.

NOTE

If you use the Enterprise Edition, skip the step 1, get a bundle from the Arenadata support team, and move to the step 2.

Step 1. Download a cluster bundle

1. Go to https://network.arenadata.io/ and choose ***Arenadata Postgres***.



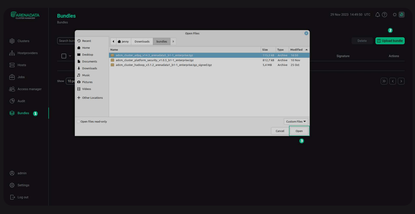
Switch to the bundle download page

2. On the next page, download a bundle for the required version of ADPG.

Step 2. Upload a cluster bundle to ADCM

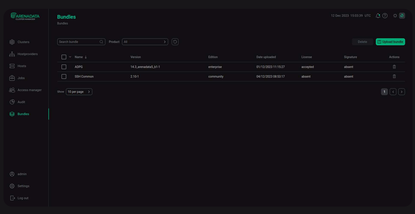
To upload a bundle to ADCM, follow the steps:

1. Select the ***Bundles*** item in the left navigation menu and click ***Upload bundle***.
2. Select a bundle in the Open File Dialog.



Upload a bundle

3. As a result of the performed actions, a bundle is displayed on the ***Bundles*** page.



The result of successful uploading a bundle

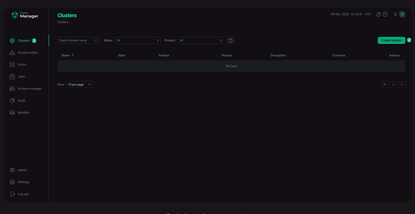
Step 3. Create a cluster based on the uploaded bundle

IMPORTANT

When creating a cluster via the ADCM web interface, you generate a new ADCM cluster instance. It means only adding the cluster to ADCM — at this stage no services are deployed.

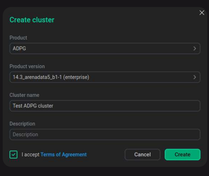
The steps for cluster creation are given below:

1. Select the ***Clusters*** item in the left navigation menu and click ***Create cluster***.



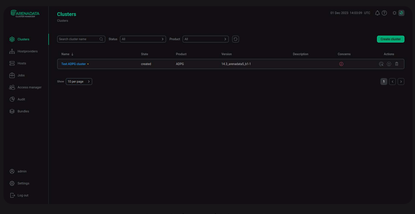
Switch to cluster creation

2. In the opened dialog, describe a new cluster instance:
   - Select an uploaded cluster bundle in the ***Product*** field.
   - Select a bundle version in the ***Product version*** field. Several versions become available in the case of different versions of the same bundle being uploaded.
   - Enter a cluster name in the ***Cluster name*** field.
   - Enter a cluster description in the ***Description*** field if necessary.
   - Set the ***I accept Terms of Agreement*** flag to sign the license agreement. This flag is displayed only for Enterprise bundles. To read the agreement text, refer to the ***Terms of Agreement*** link. You can also sign the license agreement on the ***Bundles*** page by clicking the desired product bundle.
   - Click ***Create***.



Create a cluster

3. As a result, the created cluster instance is displayed on the ***Clusters*** page.



The result of successful cluster creation

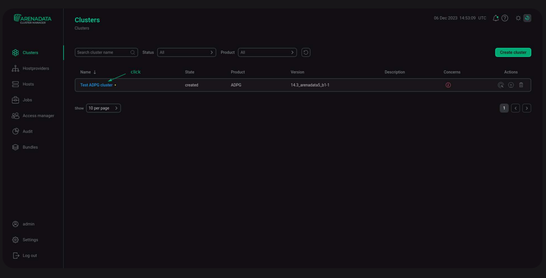
Add services

Daria Barysheva, Eugenia Kuzina

To Table of Contents

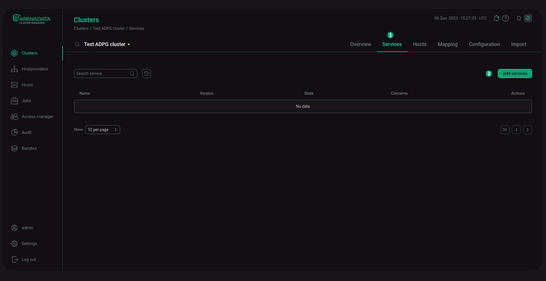
In ADCM a **service** means a software that performs some function. Examples of services for ADPG clusters: ADPG, Chrony, etc. The steps for adding services to a cluster are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



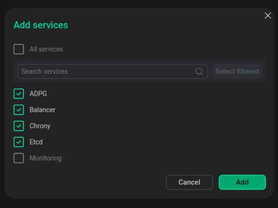
Select a cluster

2. Open the ***Services*** tab on the cluster page and click ***Add service***.



Switch to adding services

3. In the opened dialog, select services that should be added to the cluster and click ***Add***.



Select services

IMPORTANT

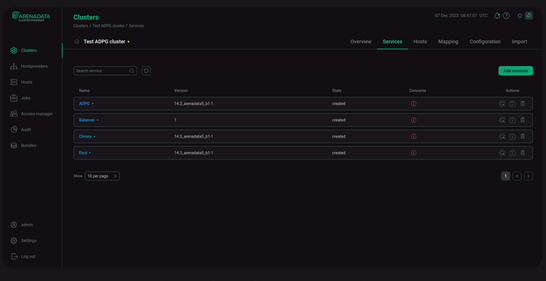
ADPG Enterprise Edition requires etcd (the built-in ADPG Etcd service or an external etcd cluster) to use it as Patroni Distributed Configuration Store (DCS). Its settings must be specified during installation, after installation they cannot be changed.

The brief description of available services is listed below. CE means the ADPG Community Edition, EE — the ADPG Enterprise Edition.

Services that can be added to the ADPG cluster

Service	Mandatory	Edition	Purpose
ADPG	Yes	CE/EE	A core service corresponding to the object-relational DBMS PostgreSQL
ADPG Control Agents	No	EE	A service that allows you to manage an ADPG cluster with ADPG Control
Balancer	No	EE	The service is based on HAProxy that implements load balancing
Chrony	No	CE/EE	A time service. Provides time synchronization for the cluster hosts, that is necessary for the cluster to work correctly. If the time service is already configured on the hosts, there is no need to install Chrony
Etcd	No	EE	The ADPG enterprise edition requires etcd to use it as the Patroni Distributed Configuration Store (DCS). You can utilize the built-in ADPG Etcd service or an external etcd cluster. If you want to use the external etcd cluster, do not add the built-in Etcd service on the ***Services*** tab. Instead, set ***External Etcd cluster*** settings on the ***Primary configuration*** tab of the ADPG service. To specify them, open ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** as described in Configure services
Monitoring	No	CE/EE	Service for collecting and storing ADPG cluster monitoring metrics based on the Prometheus monitoring system and the Grafana service

4. As a result, the added services are displayed at the *Services* tab.



The result of successful adding services to a cluster

NOTE

You can also add services later. The process of adding new services to already running cluster does not differ from installing a service from scratch.

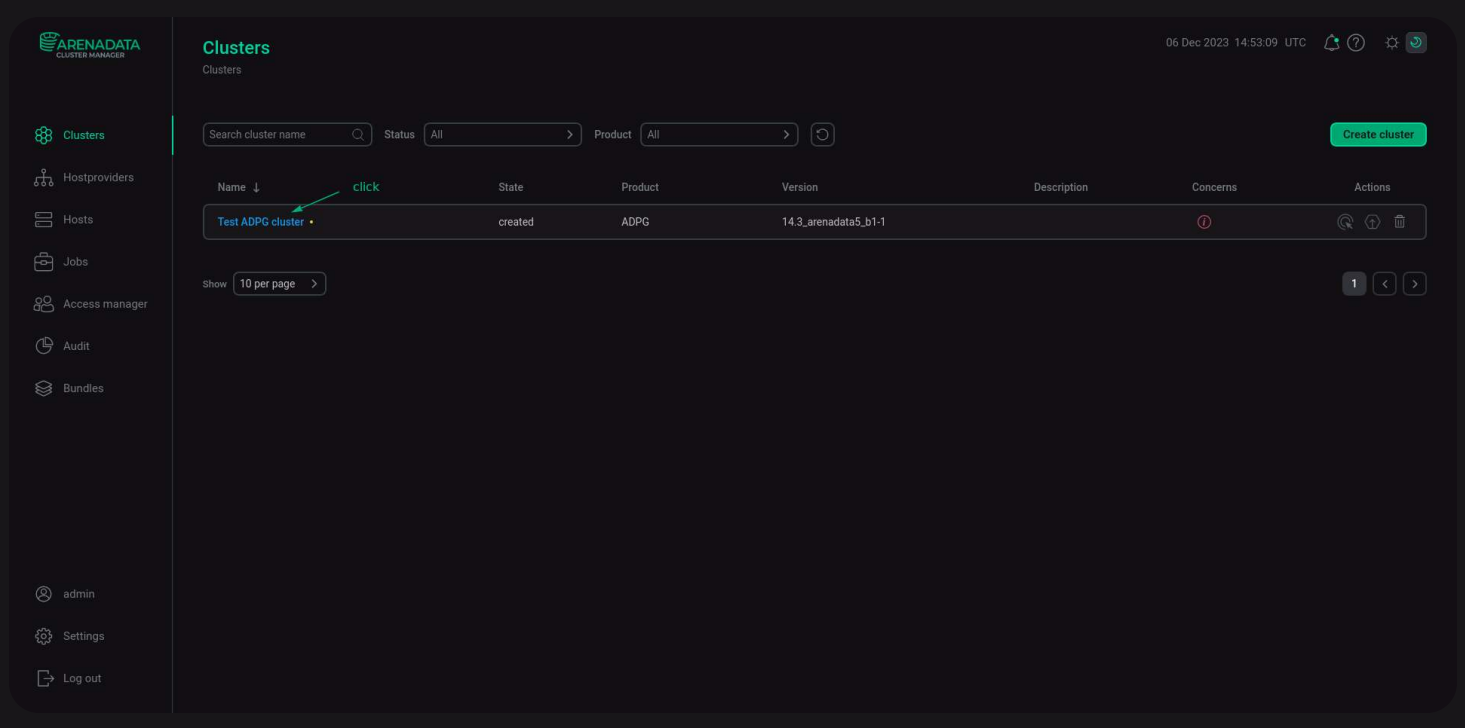
Add hosts to a cluster

Eugenia Kuzina, Daria Barysheva

To Table of Contents

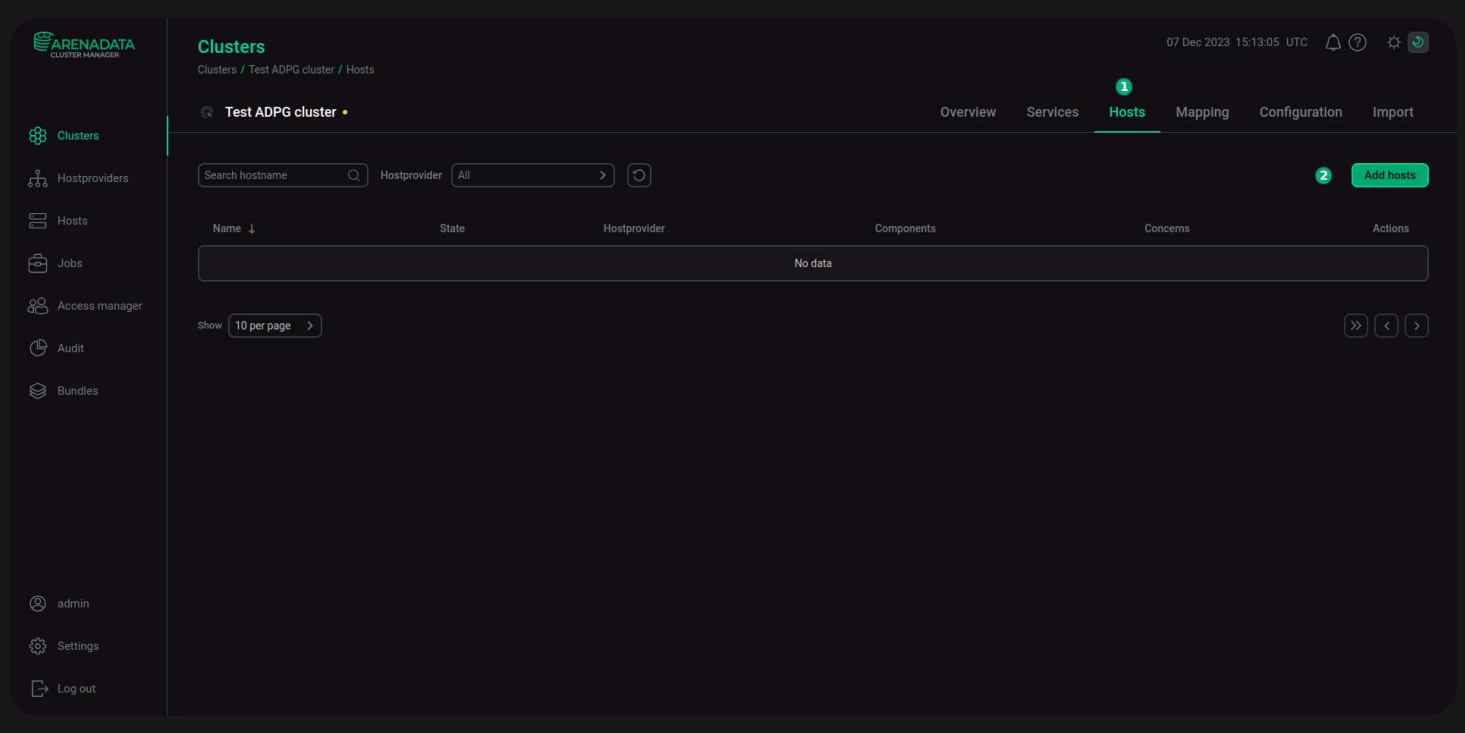
The steps for adding hosts to a cluster are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



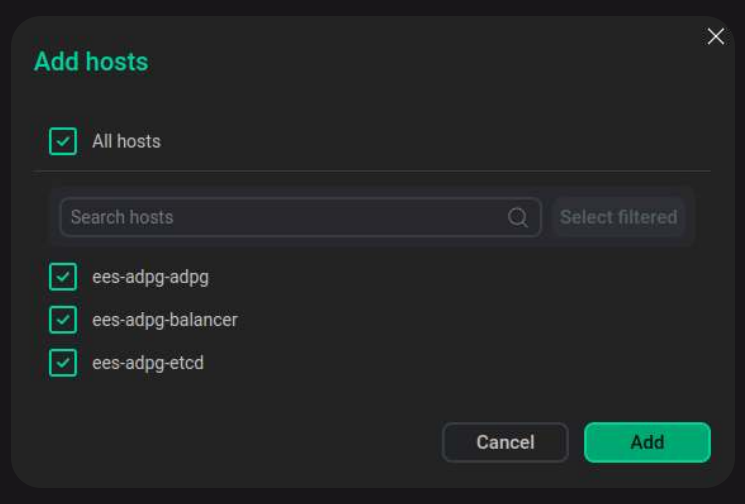
Select a cluster

2. Open the ***Hosts*** tab on the cluster page and click ***Add hosts***.



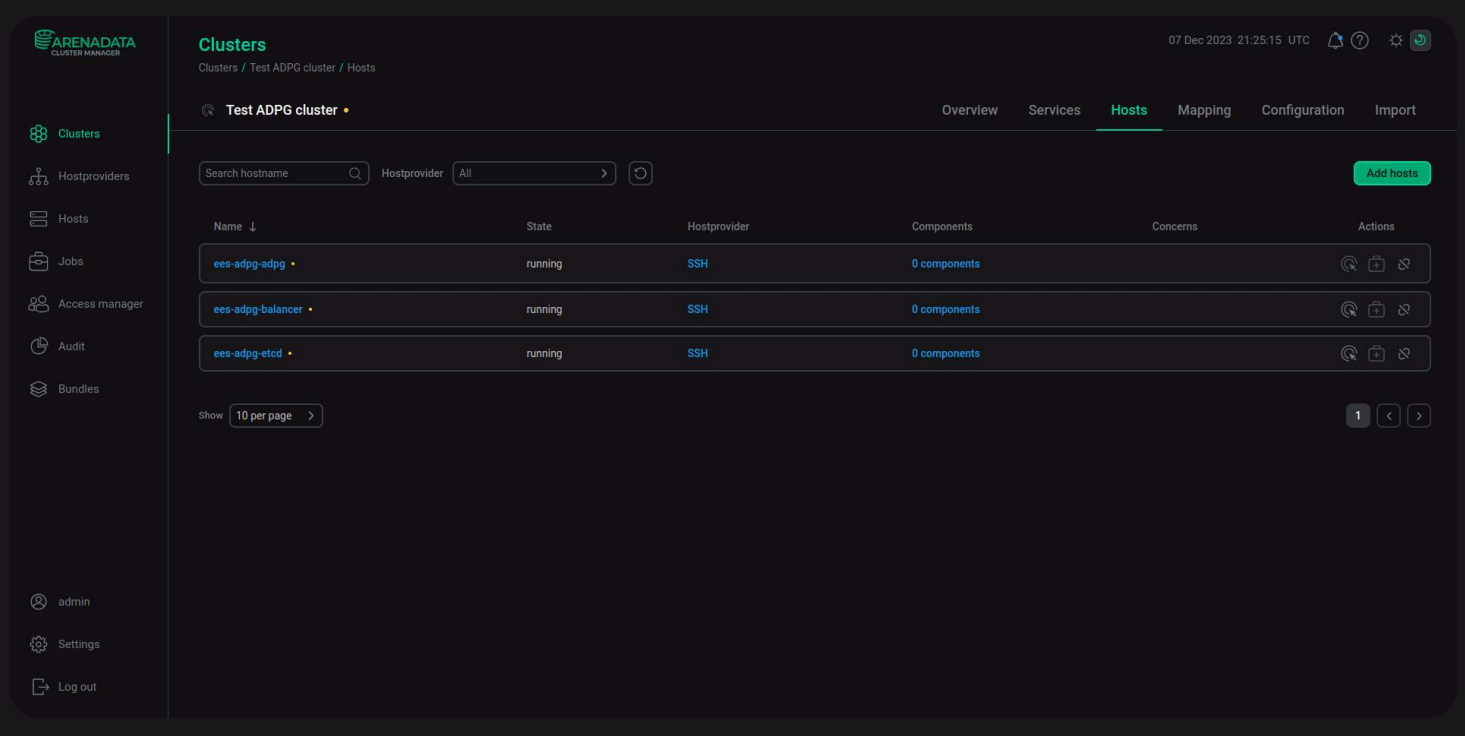
Switch to adding hosts

3. In the opened dialog, select hosts that should be added to the cluster and click ***Add***.



Select hosts

4. As a result, the added hosts are displayed on the ***Hosts*** tab.



The result of successful adding hosts to a cluster

Add components

Daria Baryshева, Eugenia Kuzina

AD.PG

To Table of Contents

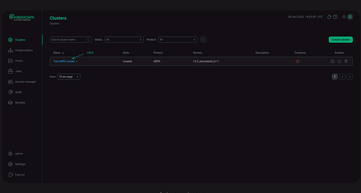
In ADCM a **component** means a part of a service that should be deployed on one or several cluster hosts.

IMPORTANT

Initially, there are no components on any of cluster hosts. You should map components to hosts manually.

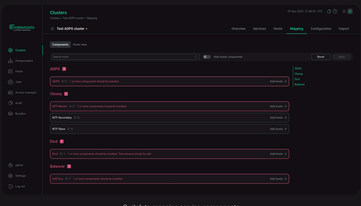
The steps for adding components to hosts are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



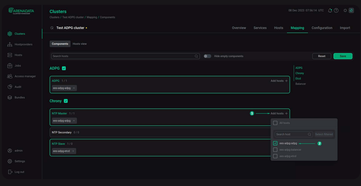
Select a cluster

2. Open the ***Mapping*** tab on the cluster page.



Switch to mapping service components

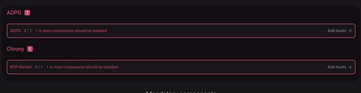
3. In the window that opens, map components to hosts. To assign hosts for each component, click ***Add hosts*** and select hosts in the pop-up window.



Select hosts for a component

Mandatory components

Each service can have mandatory and optional components. Mandatory components are highlighted in red. The ADCM UI also displays how many mandatory components should be installed.



Mandatory components

Currently, ADPG and NTP Master components are mandatory and should be placed on one host.

CAUTION

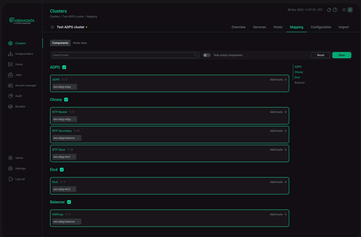
Without assigning a necessary number of hosts to mandatory components, the component mapping cannot be saved.

An example of components distribution for the ADPG Enterprise Edition is shown in the table below.

ADPG components distribution

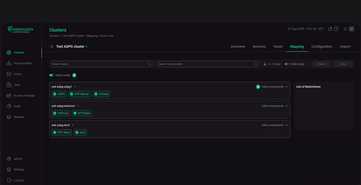
Host	Service	Component	Notes
Host 1	ADPG	ADPG	Can be a leader or replica in the HA architecture
	ADPG Control Agents	ADPG Control Agent	An agent that allows you to manage an ADPG cluster with ADPG Control
		ADPG Control ADBM Agent	A backup manager agent that ADPG Control uses
	Chrony	NTP Master	—
Host 2	ADPG	ADPG	Can be a leader or replica in the HA architecture
	ADPG Control Agents	ADPG Control Agent	An agent that allows you to manage an ADPG cluster with ADPG Control
		ADPG Control ADBM Agent	A backup manager agent that ADPG Control uses
	Chrony	NTP Secondary	—
Host 3	ADPG	ADPG	Can be a leader or replica in the HA architecture
	ADPG Control Agents	ADPG Control Agent	An agent that allows you to manage an ADPG cluster with ADPG Control
		ADPG Control ADBM Agent	A backup manager agent that ADPG Control uses
	Chrony	NTP Secondary	—
Host 4	Etcd	Etcd	You must use an odd number of nodes for the Etcd component. Three or more nodes are recommended
	Chrony	NTP Slave	—
Host 5	Etcd	Etcd	You must use an odd number of nodes for the Etcd component. Three or more nodes are recommended
	Chrony	NTP Slave	—
Host 6	Etcd	Etcd	You must use an odd number of nodes for the Etcd component. Three or more nodes are recommended
	Chrony	NTP Slave	—
Host 7	Balancer	HAProxy	—

4. After component distribution is completed, click ***Save***.



Save mapping of components

Starting with ADCM 2.2.0, you can use another mapping mode — by hosts. To do this, switch on the ***Hosts mode*** toggle and map components to hosts by clicking ***Add components*** for each host sequentially.



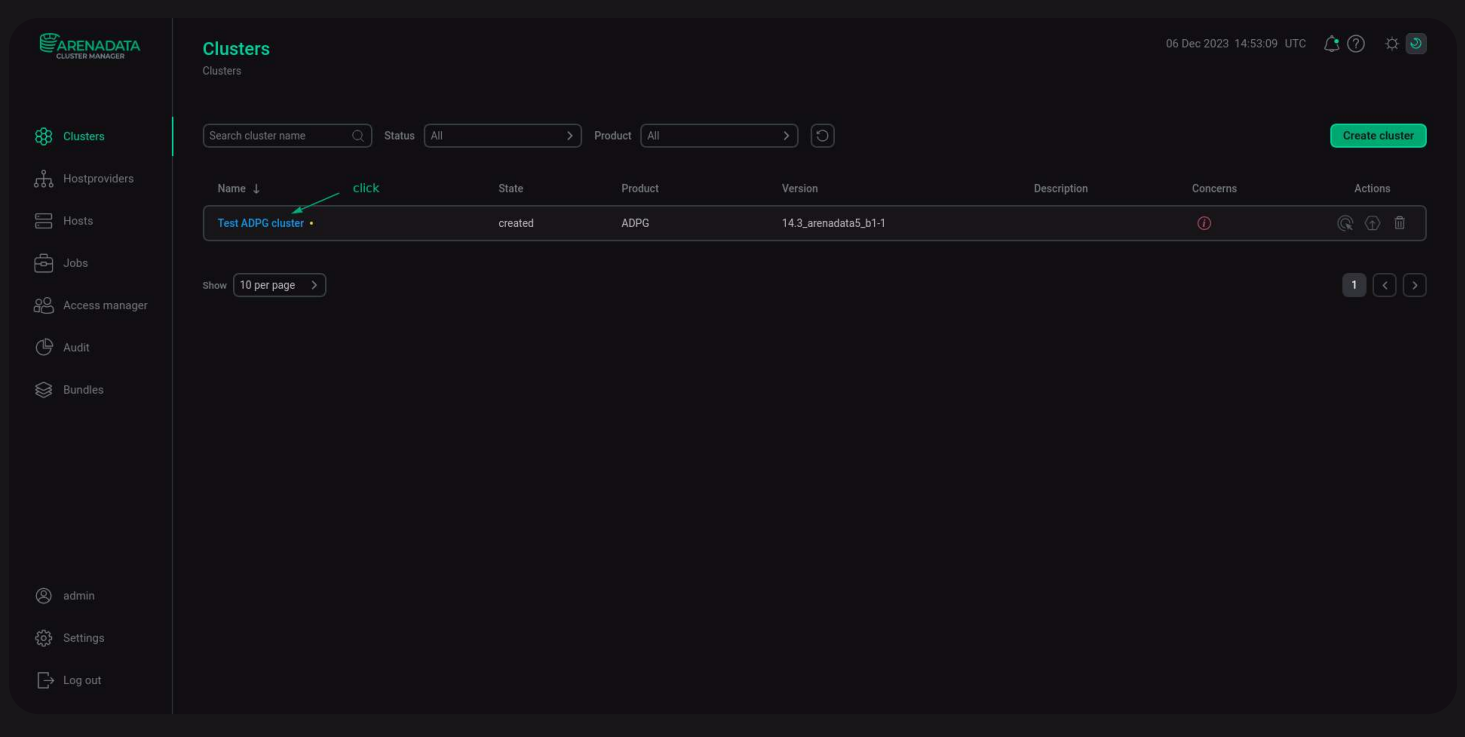
Use mapping by hosts

Configure services

Eugenia Kuzina, Daria Barysheva

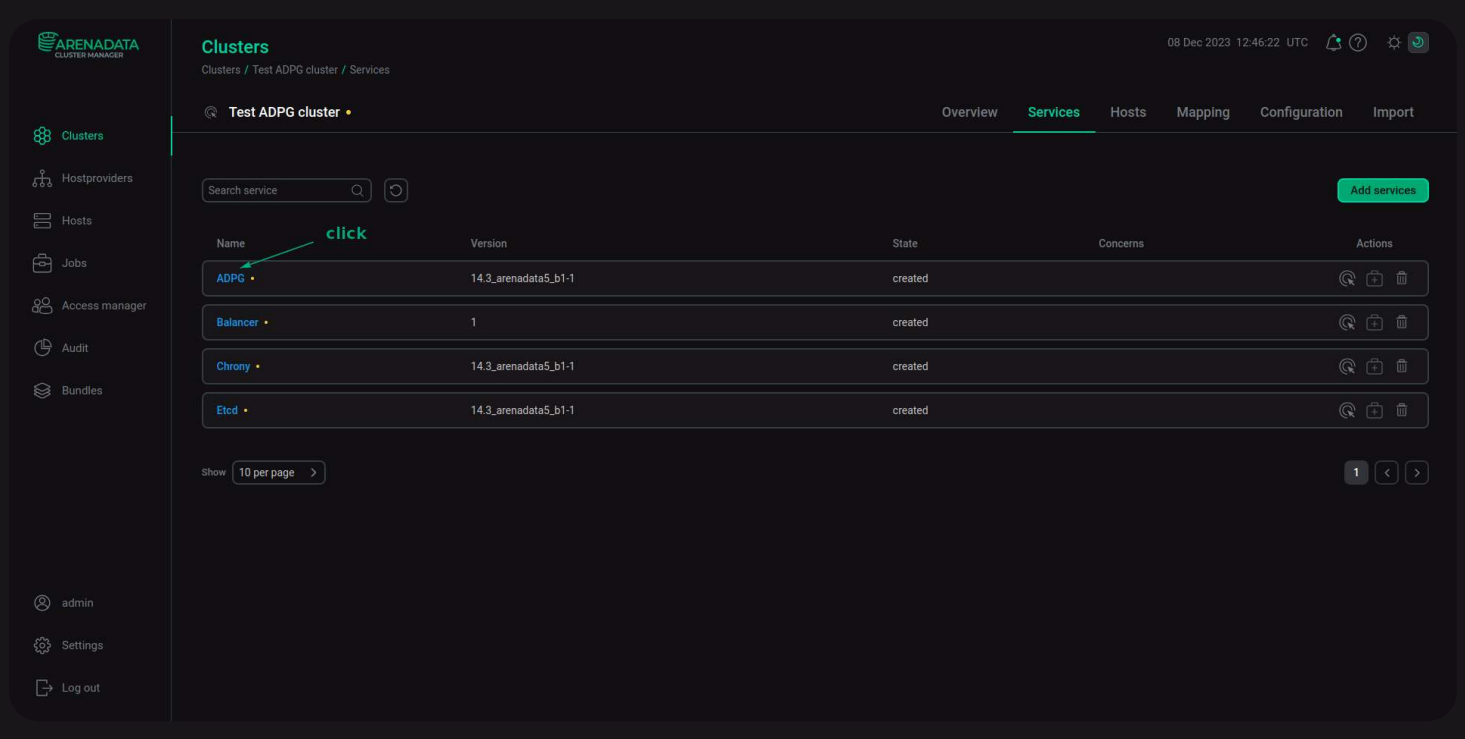
To Table of Contents

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



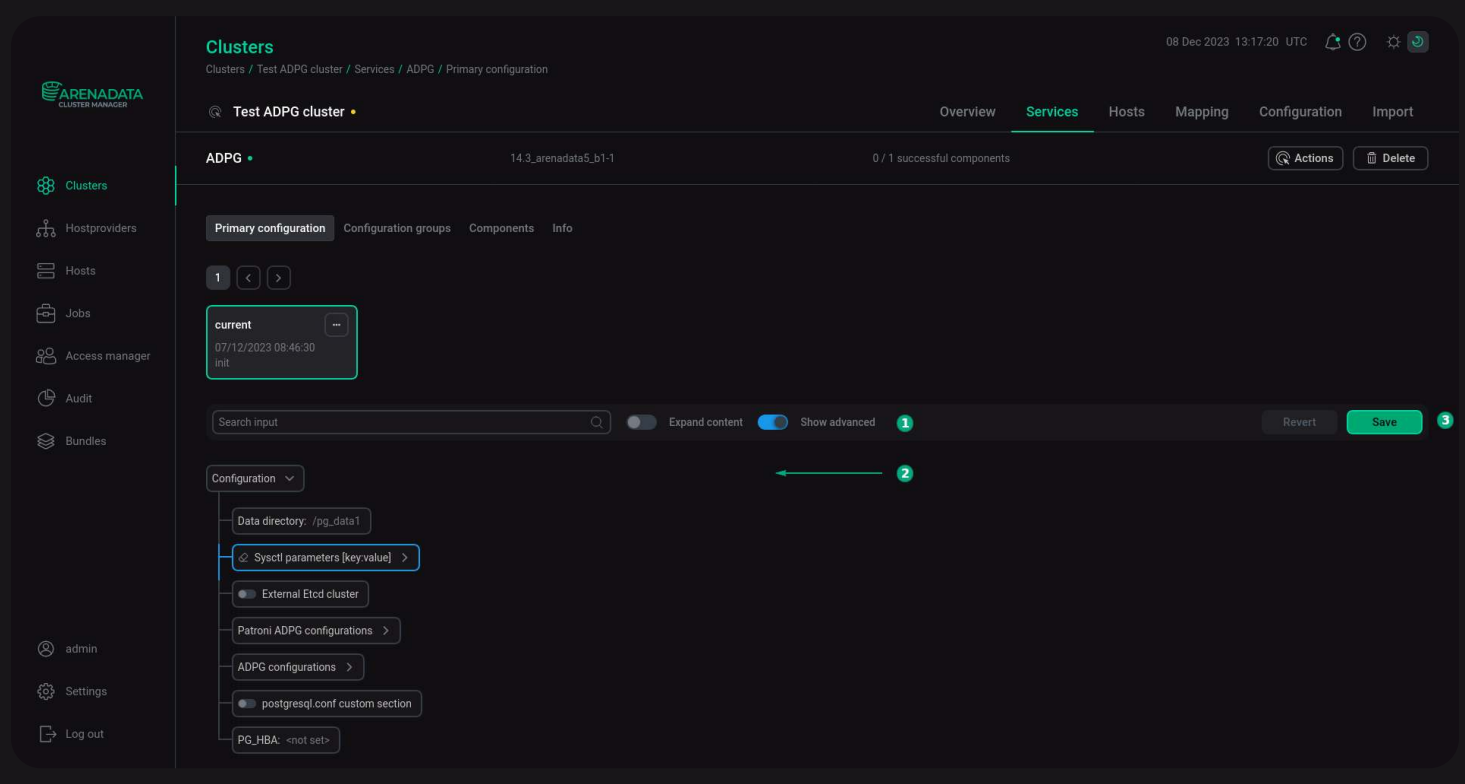
Select a cluster

2. Open the ***Services*** tab on the cluster page, find the row that contains the desired service, and click its name in the ***Name*** column.



Switch to configuring a service

3. In the window that opens:
   - Open the ***Primary Configuration*** tab.
   - Switch on the ***Show advanced*** toggle.
   - Fill in configuration parameters of the selected service. The fields highlighted in red are mandatory.
   - Click ***Save***.



Configure a service

Pay attention to the following parameters:

- ***Chrony → NTP servers***. Though this parameter is optional, we recommend to set it to the nearest NTP server, e.g. the one that your company uses.

NOTE

- If you need to install the ADPG Control ADBM Agent component that belongs to the ADPG Control Agents service, make sure that the Enable backups toggle button is disabled. This component is a backup manager, and it is impossible to use both backup methods simultaneously.
- For more information about service configuration parameters, see Configuration parameters in the References.

Configure a cluster

Daria Baryshevа, Eugenia Kuzina

Contents

To Table of Contents

Primary configuration

Configuration groups

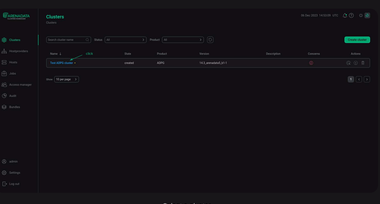
Ansible settings

IMPORTANT

There is usually no need to change cluster configuration parameters. You can leave all parameters at the default values.

After creation a new cluster, you can configure it by performing the following steps:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



Select a cluster

2. Open the ***Configuration*** tab on the cluster page. The ***Configuration*** tab includes the following sections: ***Primary configuration***, ***Configuration groups***, ***Ansible settings***.
3. Fill in all necessary parameters on the selected tab and click ***Save***.

Primary configuration

The ***Configuration*** → ***Primary configuration*** tab allows you to configure the general cluster settings.

Available configuration parameters are listed below. Basically, they describe paths to the repositories that are used during the installing process — depending on the cluster operating system (ALT Linux 8, Astra Linux 1.7, CentOS 7, or Ubuntu 22.04). Only the marked repositories are registered on the cluster servers. For each parameter, you can change the default url to the required one.

General configuration parameters

Parameter	Description
Cluster UUID	Unique cluster identifier
Disable firewall	Disables **firewalld** on the cluster hosts

Altlinux 8 repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

Astra linux repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

Centos 7 repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

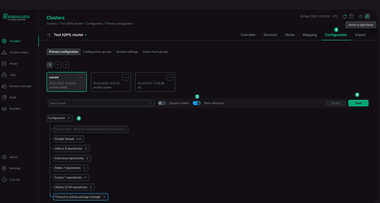
Ubuntu 22.04 repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

Timeout to unlock package manager

The section is available when the ***Show advanced*** toggle button is activated. Parameters of this section are intended to set the timeout of the DNF software package manager unlocking wait. In Ubuntu, after the host reboot, automatic update starts, which locks DNF.

Parameter	Description	Default value
retries	The maximum number of retries	3
delay	The time delay between retries (in seconds)	10



The cluster "Configuration" tab

Configuration groups

The ***Configuration*** → ***Configuration groups*** tab is designed to set cluster configuration groups.

Ansible settings

The ***Configuration*** → ***Ansible settings*** tab allows you to set Ansible configuration options at the cluster level. The tab is available starting with ADCM 2.2.0.

Ansible settings

Name	Description	Default
forks	The number of parallel processes to spawn when communicating with remote hosts	5

Install a cluster

Contents

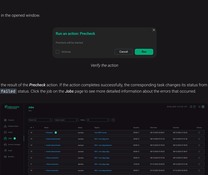
To Table of Contents

Step 1. Precheck
Step 2. Run installation
  Whole cluster
  Single services
Step 3. Check the cluster state after installation
Step 4. Connect to PostgreSQL via psql

Step 1. Precheck

Before installing a new cluster it is recommended to run the **Precheck** action. By doing this, you make sure that all necessary repositories are available, the cluster does not contain hosts with the same IP addresses, and all other prerequisites are met. Do the following:

1. Apply the **Precheck** action to the cluster selected on the **Clusters** page by clicking the icon in the **Actions** column.

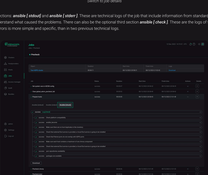


Run the cluster precheck

2. Verify the requested action in the opened window.

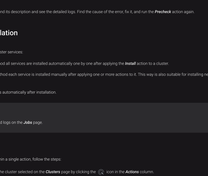
Verify the action

3. Open the **Jobs** page to see the result of the **Precheck** action. If the action completes successfully, the corresponding task changes its status from `running` to `success`. If the job fails, it switches to the `failed` status. Click the job on the **Jobs** page to see more detailed information about the errors that occurred.



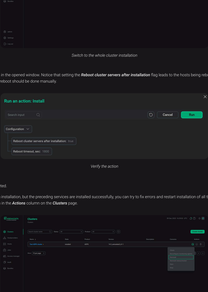
View jobs

The window with job inner steps is opened. Click the ellipsis in the **Details** column for additional information on the current step.



Switch to job details

A job page contains two sections: **ansible [stdout]** and **ansible [stderr]**. These are technical logs of the job that include information from standard I/O streams stdout/stderr. These logs can help to understand what caused the problems. There can also be the optional third section **ansible [check]**. These are the logs of the most frequent errors checks. The description of these errors is more simple and specific, than in two previous technical logs.



Job details

4. If a task fails, you can expand its description and see the detailed logs. Find the cause of the error, fix it, and run the **Precheck** action again.

Step 2. Run installation

There are two ways to install cluster services:

1. **Whole cluster** - in this method all services are installed automatically one by one after applying the **Install** action to a cluster.
2. **Single services** - in this method each service is installed manually after applying one or more actions to it. This way is also suitable for installing new services in the already deployed cluster.

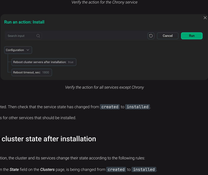
In both cases each service starts automatically after installation.

> **NOTE**
> In case of errors, you can find logs on the **Jobs** page.

Whole cluster

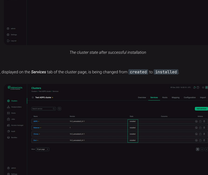
To install all cluster services within a single action, follow the steps:

1. Apply the **Install** action to the cluster selected on the **Clusters** page by clicking the icon in the **Actions** column.



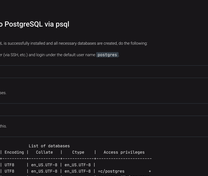
Switch to the whole cluster installation

2. Verify the requested action in the opened window. Notice that setting the **Reboot cluster servers after installation** flag leads to the hosts being reboot after the installation is completed. Otherwise, the reboot should be done manually.



Verify the action

3. Wait until the job is completed.
4. If some service fails during installation, but the preceding services are installed successfully, you can try to fix errors and restart installation of all the failed services. To do this, choose the **Reinstall** action in the **Actions** column on the **Clusters** page.



Installation restart

Single services

To install one or several cluster services manually, add these services to the cluster and then perform the actions described in the following subsections.

> **CAUTION**
> Notice that services are installed one by one. Do not install another service until the installation of the previous one is completed.

1. Open the **Services** tab on the cluster page. For each service that should be installed, click the icon in the **Actions** column and select the **Install** action.

Switch to installation of single services

2. Verify the action in the opened window.

Verify the action for the Chrony service

Verify the action for all services except Chrony

3. Wait until the job is completed. Then check that the service state has changed from `created` to `installed`.
4. Repeat the previous actions for other services that should be installed.

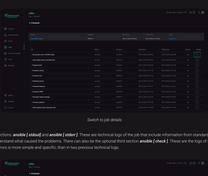
Step 3. Check the cluster state after installation

As a result of successful installation, the cluster and its services change their state according to the following rules:

- The cluster state, displayed in the **State** field on the **Clusters** page, is being changed from `created` to `installed`.

The cluster state after successful installation

- The state of cluster services, displayed on the **Services** tab of the cluster page, is being changed from `created` to `installed`.



The state of cluster services after successful installation

Step 4. Connect to PostgreSQL via psql

In order to check that PostgreSQL is successfully installed and all necessary databases are created, do the following:

1. Connect to the ADPG Server (via SSH, etc.) and login under the default user name `postgres`:

```
$ sudo su - postgres
```

2. List all PostgreSQL databases:

```
$ psql -l
```

The result should look like this.

```
                                  List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    |   Access privileges
-----------+----------+----------+------------+------------+-----------------------
postgres   | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
template0  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
template1  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
(3 rows)
```

3. Use one of the two commands:

- `psql` - to connect to the default database,
- `psql -d<database_name>` - to connect to the database with the specified name `<database_name>`.

```
$ psql postgres
```

After this you can send queries to the selected database via the `psql` client.

```
postgres=# SELECT 1 AS test;
 test
------
    1
(1 row)
```

Install monitoring

Eugenia Kuzmina

Contents

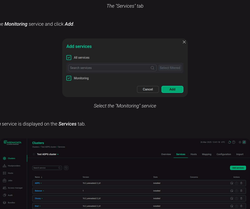
To Table of Contents

Step 1. Add the Monitoring service
Step 2. Configure monitoring
Step 3. Install the Monitoring service
Step 4. View dashboards

ADPG bundle includes a monitoring system. To install monitoring, perform the steps described below.

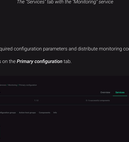
Step 1. Add the Monitoring service

Open the **Clusters** page in ADCM UI and select an ADPG cluster. Open the **Services** tab and click **Add services**.



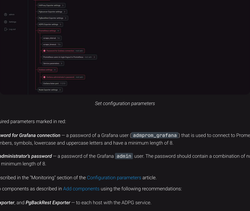
The "Services" tab

In the dialog that opens, select the **Monitoring** service and click **Add**.



Select the "Monitoring" service

As a result, the added **Monitoring** service is displayed on the **Services** tab.

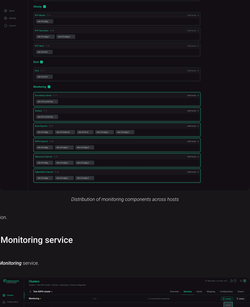


The "Services" tab with the "Monitoring" service

Step 2. Configure monitoring

The **Concerns** icon (!) indicates that you need to specify required configuration parameters and distribute monitoring components across hosts.

Click the **Monitoring** service and set configuration parameters on the **Primary configuration** tab.



Set configuration parameters

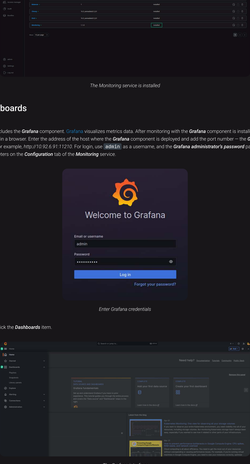
You should set the following required parameters marked in red:

- **Prometheus settings** → **Password for Grafana connection** – a password of a Grafana user `adpgmon_grafana` that is used to connect to Prometheus. The password should contain a combination of numbers, symbols, lowercase and uppercase letters and have a minimum length of 8.
- **Grafana settings** → **Grafana administrator's password** – a password of the Grafana `admin` user. The password should contain a combination of numbers, symbols, lowercase and uppercase letters and have a minimum length of 8.

All monitoring parameters are described in the "Monitoring" section of the Configuration parameters article.

Then open the **Mapping** tab. Map components as described in Add components using the following recommendations:

- *ADPG Exporter*, *Pgbouncer Exporter*, and *PgBackRest Exporter* – to each host with the ADPG service.
- *Prometheus Server* and *Grafana* – to a separate host.
- *Node Exporter* – to each host from which you need to collect system metrics.

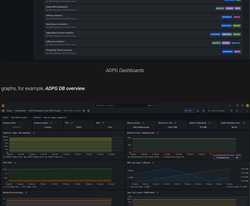


Distribution of monitoring components across hosts

Click **Save** to save the configuration.

Step 3. Install the Monitoring service

Execute the **Install** action of the **Monitoring** service.



The "Install" action of the Monitoring service

In the next dialog window, click **Run** to confirm the action. You can also activate the **Verbose** checkbox to see additional execution details on the **Jobs** page.

Confirm the "Install" action

As a result, the Monitoring service state is changed to `installed`.

The Monitoring service is installed

Step 4. View dashboards

The ADPG monitoring system includes the **Grafana** component. Grafana visualizes metrics data. After monitoring with the **Grafana** component is installed, you will be able to view dashboards that display metrics in a browser. Enter the address of the host where the **Grafana** component is deployed and add the port number – the **Grafana** *listen port* parameter (the default value is `3123`). For example, http://10.92.1.49:11725. For login, use `admin`, as a username, and the **Grafana administrator's password** parameter value as a password. You can find the Grafana parameters on the **Configuration** tab of the **Monitoring** service.

Enter Grafana credentials

Expand the Grafana menu and click the **Dashboards** item.

The Grafana interface

The page with the dashboard list will be displayed. Expand the **ADPG Dashboard...** item for your cluster to display available dashboards.

ADPG Dashboards

Click on any desired item to view graphs, for example, **ADPG DB overview**.

ADPG DB overview graphs

Offline installation

To Table of Contents

This topic describes **offline** installation of ADPG. It can be applied to the hosts without access to Internet.

For offline installation, you need to install and configure Arenadata Enterprise Tools (ET). ET is a solution that provides the necessary infrastructure for Arenadata product deployment in environments with limited internet access. The solution is a part of the Arenadata data platform. It provides the repositories necessary for installing and updating the platform components.

The process of offline installation via ADCM includes the following steps.

NOTE

This function is available only in the Enterprise Edition of ADPG.

Step 1.

Install ADCM

Step 2.

Prepare hosts

Step 3.

Install ET cluster

Step 4.

Install ADPG cluster

Step 5.

Install monitoring

Step 6.

Install ADPG Control

Install ADCM

Contents

Step 1. Check prerequisites
Software and hardware requirements
Install Docker
Disable SELinux
Open firewalld
Step 2. Install ADCM
Step 3. Start ADCM
Step 4. Check installation
Check the Docker container status
Check the RPM package availability
Check the ADCM web interface
Get the ADCM URL
Step 5. Update ADCM

ADCM is easy to deploy and maintain. Before installing ADCM, we recommend to read the [Architecture](#) section.

The installation process is as follows:

Step 1. Check prerequisites

Software and hardware requirements

Whether you choose to run ADCM in a container or from an RPM package, it needs to provide:

- One of the following operating systems:
  - Astra Linux
  - CentOS 7
  - RHEL 7
  - Oracle Linux 7
  - RHEL 8 or 9
  - Oracle Linux 8 or 9
  - AlmaLinux 8 or 9 (tested)
  - Rocky Linux 8 or 9 (tested)
- Hardware that meets the following conditions:

RECOMMENDED

Suitable for production environments. For larger setups the hardware can be allocated to the customer requirements.

- CPU: 4 CPU cores
- RAM: 16 GB
- Disk: 100 GB

MINIMUM

Starting with ADCM 2.1.1, we have a Docker image for arm64 architecture (Apple M1/M2, AWS Graviton) and the sequence is made ARM-native (for example, Postgres). Also, we have a Docker image for amd64 architecture (Intel, AMD). Starting with ADCM 2.2.0, we have RPM packages for amd64 architecture (Intel, AMD).

Suitable only for test purposes, without a guarantee of full functionality.

- CPU: 2 CPU cores
- RAM: 8 GB
- Disk: 40 GB

Centos 7 RHEL 7 OEL 7 (7.6+) Linux (any 3rd)

```
$ sudo systemctl stop firewalld
$ sudo systemctl disable firewalld
```

Oracle 8 (8 Minimal) Linux (8) 9 Linux

```
$ sudo systemctl stop firewalld
$ sudo systemctl disable firewalld
```

Astra Linux (SE 1.7.3)

Astra Linux (SE 1.7.3) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-4.15.0-76-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.4)

Astra Linux (SE 1.7.4) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.5)

Astra Linux (SE 1.7.5) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.6)

Astra Linux (SE 1.7.6) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.7)

Astra Linux (SE 1.7.7) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

RED OS 7.3

Install Docker as described in the Docker documentation.

To disable SELinux open the file **/etc/selinux/config** in the text editor and set the SELINUX parameter:

```
SELINUX=disabled
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

RED OS 7.3 MUROM

Install Docker as described in the Docker documentation.

To disable SELinux open the file **/etc/selinux/config** in the text editor and set the SELINUX parameter:

```
SELINUX=disabled
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

RED OS 7.3.1

Install Docker as described in the Docker documentation.

To disable SELinux open the file **/etc/selinux/config** in the text editor and set the SELINUX parameter:

```
SELINUX=disabled
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

RED OS 7.3.2

Install Docker as described in the Docker documentation.

To disable SELinux open the file **/etc/selinux/config** in the text editor and set the SELINUX parameter:

```
SELINUX=disabled
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Disable SELinux

CentOS

SELinux is enabled in Docker by default. If you disabled it, run the following:

To disable SELinux open the file **/etc/selinux/config** in the text editor and set the SELINUX parameter:

```
SELINUX=disabled
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.3)

Astra Linux (SE 1.7.3) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-4.15.0-76-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.4)

Astra Linux (SE 1.7.4) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.5)

Astra Linux (SE 1.7.5) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.6)

Astra Linux (SE 1.7.6) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Astra Linux (SE 1.7.7)

Astra Linux (SE 1.7.7) has SELinux by default. You need to disable it. Open the file /boot/grub/grub.cfg and find the string with linux:

```
linux /boot/vmlinuz-5.4.0-33-generic root=UUID=<UUID> ro quiet splash vt.handoff=7
```

Add the following to the end of the string:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

RED OS 7.3

Add the following to the end of the line that starts with linux in the file /boot/grub2/grub.cfg:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

RED OS 7.3.1

Add the following to the end of the line that starts with linux in the file /boot/grub2/grub.cfg:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

RED OS 7.3.2

Add the following to the end of the line that starts with linux in the file /boot/grub2/grub.cfg:

```
selinux=0
```

Reboot the system:

```
$ sudo reboot
```

Check that SELinux is disabled:

```
$ sestatus
SELinux status:                 disabled
```

Open firewalld

CentOS

If the firewall is enabled, open the ports 8000, 8020, 8443, 8501, 8502, 8503, 8504, 8505, 8506, 8507, 8508, 8509, 8510, 8511, 8512, 8513, 8514, 8515, 8516, 8517, 8518, 8519, 8520, 8521, 8522, 8523, 8524, 8525, 8526, 8527, 8528, 8529, 8530, 8531, 8532, 8533, 8534, 8535, 8536, 8537, 8538, 8539, 8540,

Prepare hosts

To Table of Contents

To manage a cluster and its nodes via ADCM, it is necessary to prepare hosts for a cluster and link these hosts to ADCM. This is the function of ADCM **hostproviders** — special plugins for ADCM, which encapsulate the complex mechanism of interaction between ADCM and the hosts controlled by it.

ADCM supports several types of hostproviders. For more information on how to configure hostproviders and use them to create hosts, refer to the following documentation:

- SSH
- Yandex Cloud
- VCloud (VMware)
- Cloud.ru Advanced
- CROC Cloud

For offline installation, the SSH hostprovider is recommended.

IMPORTANT

It is forbidden to use the reserved words `adpg`, `balancer`, `chrony`, `etcd`, and `prometheus_monitoring` as hostnames in an ADPG cluster.

Make sure that routing tables on each host contain the default route — `0.0.0.0/0`.

Use the `route` command to check this:

```
$ route
```

The output should be similar to:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         gateway         0.0.0.0         UG    100    0        0 eth0
10.92.40.0      0.0.0.0         255.255.252.0   U     100    0        0 eth0
```

If the `0.0.0.0/0` default route is missing in the output, add it using the following command:

```
$ sudo route add default gw 0.0.0.0 eth0
```

Create a cluster

Daria Barysheva

Contents

To Table of Contents

Step 1. Get the Enterprise version of a cluster bundle

Step 2. Upload a cluster bundle to ADCM

Step 3. Create a cluster based on the uploaded bundle

In ADCM a **cluster** means a set of services running on a set of hosts.

Clusters distributions for ADCM are made in the form of so-called bundles. Generally a bundle is a regular archive that includes a description and software logic of cluster deployment and exploitation.

The steps for adding a cluster to ADCM are listed below.

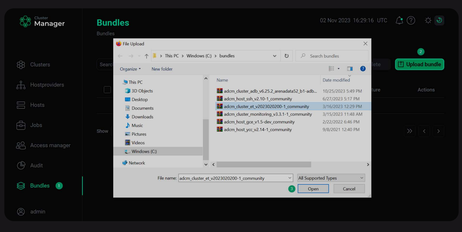
Step 1. Get the Enterprise version of a cluster bundle

Please get a cluster bundle from the Arenadata support team.

Step 2. Upload a cluster bundle to ADCM

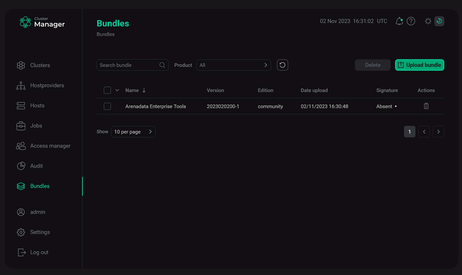
To upload a bundle to ADCM, follow the steps:

1. Select the ***Bundles*** item in the left navigation menu and click ***Upload bundle***.
2. Select a bundle in the Open File Dialog.



Upload a bundle

3. As a result of the performed actions, a bundle is displayed on the ***Bundles*** page.



The result of successful uploading a bundle

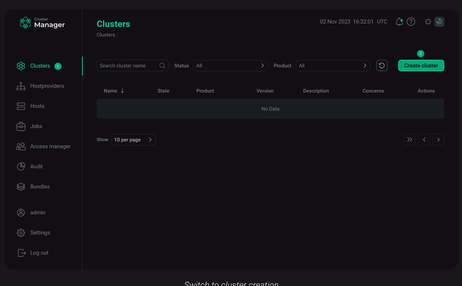
Step 3. Create a cluster based on the uploaded bundle

IMPORTANT

When creating a cluster via the ADCM web interface, you generate a new ADCM cluster instance. It means only adding the cluster to ADCM — at this stage no services are deployed.

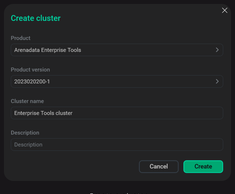
The steps for cluster creation are given below:

1. Select the ***Clusters*** item in the left navigation menu and click ***Create cluster***.



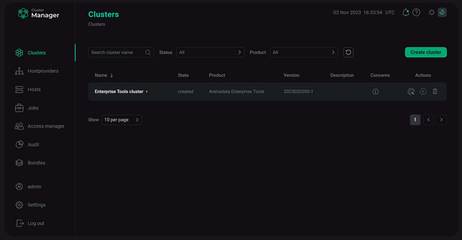
Switch to cluster creation

2. In the opened dialog, describe a new cluster instance:
   - Select an uploaded cluster bundle in the ***Product*** field.
   - Select a bundle version in the ***Product version*** field. Several versions become available in the case of different versions of the same bundle being uploaded.
   - Enter a cluster name in the ***Cluster name*** field.
   - Enter a cluster description in the ***Description*** field if necessary.
   - Set the ***I accept Terms of Agreement*** flag to sign the license agreement. This flag is displayed only for Enterprise bundles. To read the agreement text, refer to the ***Terms of Agreement*** link. You can also sign the license agreement on the ***Bundles*** page by clicking the desired product bundle.
   - Click ***Create***.



Create a cluster

3. As a result, the created cluster instance is displayed on the ***Clusters*** page.



The result of successful cluster creation

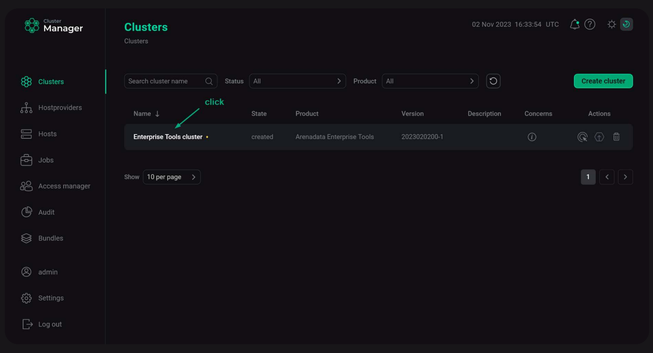
Add services

Daria Barysheva

To Table of Contents

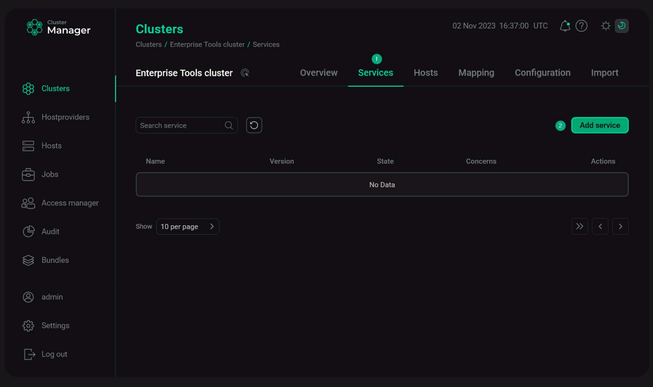
In ADCM a **service** means a software that performs some function. The steps for adding services to a cluster are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



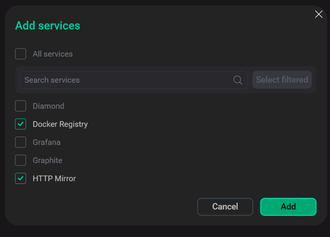
Select a cluster

2. Open the ***Services*** tab on the cluster page and click ***Add service***.



Switch to adding services

3. In the opened dialog, select services that should be added to the cluster and click ***Add***.



Select services

The brief description of available services is listed below.

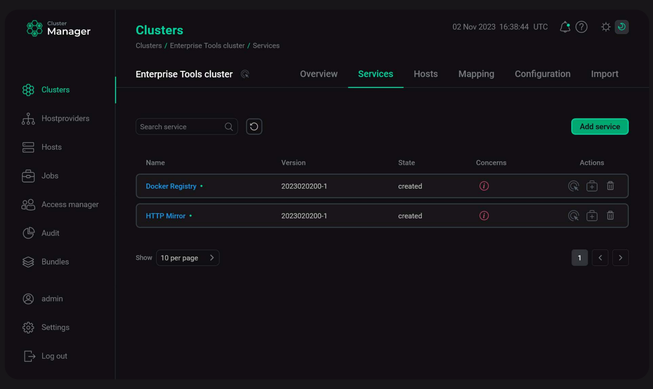
Services that should be added to the Enterprise Tools cluster

Service	Purpose
HTTP Mirror	A service used for hosting packages of the repositories that are needed for offline installation of products
Docker Registry	A service used to store Docker Images that are needed for offline installation of products

NOTE

The CoreDNS service is deprecated — do not select it. The Diamond, Graphite, and Grafana services are used for building monitoring — they will be described separately.

4. As a result, the added services are displayed on the ***Services*** tab.



The result of successful adding services to a cluster

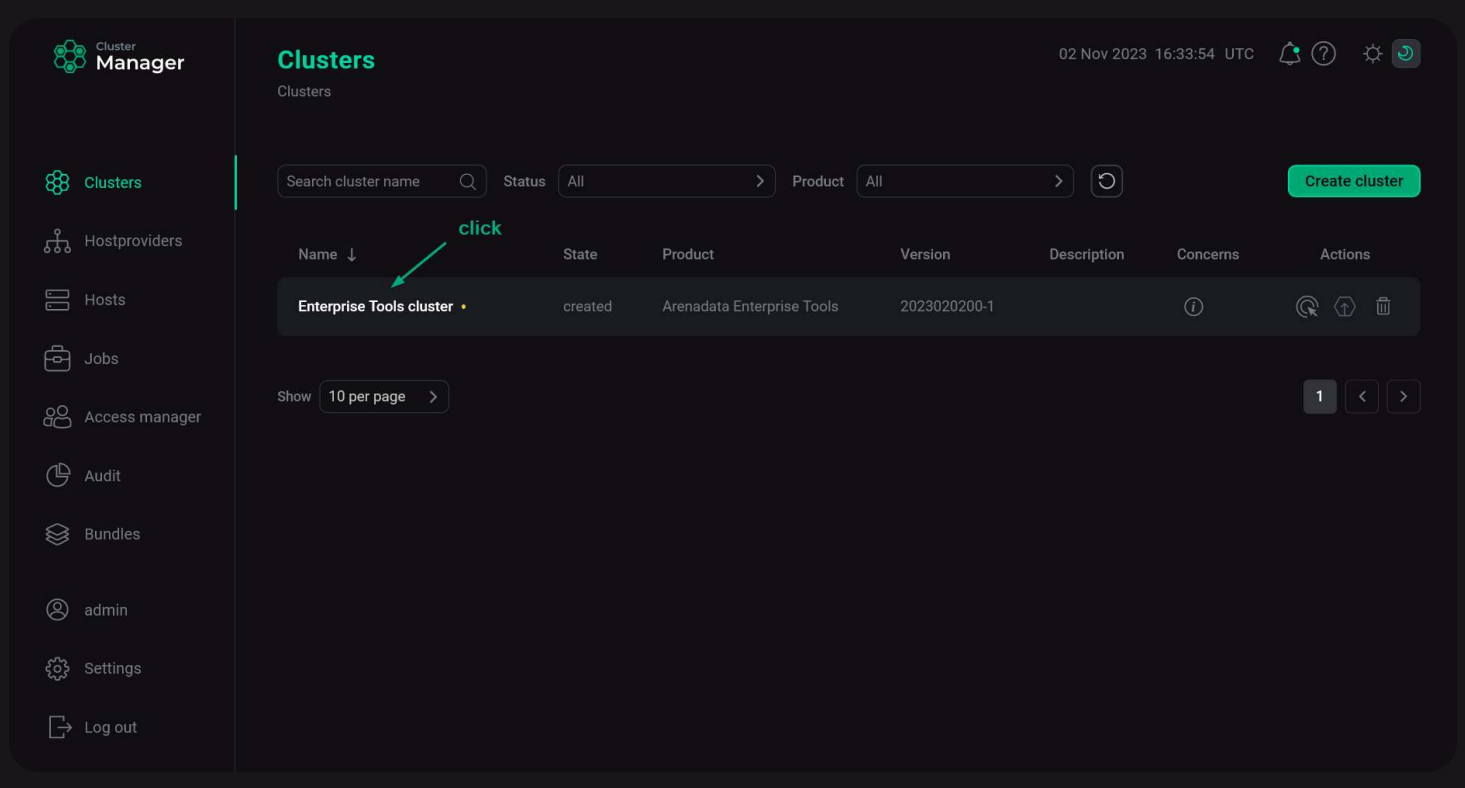
Add hosts to a cluster

Daria Barysheva

To Table of Contents

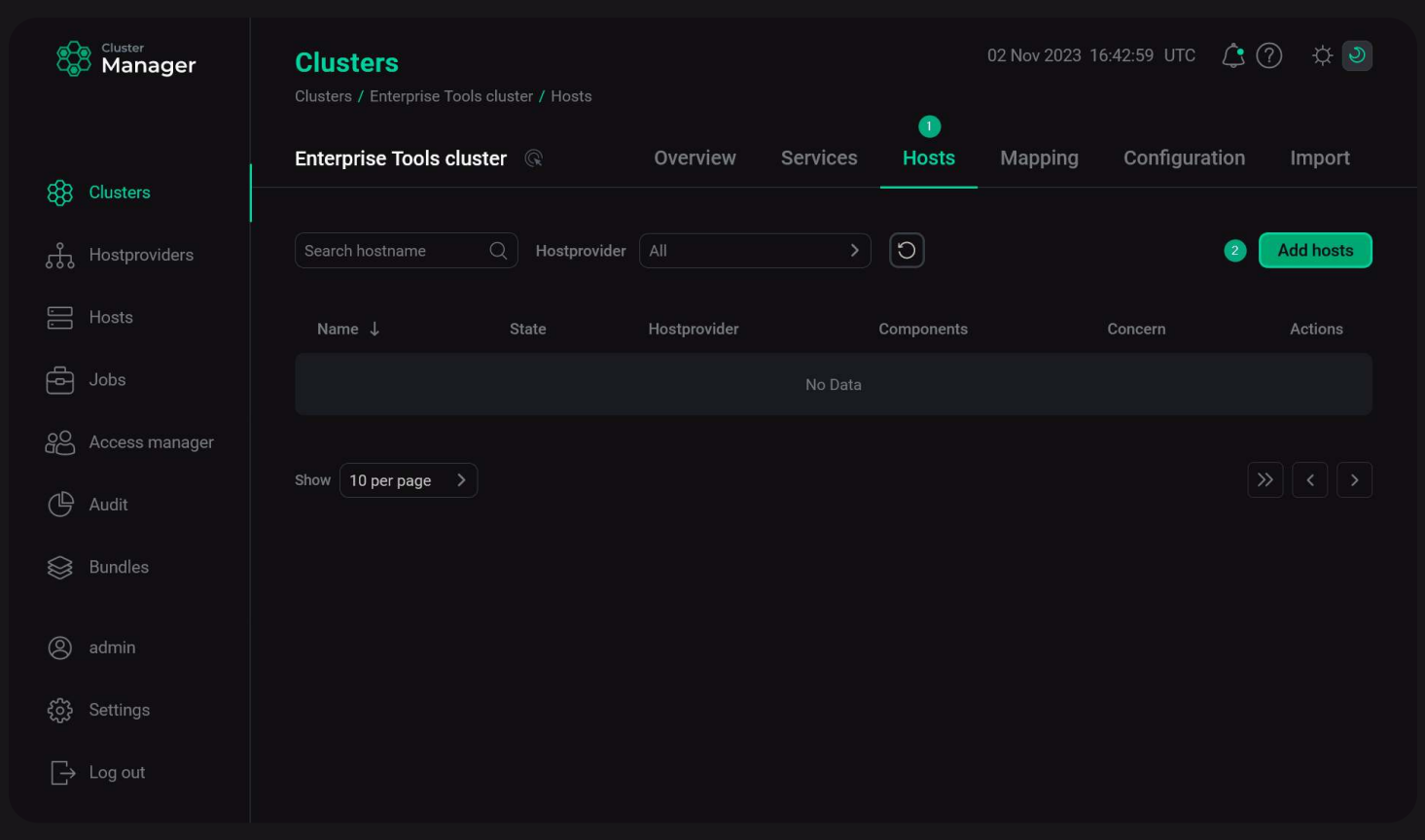
The steps for adding hosts to a cluster are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



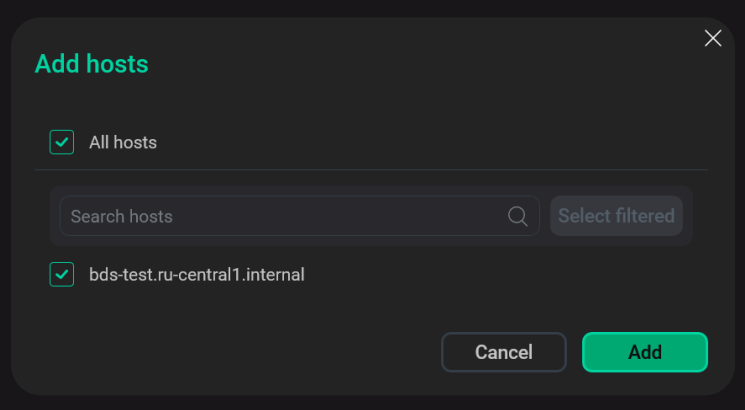
Select a cluster

2. Open the ***Hosts*** tab on the cluster page and click ***Add hosts***.



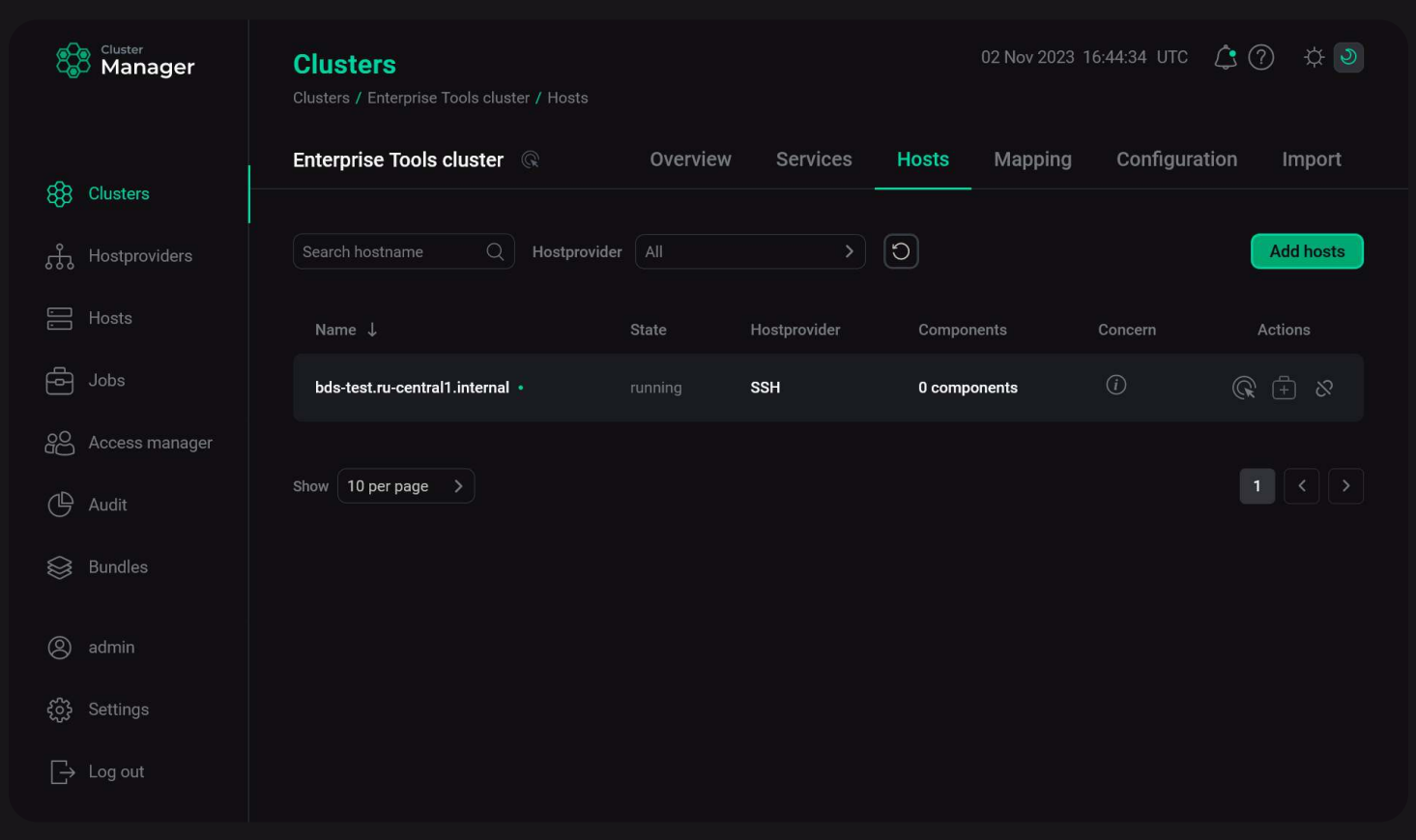
Switch to adding hosts

3. In the opened dialog, select hosts that should be added to the cluster and click ***Add***.



Select hosts

4. As a result, the added hosts are displayed on the ***Hosts*** tab.



The result of successful adding hosts to a cluster

Add components

Daria Barysheva

To Table of Contents

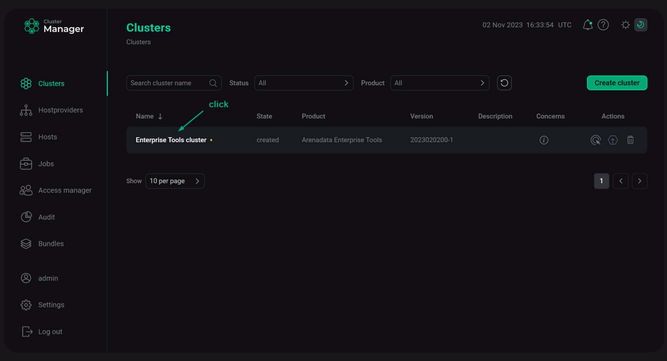
In ADCM a **component** means a part of a service that should be deployed on one or several cluster hosts.

IMPORTANT

Initially, there are no components on any of cluster hosts. You should map components to hosts manually.

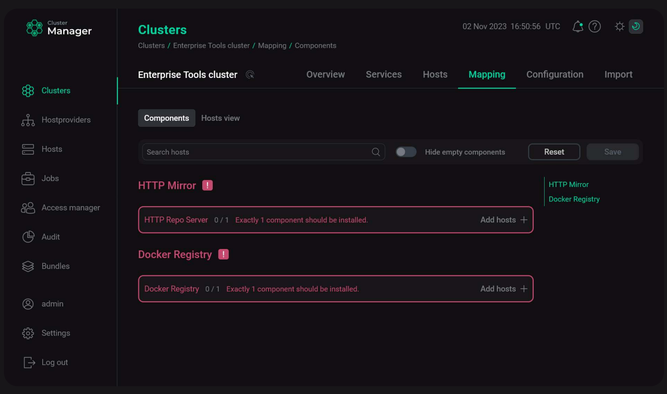
The steps for adding components to hosts are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



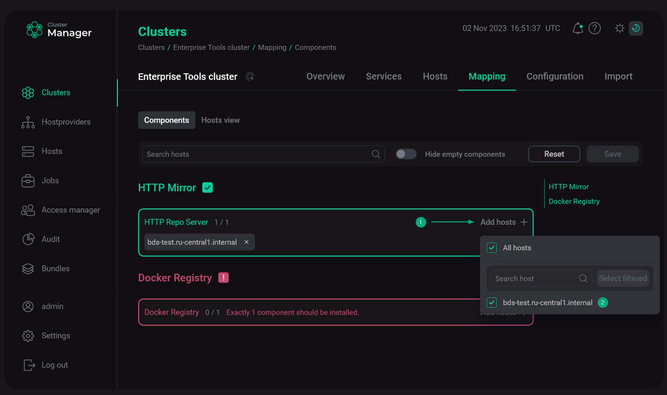
Select a cluster

2. Open the ***Mapping*** tab on the cluster page.



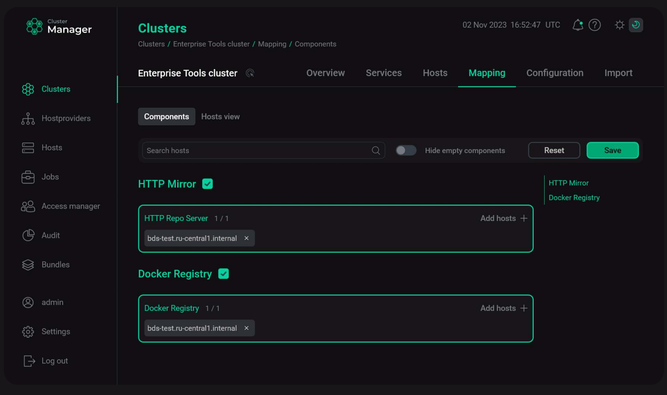
Switch to mapping service components

3. In the window that opens, map components to hosts. To assign hosts for each component, click ***Add hosts*** and select hosts in the pop-up window.



Select hosts for a component

4. After component distribution is completed, click ***Save***.



Save mapping of components

Install a cluster

Daria Barysheva

Contents

To Table of Contents

Step 1. Run installation

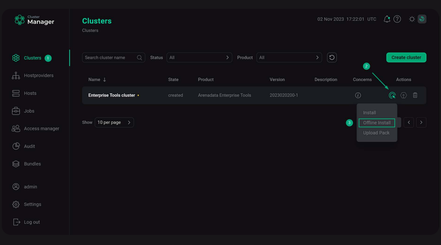
Step 2. Upload packages for offline installation

Step 3. Verify results

Step 1. Run installation

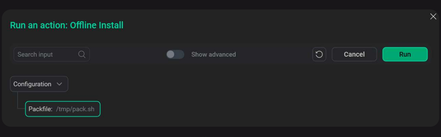
The last step of adding a new Enterprise Tools cluster is to install its services. To do this:

1. Apply the ***Offline Install*** action to the cluster selected on the ***Clusters*** page by clicking the icon in the ***Actions*** column.



Switch to cluster installation

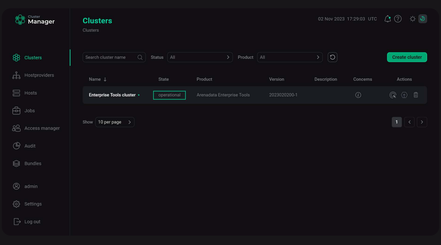
2. In the opened window, enter a full path to the shell script that is received from the Arenadata support team and click ***Run***.



Verify installation

3. Wait until the installation process is completed.

> **NOTE**
>
> In case of errors, you can find logs on the ***Jobs*** page.

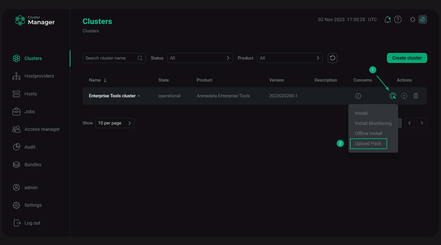


Cluster installation is completed

Step 2. Upload packages for offline installation

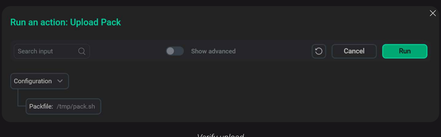
In addition to installing the cluster, you should process the shell script that is received from the Arenadata support team. It is necessary to enable offline installation of products in the future. To do this:

1. Apply the ***Upload Pack*** action to the cluster selected on the ***Clusters*** page by clicking the icon in the ***Actions*** column.



Switch to uploading packages

2. In the opened window, enter a full path to the shell script received from the Arenadata support team and click ***Run***.



Verify upload

3. Wait until the upload process is completed.

> **NOTE**
>
> In case of errors, you can find logs on the ***Jobs*** page.

Step 3. Verify results

The steps for checking the installation results are listed below:

1. Check that ports `443` and `81` are listened at the host where cluster is installed. The first port is used by the Docker Registry service, the second one — by the HTTP Mirror service.

```
$ sudo netstat -ntlp|grep L|grep -E "81|443"
```

The result should look like this:

```
tcp6       0      0 :::443                  :::*                    LISTEN      1694/docker-proxy-c
tcp6       0      0 :::81                   :::*                    LISTEN      1708/docker-proxy-c
```

2. Check that the */opt/arenadata/etools/httprepo/arenadata-repo* folder exists at the host where the cluster is installed. This folder should contain the data necessary for offline installation of the products that you have chosen before getting a shell script from the Arenadata support team.

```
$ ls -la /opt/arenadata/etools/httprepo/arenadata-repo
```

The following example displays the data that can be placed in this folder for ADPG installation.

```
total 0
drwxr-xr-x 5 root root 40 Dec 26 17:46 .
drwxr-xr-x 3 root root 28 Dec 26 17:46 ..
drwxr-xr-x 4 root root 28 Dec 26 17:46 ADM
drwxr-xr-x 3 root root 29 Dec 26 17:46 ADPG
```

Create a cluster

Eugenia Kuzina, Daria Barysheva

Contents

To Table of Contents

Step 1. Get the Enterprise version of a cluster bundle

Step 2. Upload a cluster bundle to ADCM

Step 3. Create a cluster based on the uploaded bundle

In ADCM a **cluster** means a set of services running on a set of hosts.

Clusters distributions for ADCM are made in the form of so-called bundles. Generally a bundle is a regular archive that includes a description and software logic of cluster deployment and exploitation.

The steps for adding a cluster to ADCM are listed below.

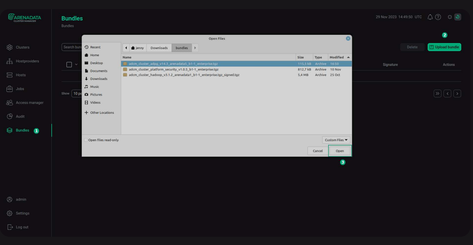
Step 1. Get the Enterprise version of a cluster bundle

Please get a cluster bundle from the Arenadata support team.

Step 2. Upload a cluster bundle to ADCM

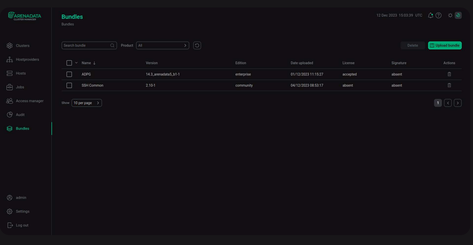
To upload a bundle to ADCM, follow the steps:

1. Select the ***Bundles*** item in the left navigation menu and click ***Upload bundle***.
2. Select a bundle in the Open File Dialog.



Upload a bundle

3. As a result of the performed actions, a bundle is displayed on the ***Bundles*** page.



The result of successful uploading a bundle

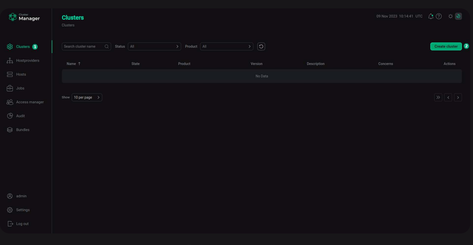
Step 3. Create a cluster based on the uploaded bundle

IMPORTANT

When creating a cluster via the ADCM web interface, you generate a new ADCM cluster instance. It means only adding the cluster to ADCM — at this stage no services are deployed.

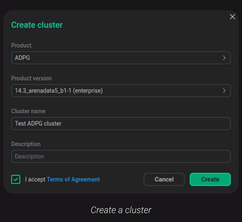
The steps for cluster creation are given below:

1. Select the ***Clusters*** item in the left navigation menu and click ***Create cluster***.



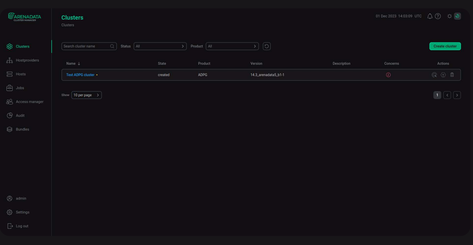
Switch to cluster creation

2. In the opened dialog, describe a new cluster instance:
   - Select an uploaded cluster bundle in the ***Product*** field.
   - Select a bundle version in the ***Product version*** field. Several versions become available in the case of different versions of the same bundle being uploaded.
   - Enter a cluster name in the ***Cluster name*** field.
   - Enter a cluster description in the ***Description*** field if necessary.
   - Set the ***I accept Terms of Agreement*** flag to sign the license agreement. This flag is displayed only for Enterprise bundles. To read the agreement text, refer to the ***Terms of Agreement*** link. You can also sign the license agreement on the ***Bundles*** page by clicking the desired product bundle.
   - Click ***Create***.



Create a cluster

3. As a result, the created cluster instance is displayed on the ***Clusters*** page.



The result of successful cluster creation

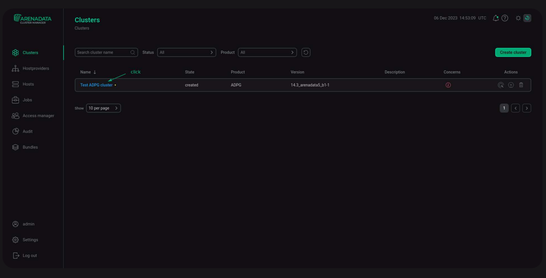
Add services

Eugenia Kuzina, Daria Barysheva

To Table of Contents

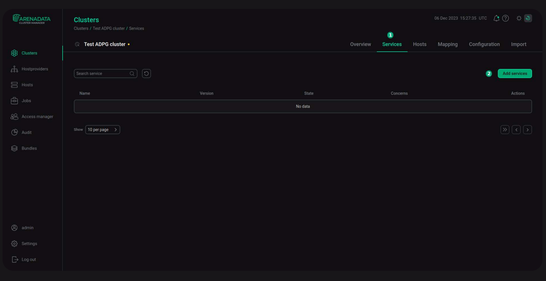
In ADCM a **service** means a software that performs some function. Examples of services for ADPG clusters: ADPG, Chrony, etc. The steps for adding services to a cluster are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



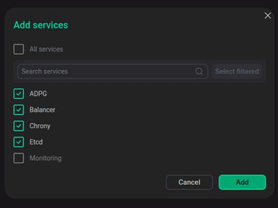
Select a cluster

2. Open the ***Services*** tab on the cluster page and click ***Add service***.



Switch to adding services

3. In the opened dialog, select services that should be added to the cluster and click ***Add***.



Select services

IMPORTANT

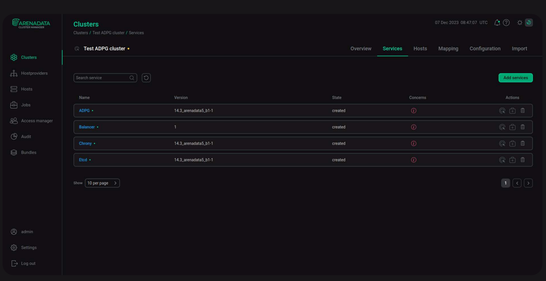
ADPG Enterprise Edition requires etcd (the built-in ADPG Etcd service or an external etcd cluster) to use it as Patroni Distributed Configuration Store (DCS). Its settings must be specified during installation, after installation they cannot be changed.

The brief description of available services is listed below. CE means the ADPG Community Edition, EE — the ADPG Enterprise Edition.

Services that can be added to the ADPG cluster

Service	Mandatory	Edition	Purpose
ADPG	Yes	CE/EE	A core service corresponding to the object-relational DBMS PostgreSQL
ADPG Control Agents	No	EE	A service that allows you to manage an ADPG cluster with ADPG Control
Balancer	No	EE	The service is based on HAProxy that implements load balancing
Chrony	No	CE/EE	A time service. Provides time synchronization for the cluster hosts, that is necessary for the cluster to work correctly. If the time service is already configured on the hosts, there is no need to install Chrony
Etcd	No	EE	The ADPG enterprise edition requires etcd to use it as the Patroni Distributed Configuration Store (DCS). You can utilize the built-in ADPG Etcd service or an external etcd cluster. If you want to use the external etcd cluster, do not add the built-in Etcd service on the ***Services*** tab. Instead, set ***External Etcd cluster*** settings on the ***Primary configuration*** tab of the ADPG service. To specify them, open ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** as described in Configure services
Monitoring	No	CE/EE	Service for collecting and storing ADPG cluster monitoring metrics based on the Prometheus monitoring system and the Grafana service

4. As a result, the added services are displayed at the *Services* tab.



The result of successful adding services to a cluster

NOTE

You can also add services later. The process of adding new services to already running cluster does not differ from installing a service from scratch.

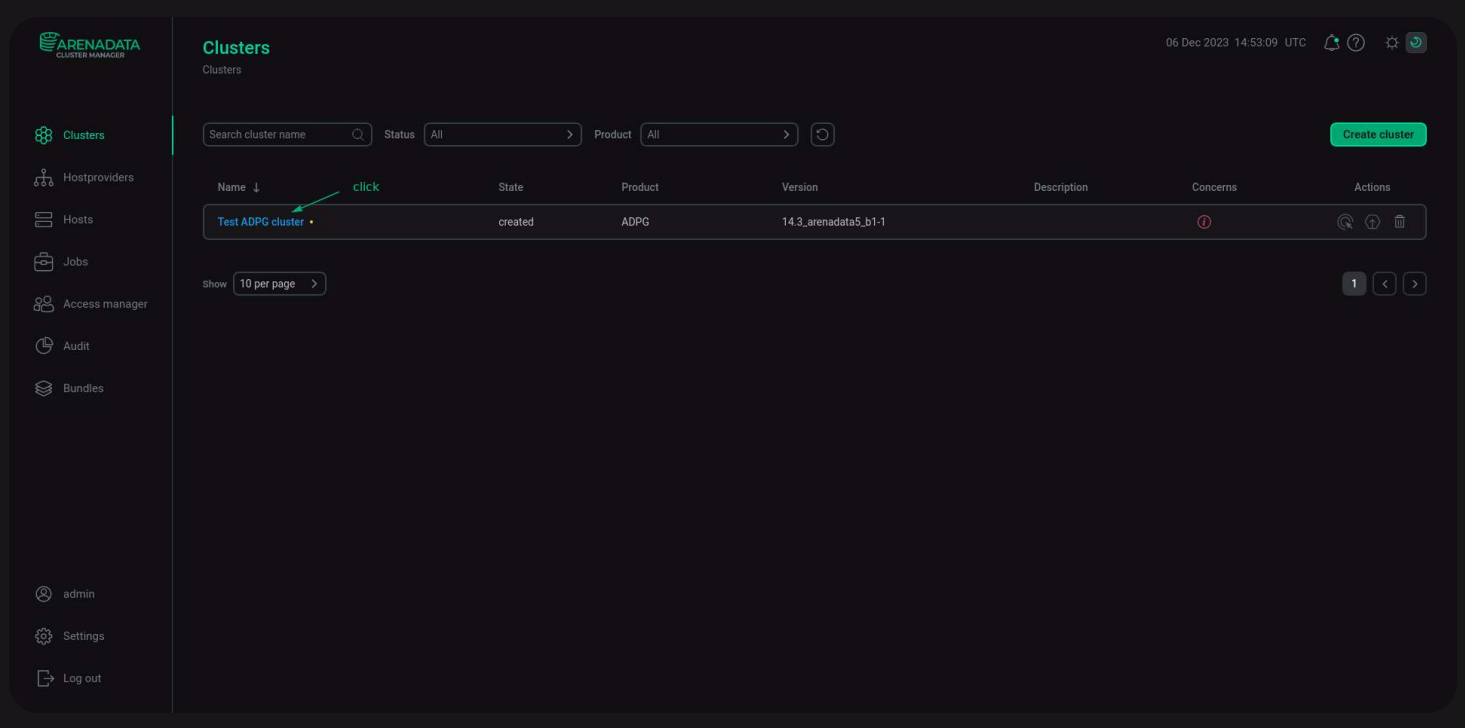
Add hosts to a cluster

Eugenia Kuzina, Daria Barysheva

To Table of Contents

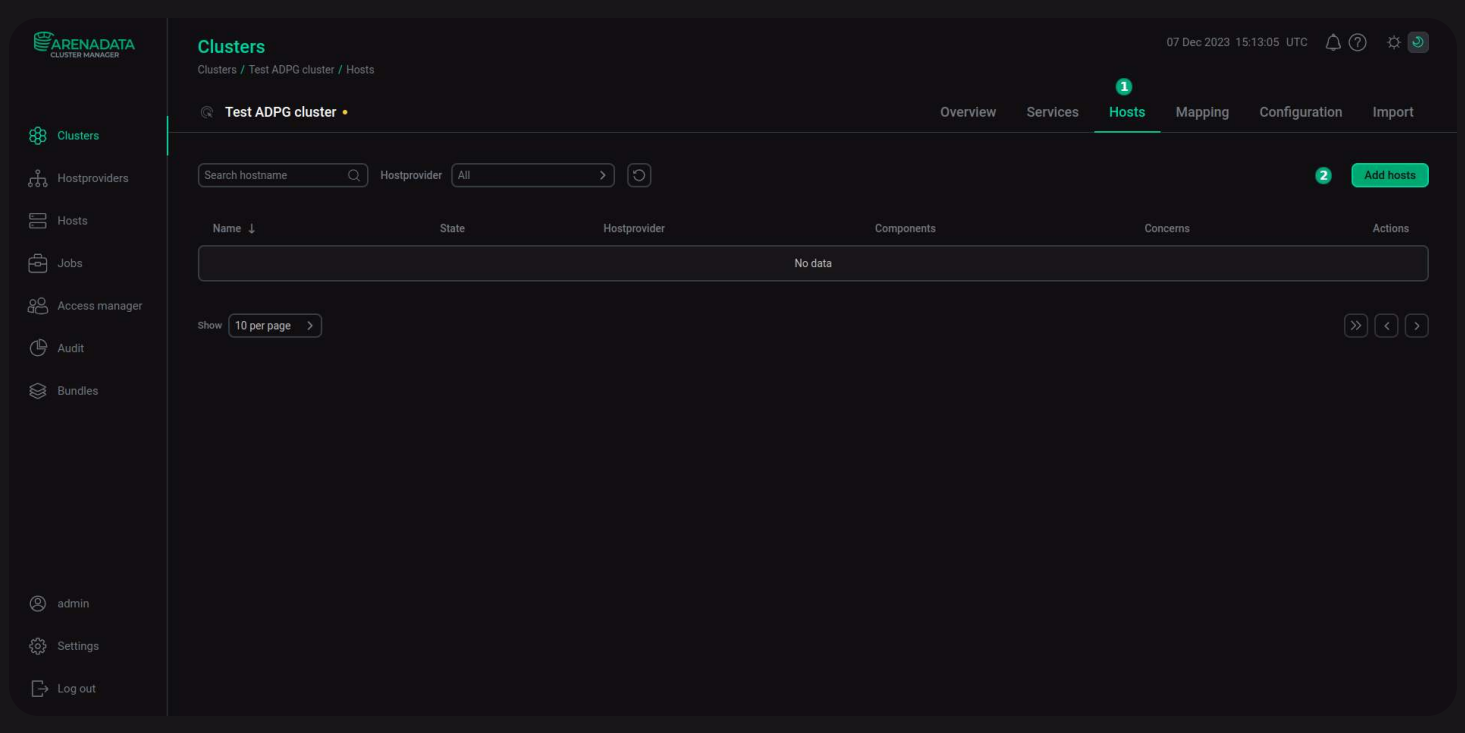
The steps for adding hosts to a cluster are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



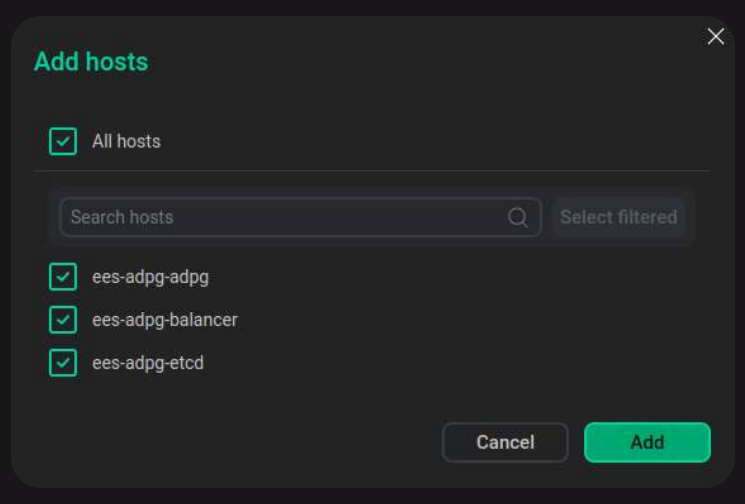
Select a cluster

2. Open the ***Hosts*** tab on the cluster page and click ***Add hosts***.



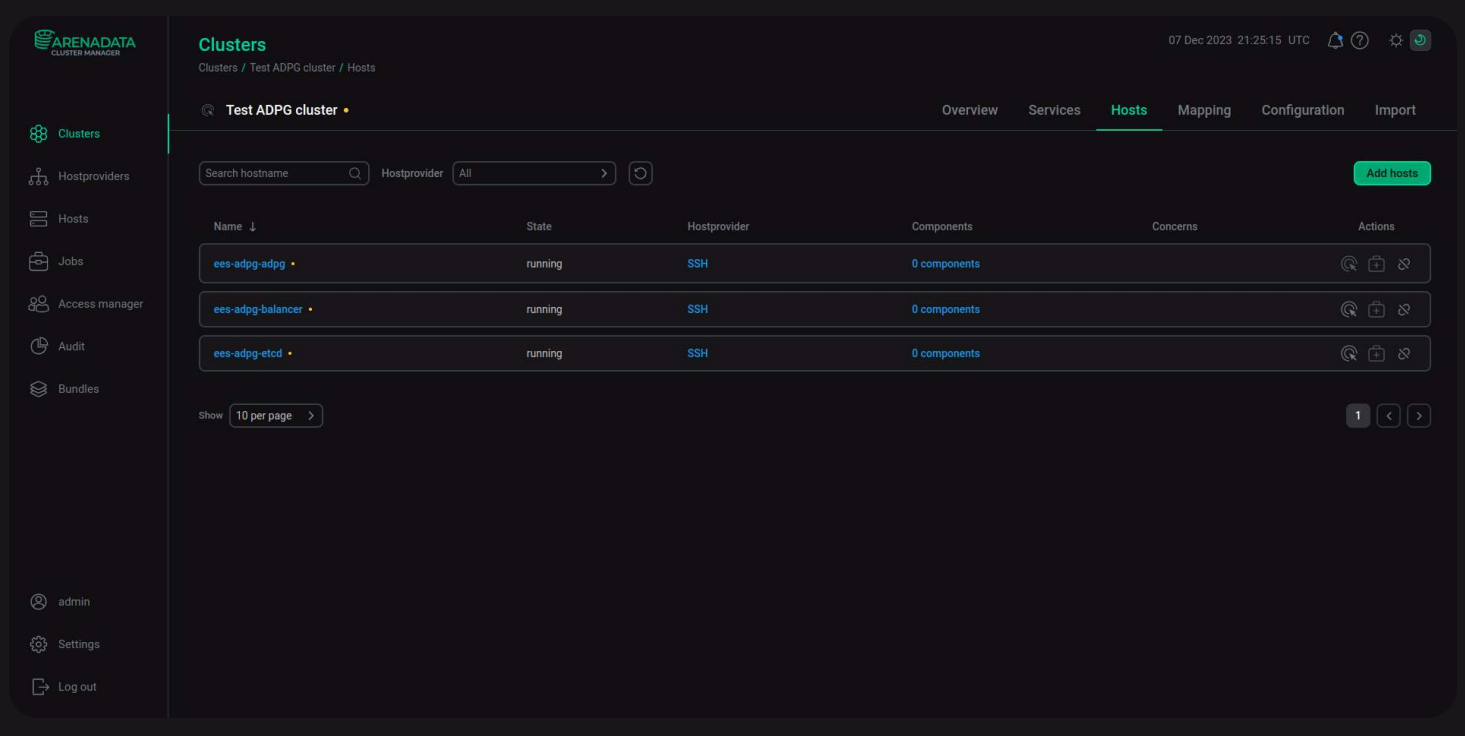
Switch to adding hosts

3. In the opened dialog, select hosts that should be added to the cluster and click ***Add***.



Select hosts

4. As a result, the added hosts are displayed on the ***Hosts*** tab.



The result of successful adding hosts to a cluster

Add components

Eugenia Kuzina, Daria Barysheva

To Table of Contents

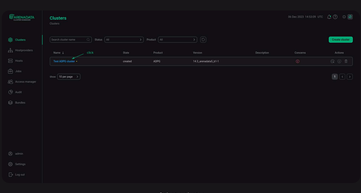
In ADCM a **component** means a part of a service that should be deployed on one or several cluster hosts.

IMPORTANT

Initially, there are no components on any of cluster hosts. You should map components to hosts manually.

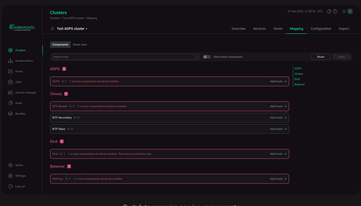
The steps for adding components to hosts are listed below:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



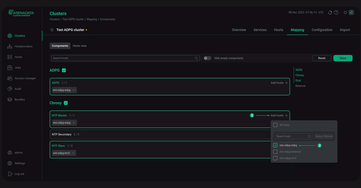
Select a cluster

2. Open the ***Mapping*** tab on the cluster page.



Switch to mapping service components

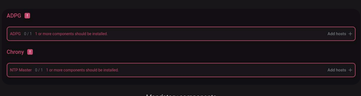
3. In the window that opens, map components to hosts. To assign hosts for each component, click ***Add hosts*** and select hosts in the pop-up window.



Select hosts for a component

Mandatory components

Each service can have mandatory and optional components. Mandatory components are highlighted in red. The ADCM UI also displays how many mandatory components should be installed.



Mandatory components

Currently, ADPG and NTP Master components are mandatory and should be placed on one host.

CAUTION

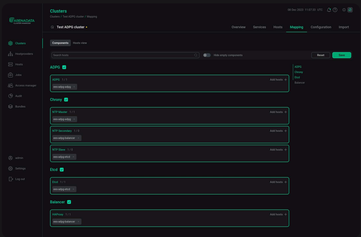
Without assigning a necessary number of hosts to mandatory components, the component mapping cannot be saved.

An example of components distribution for the ADPG Enterprise Edition is shown in the table below.

ADPG components distribution

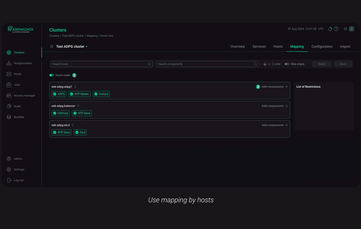
Host	Service	Component	Notes
Host 1	ADPG	ADPG	Can be a leader or replica in the HA architecture
	ADPG Control Agents	ADPG Control Agent	An agent that allows you to manage an ADPG cluster with ADPG Control
		ADPG Control ADBM Agent	A backup manager agent that ADPG Control uses
	Chrony	NTP Master	—
Host 2	ADPG	ADPG	Can be a leader or replica in the HA architecture
	ADPG Control Agents	ADPG Control Agent	An agent that allows you to manage an ADPG cluster with ADPG Control
		ADPG Control ADBM Agent	A backup manager agent that ADPG Control uses
	Chrony	NTP Secondary	—
Host 3	ADPG	ADPG	Can be a leader or replica in the HA architecture
	ADPG Control Agents	ADPG Control Agent	An agent that allows you to manage an ADPG cluster with ADPG Control
		ADPG Control ADBM Agent	A backup manager agent that ADPG Control uses
	Chrony	NTP Secondary	—
Host 4	Etcd	Etcd	You must use an odd number of nodes for the Etcd component. Three or more nodes are recommended
	Chrony	NTP Slave	—
Host 5	Etcd	Etcd	You must use an odd number of nodes for the Etcd component. Three or more nodes are recommended
	Chrony	NTP Slave	—
Host 6	Etcd	Etcd	You must use an odd number of nodes for the Etcd component. Three or more nodes are recommended
	Chrony	NTP Slave	—
Host 7	Balancer	HAProxy	—

4. After component distribution is completed, click ***Save***.



Save mapping of components

Starting with ADCM 2.2.0, you can use another mapping mode — by hosts. To do this, switch on the ***Hosts mode*** toggle and map components to hosts by clicking ***Add components*** for each host sequentially.



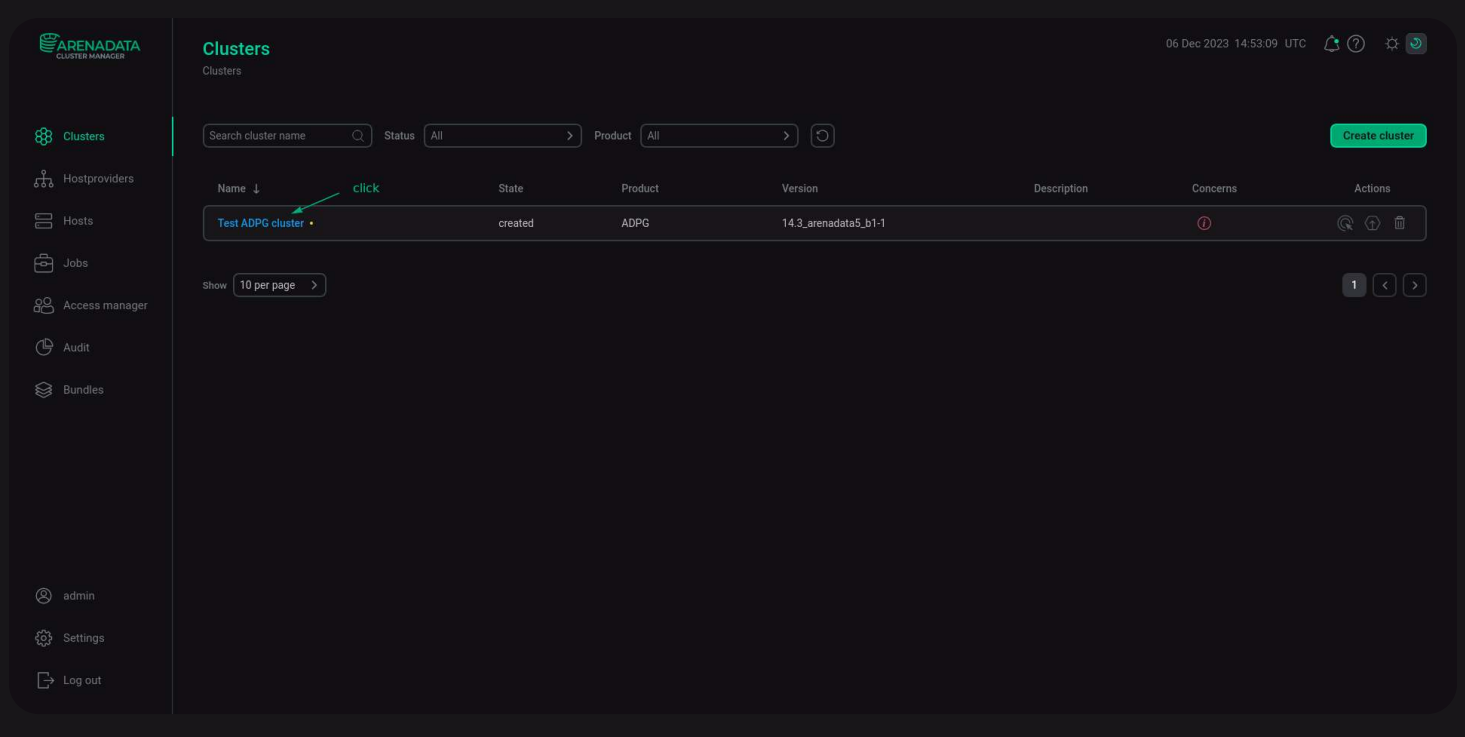
Use mapping by hosts

Configure services

Eugenia Kuzina, Daria Barysheva

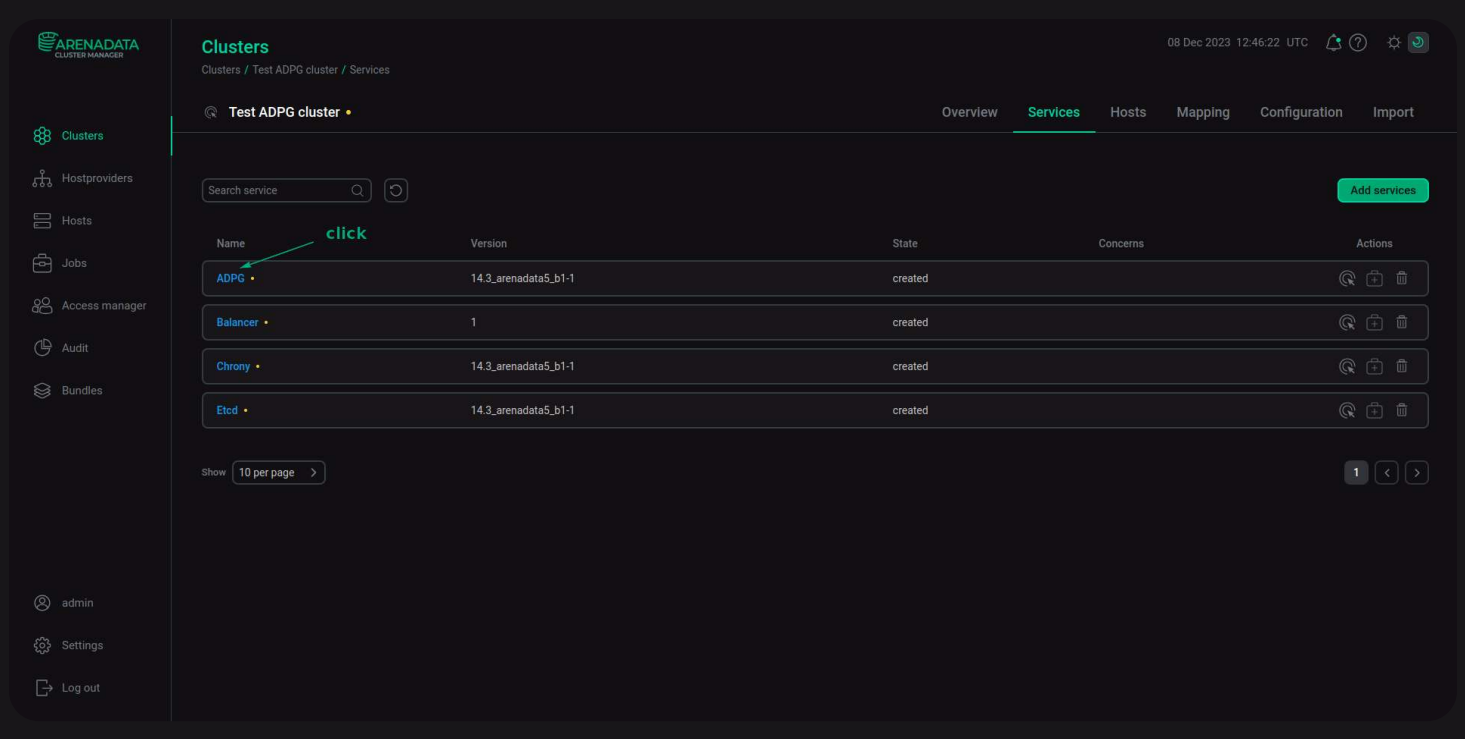
To Table of Contents

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



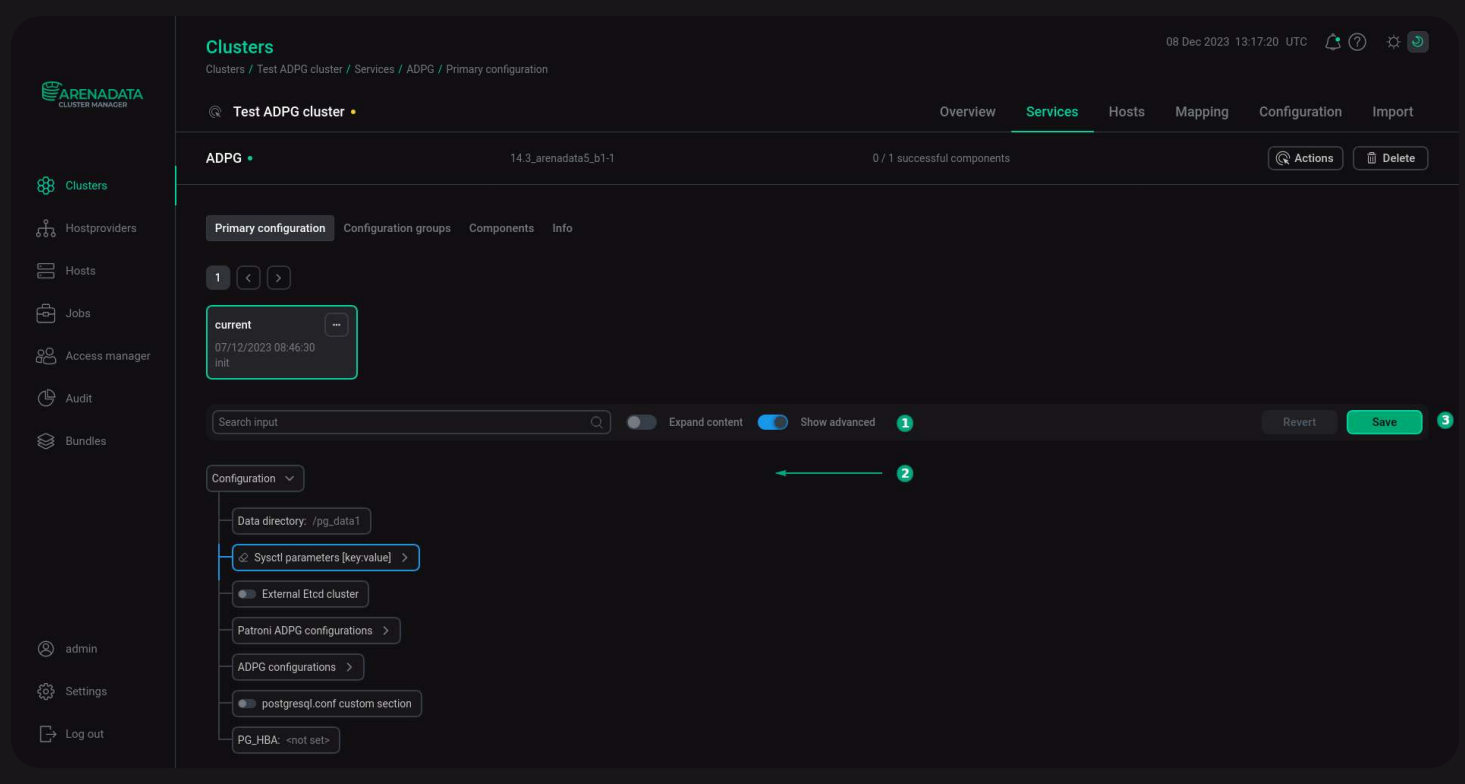
Select a cluster

2. Open the ***Services*** tab on the cluster page, find the row that contains the desired service, and click its name in the ***Name*** column.



Switch to configuring a service

3. In the window that opens:
   - Open the ***Primary Configuration*** tab.
   - Switch on the ***Show advanced*** toggle.
   - Fill in configuration parameters of the selected service. The fields highlighted in red are mandatory.
   - Click ***Save***.



Configure a service

Pay attention to the following parameters:

- ***Chrony → NTP servers***. Though this parameter is optional, we recommend to set it to the nearest NTP server, e.g. the one that your company uses.

NOTE

- If you need to install the ADPG Control ADBM Agent component that belongs to the ADPG Control Agents service, make sure that the Enable backups toggle button is disabled. This component is a backup manager, and it is impossible to use both backup methods simultaneously.
- For more information about service configuration parameters, see Configuration parameters in the References.

Configure a cluster

Eugenia Kuzina, Daria Barysheva

Contents

To Table of Contents

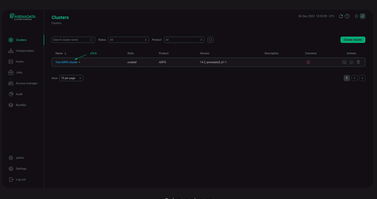
Primary configuration
Configuration groups
Ansible settings

IMPORTANT

There is usually no need to change cluster configuration parameters. You can leave all parameters at the default values.

After creation a new cluster, you can configure it by performing the following steps:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.



Select a cluster

2. Open the ***Configuration*** tab on the cluster page. The ***Configuration*** tab includes the following sections: ***Primary configuration***, ***Configuration groups***, ***Ansible settings***.
3. Fill in all necessary parameters on the selected tab and click ***Save***.

Primary configuration

The ***Configuration → Primary configuration*** tab allows you to configure the general cluster settings.

Available configuration parameters are listed below. Basically, they describe paths to the repositories that are used during the installing process – depending on the cluster operating system (ALT Linux 8, Astra Linux 1.7, CentOS 7, or Ubuntu 22.04). Only the marked repositories are registered on the cluster servers. For each parameter, you can change the default url to the required one.

General configuration parameters

Parameter	Description
Cluster UUID	Unique cluster identifier
Disable firewall	Disables **firewalld** on the cluster hosts

Altlinux 8 repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

Astra linux repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

Centos 7 repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

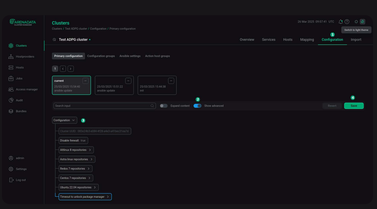
Ubuntu 22.04 repositories

Parameter	Description
Set up ADPG repo	Whether to use the repositories that are described in the `ADPG CORE repo url` and `ADPG repo url` fields
ADPG CORE repo url	The ADPG CORE repository that is required to install ADPG
ADPG repo url	The ADPG repository that is required to install ADPG
Set up ADPG Monitoring repo	Whether to use the repository specified in the `ADPG Monitoring repo url` field
ADPG Monitoring repo url	The ADPG Monitoring repository that is required to install monitoring

Timeout to unlock package manager

The section is available when the ***Show advanced*** toggle button is activated. Parameters of this section are intended to set the timeout of the DNF software package manager unlocking wait. In Ubuntu, after the host reboot, automatic update starts, which locks DNF.

Parameter	Description	Default value
retries	The maximum number of retries	3
delay	The time delay between retries (in seconds)	10



The cluster "Configuration" tab

Configuration groups

The ***Configuration → Configuration groups*** tab is designed to set cluster configuration groups.

Ansible settings

The ***Configuration → Ansible settings*** tab allows you to set Ansible configuration options at the cluster level. The tab is available starting with ADCM 2.2.0.

Ansible settings

Name	Description	Default
forks	The number of parallel processes to spawn when communicating with remote hosts	5

NOTE

During offline installation the *Arenadata repo url* and *Monitoring repo url* parameters are ignored. The data stored locally in the Enterprise Tools cluster is used instead.

Import ET settings

Eugenia Kuzina, Daria Barysheva

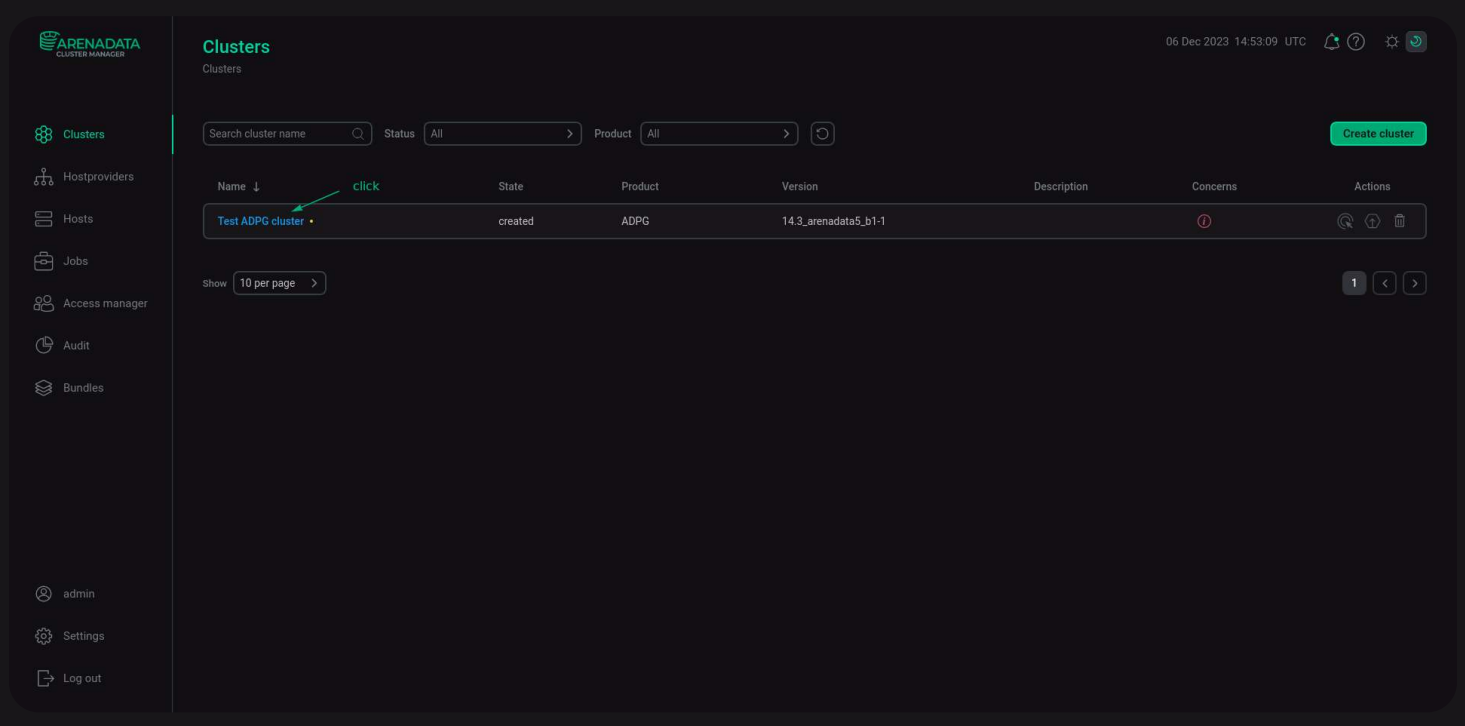
To Table of Contents

IMPORTANT

This step should be performed only in the case of offline installation.

In order to determine that a new cluster should be installed offline, it is necessary to integrate it with the previously created Enterprise Tools (ET) cluster. Follow the steps:

1. Select a cluster on the ***Clusters*** page. To do this, click a cluster name in the ***Name*** column.

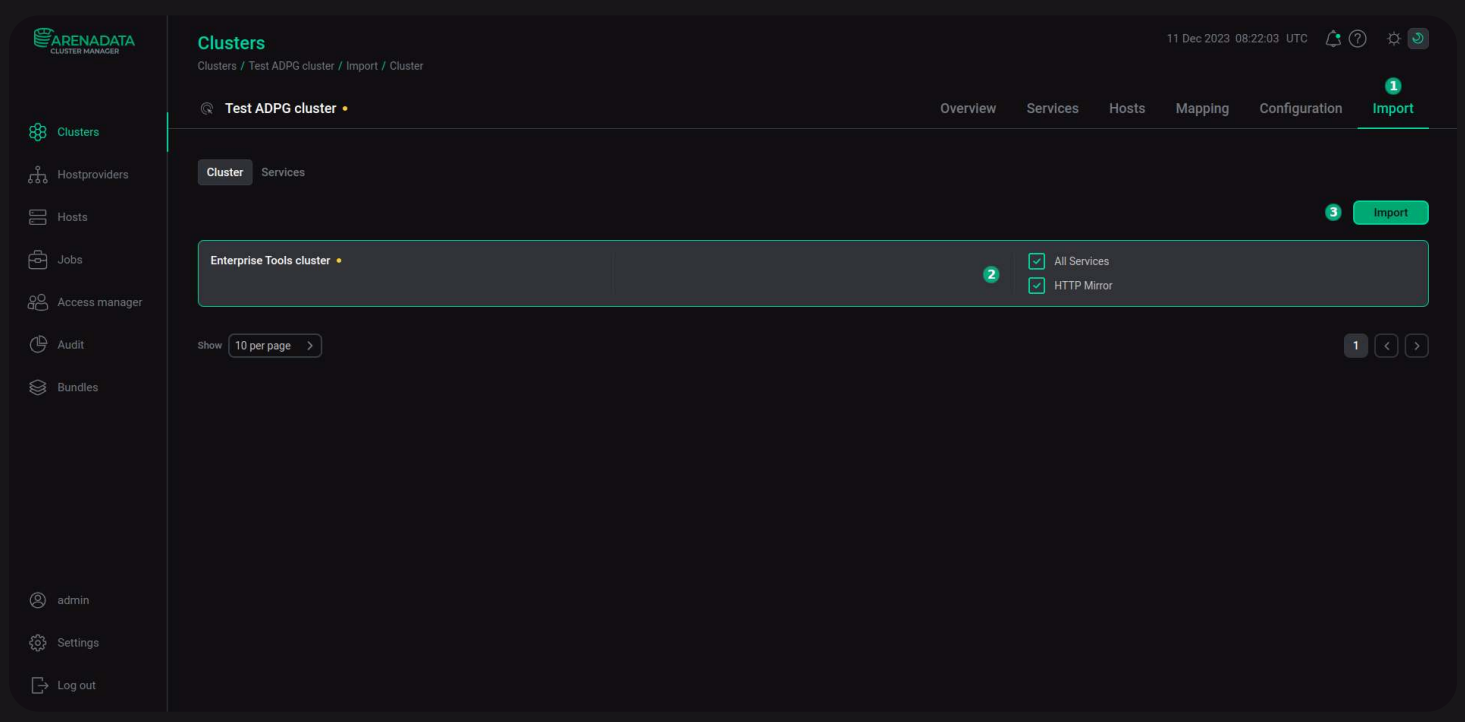


Select a cluster

2. Open the ***Import*** tab in the cluster menu.

3. Select the ***HTTP Mirror*** service of the previously created ET cluster.

4. Click ***Import***.



Integrate with the ET cluster

NOTE

After successful import, the data stored locally in the Enterprise Tools cluster can be used at the cluster installation stage. So there is no need to extract this data via Internet.

Install a cluster

Contents

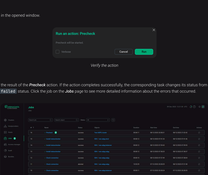
To Table of Contents

Step 1. Precheck
Step 2. Run installation
  Whole cluster
  Single services
Step 3. Check the cluster state after installation
Step 4. Connect to PostgreSQL via psql

Step 1. Precheck

Before installing a new cluster it is recommended to run the **Precheck** action. By doing this, you make sure that all necessary repositories are available, the cluster does not contain hosts with the same IP addresses, and all other prerequisites are met. Do the following:

1. Apply the **Precheck** action to the cluster selected on the **Clusters** page by clicking the icon in the **Actions** column.

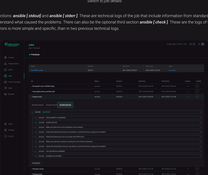


Run the cluster precheck

2. Verify the requested action in the opened window.

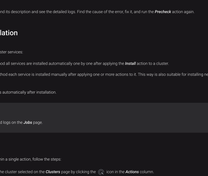
Verify the action

3. Open the **Jobs** page to see the result of the **Precheck** action. If the action completes successfully, the corresponding task changes its status from `running` to `success`. If the job fails, it switches to the `failed` status. Click the job on the **Jobs** page to see more detailed information about the errors that occurred.



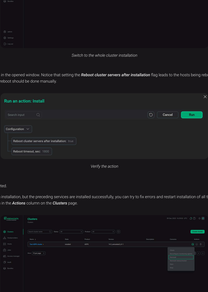
View jobs

The window with job inner steps is opened. Click the ellipsis in the **Details** column for additional information on the current step.



Switch to job details

A job page contains two sections: **ansible [stdout]** and **ansible [stderr]**. These are technical logs of the job that include information from standard I/O streams stdout/stderr. These logs can help to understand what caused the problems. There can also be the optional third section **ansible [check]**. These are the logs of the most frequent errors checks. The description of these errors is more simple and specific, than in two previous technical logs.



Job details

4. If a task fails, you can expand its description and see the detailed logs. Find the cause of the error, fix it, and run the **Precheck** action again.

Step 2. Run installation

There are two ways to install cluster services:

1. **Whole cluster** - in this method all services are installed automatically one by one after applying the **Install** action to a cluster.
2. **Single services** - in this method each service is installed manually after applying one or more actions to it. This way is also suitable for installing new services in the already deployed cluster.

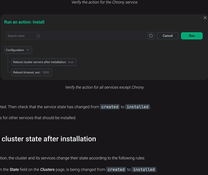
In both cases each service starts automatically after installation.

> **NOTE**
> In case of errors, you can find logs on the **Jobs** page.

Whole cluster

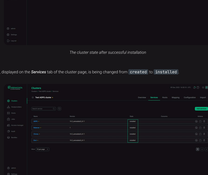
To install all cluster services within a single action, follow the steps:

1. Apply the **Install** action to the cluster selected on the **Clusters** page by clicking the icon in the **Actions** column.



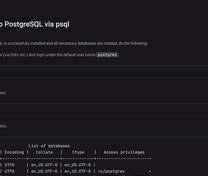
Switch to the whole cluster installation

2. Verify the requested action in the opened window. Notice that setting the **Reboot cluster servers after installation** flag leads to the hosts being reboot after the installation is completed. Otherwise, the reboot should be done manually.



Verify the action

3. Wait until the job is completed.
4. If some service fails during installation, but the preceding services are installed successfully, you can try to fix errors and restart installation of all the failed services. To do this, choose the **Reinstall** action in the **Actions** column on the **Clusters** page.



Installation restart

Single services

To install one or several cluster services manually, add these services to the cluster and then perform the actions described in the following subsections.

> **CAUTION**
> Notice that services are installed one by one. Do not install another service until the installation of the previous one is completed.

1. Open the **Services** tab on the cluster page. For each service that should be installed, click the icon in the **Actions** column and select the **Install** action.

Switch to installation of single services

2. Verify the action in the opened window.

Verify the action for the Chrony service

Verify the action for all services except Chrony

3. Wait until the job is completed. Then check that the service state has changed from `created` to `installed`.
4. Repeat the previous actions for other services that should be installed.

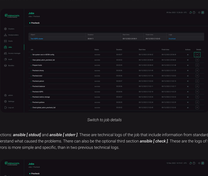
Step 3. Check the cluster state after installation

As a result of successful installation, the cluster and its services change their state according to the following rules:

- The cluster state, displayed in the **State** field on the **Clusters** page, is being changed from `created` to `installed`.

The cluster state after successful installation

- The state of cluster services, displayed on the **Services** tab of the cluster page, is being changed from `created` to `installed`.



The state of cluster services after successful installation

Step 4. Connect to PostgreSQL via psql

In order to check that PostgreSQL is successfully installed and all necessary databases are created, do the following:

1. Connect to the ADPG Server (via SSH, etc.) and login under the default user name `postgres`:

```
$ sudo su - postgres
```

2. List all PostgreSQL databases:

```
$ psql -l
```

The result should look like this.

```
                                  List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    |   Access privileges
-----------+----------+----------+------------+------------+-----------------------
postgres   | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
template0  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
template1  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
(3 rows)
```

3. Use one of the two commands:

- `psql` - to connect to the default database,
- `psql -d<database_name>` - to connect to the database with the specified name `<database_name>`.

```
$ psql postgres
```

After this you can send queries to the selected database via the `psql` client.

```
postgres=# SELECT 1 AS test;
 test
------
    1
(1 row)
```

Install monitoring

Eugenia Kuzmina

Contents

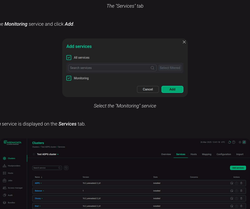
To Table of Contents

Step 1. Add the Monitoring service
Step 2. Configure monitoring
Step 3. Install the Monitoring service
Step 4. View dashboards

ADPG bundle includes a monitoring system. To install monitoring, perform the steps described below.

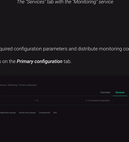
Step 1. Add the Monitoring service

Open the **Clusters** page in ADCM UI and select an ADPG cluster. Open the **Services** tab and click **Add services**.



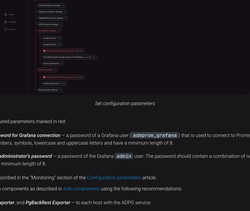
The "Services" tab

In the dialog that opens, select the **Monitoring** service and click **Add**.



Select the "Monitoring" service

As a result, the added **Monitoring** service is displayed on the **Services** tab.

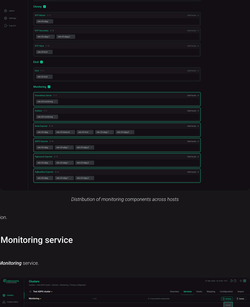


The "Services" tab with the "Monitoring" service

Step 2. Configure monitoring

The **Concerns** icon (!) indicates that you need to specify required configuration parameters and distribute monitoring components across hosts.

Click the **Monitoring** service and set configuration parameters on the **Primary configuration** tab.



Set configuration parameters

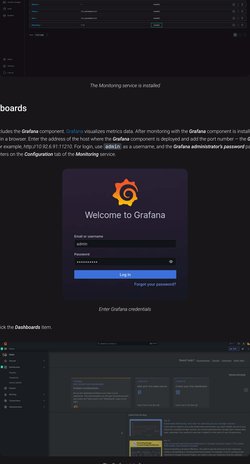
You should set the following required parameters marked in red:

- **Prometheus settings** → **Password for Grafana connection** – a password of a Grafana user `adpgmon_grafana` that is used to connect to Prometheus. The password should contain a combination of numbers, symbols, lowercase and uppercase letters and have a minimum length of 8.
- **Grafana settings** → **Grafana administrator's password** – a password of the Grafana `admin` user. The password should contain a combination of numbers, symbols, lowercase and uppercase letters and have a minimum length of 8.

All monitoring parameters are described in the "Monitoring" section of the Configuration parameters article.

Then open the **Mapping** tab. Map components as described in Add components using the following recommendations:

- *ADPG Exporter*, *Pgbouncer Exporter*, and *PgBackRest Exporter* – to each host with the ADPG service.
- *Prometheus Server* and *Grafana* – to a separate host.
- *Node Exporter* – to each host from which you need to collect system metrics.

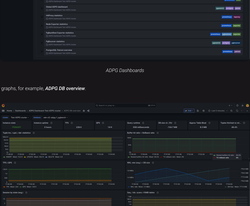


Distribution of monitoring components across hosts

Click **Save** to save the configuration.

Step 3. Install the Monitoring service

Execute the **Install** action of the **Monitoring** service.



The "Install" action of the Monitoring service

In the next dialog window, click **Run** to confirm the action. You can also activate the **Verbose** checkbox to see additional execution details on the **Jobs** page.

Confirm the "Install" action

As a result, the Monitoring service state is changed to `installed`.

The Monitoring service is installed

Step 4. View dashboards

The ADPG monitoring system includes the **Grafana** component. Grafana visualizes metrics data. After monitoring with the **Grafana** component is installed, you will be able to view dashboards that display metrics in a browser. Enter the address of the host where the **Grafana** component is deployed and add the port number – the **Grafana** *listen port* parameter (the default value is `3123`). For example, http://10.92.1.49:11725. For login, use `admin`, as a username, and the **Grafana administrator's password** parameter value as a password. You can find the Grafana parameters on the **Configuration** tab of the **Monitoring** service.

Enter Grafana credentials

Expand the Grafana menu and click the **Dashboards** item.

The Grafana interface

The page with the dashboard list will be displayed. Expand the **ADPG Dashboard...** item for your cluster to display available dashboards.

ADPG Dashboards

Click on any desired item to view graphs, for example, **ADPG DB overview**.

ADPG DB overview graphs

Connect to ADPG

Eugenia Kuzina

To Table of Contents

You can use the following options to connect to the ADPG cluster:

- The standard psql tool
- Third-party client tools:
  - pgAdmin
  - DBeaver
  - others

NOTE

After the ADPG installation, you can use `psql` out of the box. If you need to utilize third-party tools, install and configure them according to your requirements.

Use the standard PostgreSQL tool — psql to connect to ADPG

Eugenia Kuzina

To Table of Contents

The psql tool is a standard PostgreSQL client application that you can use to connect to ADPG. It is an interactive terminal that allows you to run SQL queries and view their results. You can execute queries from the command line or from an SQL file. The `psql` tool is available on each ADPG cluster node.

Run the following commands to start working with `psql`:

```
$ sudo su - postgres
```

```
$ psql
```

The first command switches the shell to a login mode with the access rights for the `postgres` user. The second one runs `psql` for the default database (`postgres`).

You can use the `psql` options from the table below to specify a database name, host, port, and user instead of default values.

psql options

Option	Description
-d <dbname> --dbname=<dbname>	Specifies the name of the database to connect to
-h <hostname> --host=<hostname>	Specifies the host name where the server is located
-p <port> --port=<port>	Specifies the TCP port on which the server listens for connections. The default value is `5432`
-U <username> --username=<username>	Connects to a database as the specified user

Example:

```
$ psql -h 10.92.6.36 -p 5432 -U user1 -d postgres
```

TIP

If you face connection problems, use the pg_isready tool to check the connection status of a PostgreSQL database server.

When the `psql` command is executed, the terminal displays the `psql` prompt with the database name and the `#` character at the end. All subsequent commands should be written after `#`.

```
psql (14.3)
Type "help" for help.

postgres=#
```

You can find examples of SQL queries executed from the command line in the article Example of working with tables via psql.

To run an SQL query from a file, use the meta-command `\i`:

```
\i /path/to/file/filename.sql
```

To logout from `psql`, use the following command:

```
\q
```

The table below lists some meta-commands.

psql meta-commands

Command	Description
`\c <database_name>`	Establishes a new connection to the specified database
`\l`	Lists all the databases on the server and shows their names, owners, character set encodings, and access privileges
`\dt`	Lists all tables in the database
`\dt+`	Lists all tables in the database with its persistence status (`permanent`, `temporary`, or `unlogged`), physical size on disk, and description
`\dn`	Lists all schemas in the database
`\d+ <table_name>`	Shows detailed information about the table
`\dv`	Lists all views in the current database
`\df`	Lists functions with their result data types, argument data types, and function types, which are classified as `agg` (aggregate), `normal`, `procedure`, `trigger`, or `window`
`\x`	Sets expanded table formatting mode
`\du`	Lists database users (roles)

Meta-commands send queries to system tables and views to get the required information. To display these queries, set the `ECHO_HIDDEN` variable to `on`:

```
\set ECHO_HIDDEN on
```

Example:

```
postgres=# \set ECHO_HIDDEN on
postgres=# \dt
********* QUERY **********
SELECT n.nspname as "Schema",
  c.relname as "Name",
  CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHEN 'i' THEN 'index' WHEN 'S' THEN
'sequence' WHEN 's' THEN 'special' WHEN 't' THEN 'TOAST table' WHEN 'f' THEN 'foreign table' WHEN 'p' THEN 'partitioned table'
WHEN 'I' THEN 'partitioned index' END as "Type",
  pg_catalog.pg_get_userbyid(c.relowner) as "Owner"
FROM pg_catalog.pg_class c
     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
     LEFT JOIN pg_catalog.pg_am am ON am.oid = c.relam
WHERE c.relkind IN ('r','p','')
      AND n.nspname <> 'pg_catalog'
      AND n.nspname !~ '^pg_toast'
      AND n.nspname <> 'information_schema'
  AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
**************************

             List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+----------
 public | author          | table | postgres
 public | book            | table | postgres
 public | us_gaz          | table | postgres
 public | us_lex          | table | postgres
 public | us_rules        | table | postgres
 public | users           | table | postgres
(6 rows)
```

To disable displaying queries with the result of a meta-command, set the `ECHO_HIDDEN` variable to `off`:

```
\set ECHO_HIDDEN off
```

Use pgAdmin to connect to ADPG

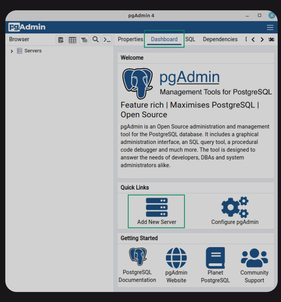
Eugenia Kuzina

To Table of Contents

To connect to the PostgreSQL server, you can use pgAdmin — a third-party client tool that provides a graphical user interface.

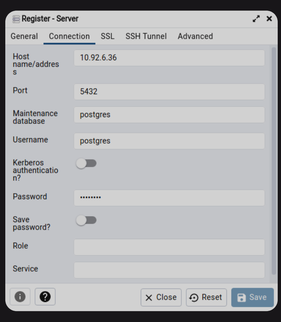
Visit Download for information on how to install pgAdmin.

To connect to an ADPG host, launch pgAdmin, open the ***Dashboard*** tab, and click ***Add New Server*** in the Quick Links section.



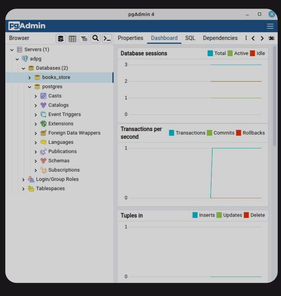
The Dashboard tab in pgAdmin

Specify connection settings according to your environment.



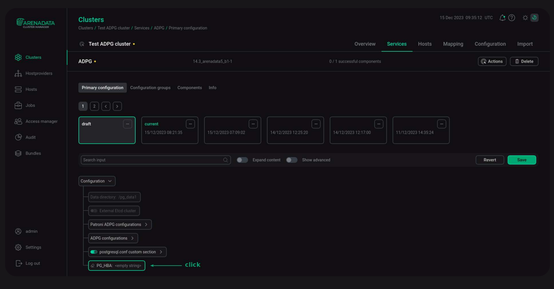
Connection settings in pgAdmin

After the connection is established, pgAdmin displays the specified database in the pgAdmin browser.



The pgAdmin browser window

If you need to change authentication settings from pg_hba.conf for the pgAdmin connection, use ADCM UI. Open the ***Clusters*** page, select an ADPG cluster, and switch to the ***Services*** tab. Select the ***ADPG*** service, click the ***PG_HBA*** option on the ***Primary configuration*** tab, and edit the value.

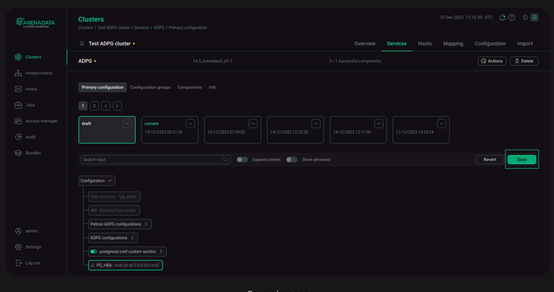


Edit the PG_HBA section

Enter a record of the pg_hba.conf file in the pop-up window and click ***Apply***.

Add a PG_HBA record

To save changes, click ***Save***.



Save changes

Execute the Reconfigure & Restart action to apply changes.

CAUTION

We do not recommend editing *postgresql.conf* via pgAdmin. When the ADPG service executes the Reconfigure & Restart action, *postgresql.conf* is rewritten with settings specified on the ***Primary configuration*** tab of the ADPG service. If you make changes to *postgresql.conf* in any other way, the changes will be lost.

For more information on pgAdmin, refer to the official documentation.

Use DBeaver to connect to ADPG

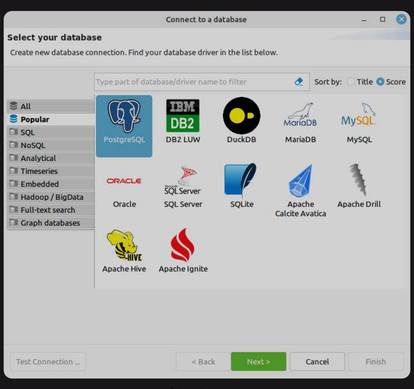
Eugenia Kuzina

To Table of Contents

The third-party client tool DBeaver supports different databases and can be used to connect to a PostgreSQL server.

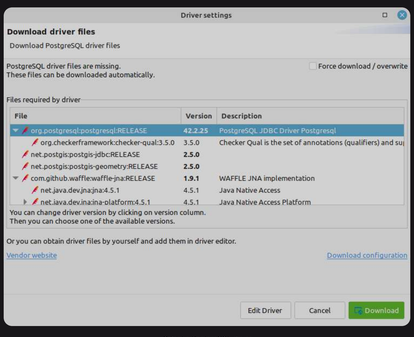
For information on how to install DBeaver, refer to Installation.

To connect to an ADPG host, launch DBeaver and select PostgreSQL in the ***Connect to a database*** window.



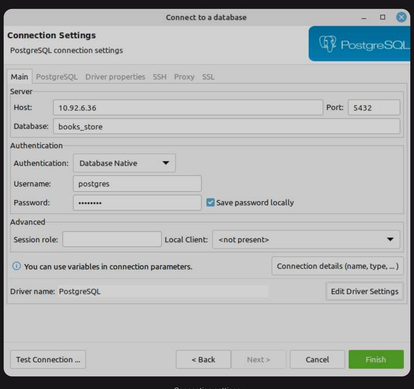
Select a database type

If you connect to a PostgreSQL server for the first time, select a driver.



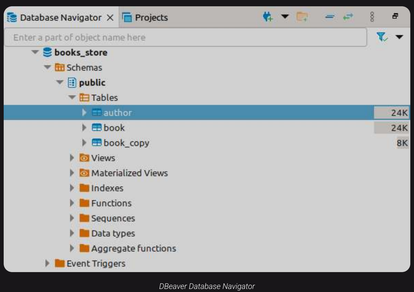
Select a PostgreSQL driver

Define the connection settings in the next window.



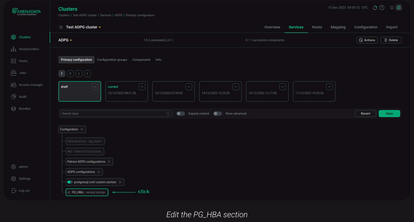
Connection settings

After the connection is established, DBeaver displays the specified database on the ***Database Navigator*** tab.



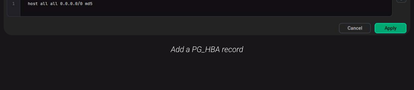
DBeaver Database Navigator

If you need to change authentication settings from pg_hba.conf for the DBeaver connection, use ADCM UI. Open the ***Clusters*** page, select an ADPG cluster, and switch to the ***Services*** tab. Select the ***ADPG*** service, click the ***PG_HBA*** option on the ***Primary configuration*** tab, and edit the value.



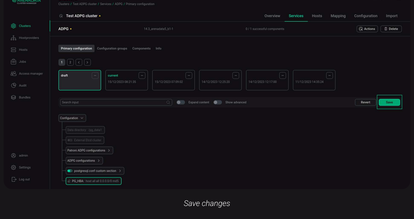
Edit the PG_HBA section

Enter a record of the pg_hba.conf file in the pop-up window and click ***Apply***.



Add a PG_HBA record

To save changes, click ***Save***.



Save changes

Execute the Reconfigure & Restart action to apply changes.

For more information on DBeaver, refer to official documentation.

Authentication in ADPG

Eugenia Kuzina

To Table of Contents

Authentication is the process of verifying the identity of a user who wants to establish the connection with the server. The server determines whether a client application or a user that runs the client application is permitted to connect with the specified user name. ADPG uses roles to manage access privileges. In the PostgreSQL terminology, a user is a role with the LOGIN privilege.

PostgreSQL roles are separate from operating system users. A database server can accept remote connections with users who have no local operating system account.

ADPG supports the following authentication methods: trust, password, GSSAPI, SSPI, ident, peer, LDAP, RADIUS, certificate, PAM, and BSD.

Overview of PG_HBA configuration

Table of Contents

Contents

- PG_HBA format
- Authentication record fields
  - Connection type
  - database
  - user
  - address
  - IP-address and IP-mask
  - auth-method
  - auth-options
- Include external files
- Display pg_hba.conf records
- PG_HBA field

In ADPG, the PG_HBA field is used to configure the client authentication that is controlled by the pg_hba.conf file in PostgreSQL. This field contains authentication records in the same format as pg_hba.conf.

CAUTION

We do not recommend editing the pg_hba.conf file directly. When the ADPG service executes the Reconfigure & Restart action, ADPG will overwrite it with service settings and settings specified in the **PG_HBA** field from the **Primary configuration** tab of the ADPG service.

PG_HBA format

PG_HBA is supported in a similar set of records with an authentication rules, one per line. A record consists of fields separated by spaces or tabs. A field can contain white spaces if the field value is quoted. Blank lines and any text after the **#** comment character are ignored. To continue a record on the next line, end the line with a backslash (****).

The **PG_HBA** records are processed from top to bottom. ADPG determines whether a requested connection (based on a connection type, database name, user and IP address) matches the first record that it finds. If it matches, then the client is authenticated using the method that is specified. If the record is successful, the connection is allowed, otherwise it is permitted. Subsequent records are not considered. A reject method will deny the connection. If no record is found, the connection is denied.

A record can have multiple formats as shown in the example.

```
local           database  user      auth-method [auth-options]
host            database  user      address     auth-method [auth-options]
hostssl         database  user      address     auth-method [auth-options]
hostnossl       database  user      address     auth-method [auth-options]
hostgssenc      database  user      address     auth-method [auth-options]
hostnogssenc    database  user      address     auth-method [auth-options]
host            database  user      IP-address  IP-mask     auth-method [auth-options]
hostssl         database  user      IP-address  IP-mask     auth-method [auth-options]
hostnossl       database  user      IP-address  IP-mask     auth-method [auth-options]
hostgssenc      database  user      IP-address  IP-mask     auth-method [auth-options]
hostnogssenc    database  user      IP-address  IP-mask     auth-method [auth-options]

include         file
include_if_exists file
include_dir     directory
```

For the **database** and **user** fields, you can specify a separate file by prepending the file name with **@**. The file may contain database names or users, respectively. Files included by the **@** construct are read as a list of names, which can be separated by spaces or commas. The hostname **all** matches any address. To match a username to a file, use the **@** symbol, unless the file name following **@** is an abbreviation, in which a failure to be related to the directory containing the referencing file. When using **@** in the **PG_HBA field**, the path or file name to the file is interpreted relative to the location of the configuration parameter. The default value is **hba_file('adpgdir')**.

The record fields are described in detail in the following sections.

Authentication record fields

This section describes the authentication record fields. The number of fields may vary depending on the record type.

Connection type

Except for records containing **include**, the first field of a record determines a connection type. The possible values are listed in the table below.

Name	Description
local	Matches connections using Unix-domain sockets. Without a record of this type, Unix-domain socket connections are not allowed.
host	Manages connections made over TCP/IP. The **host** entries correspond to different connection types. This record matches an SSL or non-SSL connection as well as GSSAPI encrypted or non-GSSAPI encrypted connection.
hostssl	Matches connections via TCP/IP when the connection uses the SSL encryption. The SSL support is enabled in ADPG during its build. To use **hostssl**, SSL must be enabled by setting the **SSL** configuration parameter as described in Health SSL encryption. Otherwise, the **hostssl** entry is ignored, except to log a warning that it cannot match any connection.
hostnossl	Matches connections made over TCP/IP that do not use SSL.
hostgssenc	Matches connections made over TCP/IP that can be GSSAPI encrypted.
hostnogssenc	Matches connections made over TCP/IP that do not use the GSSAPI encryption

database

The **database** field specifies which database the record matches. This field accepts the following values:

- The name of a specific database or a regular expression defining database names. You can specify multiple regular expressions and/or database names separated by commas. **/**(*) is the regex pattern. Make it the only value for the **all** field. Quote a regular expression with spaces. To continue a record on the next line, end the line with a backslash (****).
- **all** – all databases.
- **sameuser** – the record matches if the requested database has the same name as the requested user. For example, if the user is **jsmithpg**, they can only connect to the **jsmithpg** database.
- **samerole** – specifies that a user must be a member of the role with the same name as a database. **samegroup** is an obsolete name for the **samerole** option that is allowed. If a database name is **jsmithpgrp**, only users who are members of the **jsmithpgrp** role can connect to it. Superusers are considered to meet **samerole** only when they explicitly belong to such a role.
- **replication** – specifies that the record matches if a physical replication connection is requested. However, it does not match with logical replication connections.
- A separate file containing database names and/or regular expressions for database names, preceding the file name with **@**.

For example, the configuration below allows **jsmith** to connect to the **jsmpg** and **authdb** databases:

```
# type  database  user      address         auth-method
host    jsmpg,authdb  jsmith    0.0.0.0/0       md5
```

user

The field specifies which database user names (roles) this record matches. This field accepts the following values:

- A name of a specific role (user name) or a regular expression starting with a slash (**/**) that determines a role name. You can add multiple roles and/or regular expressions separated by commas. For example, the regular expression **/^.*support$** allows connections for users whose names end with **support**.
- **all** – matches any role.
- A separate file containing user names and/or regular expressions. It can be specified by prepending the file name with **@**.

The configuration below allows all members of the **supportt** role to connect to the **acctdata** database:

```
# type  database  user      address         auth-method
host    access_stat   +supportt   0.0.0.0/0       md5
```

address

This field specifies the client host addresses that this record matches. This field can contain either a host name, an IP address range, or one of the special keywords. The following values are allowed:

- An IP address range specified using CIDR notation.
- **all** – any IP address.
- **samehost** – any server's IP addresses.
- **samenet** – any address in a subnet that the server is directly connected to.

A host name, like **localhost**, is one of IP address over which a given hostname is treated as a host name. A host name that is used in this field should be the one that address-to-name resolution of the client's IP address returns, otherwise the line won't be matched. Some hostname databases allow associating an IP address with multiple multi-host names, but the operating system only returns one hostname when asked about an IP address. A host name specification that starts with a dot (**.**) matches a suffix of the actual host name. So **.example.com** would match **some.example.com** but not only **example.com**.

The records that specify an IP address can be used to connect to any database with any database user name.

Use local loopback TCP/IP connections:

```
# type  database  user      address         auth-method
host    all       all       127.0.0.1/32    trust

#or IPv6
#
# type  database  user      address         auth-method
host    all       all       ::1/128         trust
```

Use a host name (it typically covers both IPv4 and IPv6):

```
# type  database  user      address         auth-method
host    all       all       localhost       trust
```

Allow any user from any host with IP address **192.168.54.0/24** to connect:

```
# type  database  user      address         auth-method
host    all       all       192.168.54.0/24 trust
```

IP-address and IP-mask

These two fields can be used as an alternative to the IP address/mask length CIDR notation instead of specifying the mask length, the actual mask is set in a separate column. For example, **255.0.0.0** represents an IPv4 CIDR mask length of 8, and **255.255.255.255** denotes an IPv4 CIDR mask length of 32.

These fields are only applied to the records that are specified in the IP address format.

The configuration below allows any user on the local system to connect to any database with any database user name:

```
# type  database  user      address         auth-method
local   all       all                       trust
```

auth-method

The **auth-method** field specifies the authentication method to use when a connection matches the record. All the options are lower-case and treated case sensitively, so even acronyms like **TLS**[] must be specified as lower case. For more information on authentication methods, see Authentication methods.

Value	Description
trust	Allows the connection unconditionally. This method allows anyone who can connect to the ADPG/PostgreSQL database server, to log in as any PostgreSQL user without any authentication. See Trust authentication.
reject	Rejects the connection unconditionally. This is useful for filtering certain hosts from a group. For example, you can block a specific host from connecting, and then specify later records to allow remaining hosts in a specific network to connect.
scram-sha-256	Performs the **SCRAM-SHA-256** authentication to verify the user password. See Password authentication.
md5	Performs the **SCRAM-SHA-256** or **MD5** authentication to verify a user's password. See Password authentication.
password	Requires the client to supply an unencrypted password for authentication. Since the password is sent in clear text over the network, this should not be used on untrusted networks. See Password authentication.
gss	Uses GSSAPI to authenticate the user. This is only available for Windows clients. See GSS Authentication.
sspi	Uses SSPI to authenticate the user. This is only available for Windows clients. See SSPI Authentication.
ident	Obtains a user name of the client operating system by contacting an ident server on the client. The system checks if it matches the requested database user name. The **ident** authentication can only be used on TCP/IP connections. When specified for local connections, the **peer** authentication will be used instead. See Ident Authentication.
peer	Obtains the client's operating system user name and checks if it matches the requested database user name. This is only available for local connections. See Peer authentication.
ldap	Authenticates using an LDAP server. See Kerberos authentication.
radius	Authenticates using a RADIUS server. See RADIUS Authentication.
cert	Authenticates using SSL client certificates. See Enable SSL encryption.
pam	Authenticates using the Pluggable Authentication Modules (PAM) service provided by the operating system. See PAM Authentication.
bsd	Authenticates using the BSD Authentication service provided by the operating system. See BSD Authentication.

The **trust** method does not perform authentication. It only makes sense to use **trust** for local connections. The **reject** method unconditionally denies users. You can use it to prioritize the records that require a specific type of connection and reject it. For example, you can prohibit users (groups) connections:

```
# type  database  user      address         auth-method
hostssl all       +devgrp   0.0.0.0/0       reject
hostssl all       +qagrp    0.0.0.0/0       reject
hostssl all       all       0.0.0.0/0       cert
```

The **peer**, **gss**, and **sspi** methods provide external authentication. As a result of successful authentication, ADPG/PostgreSQL receives the same specified during authentication PostgreSQL name, the name identifies the external software. Use the **map** field to specify an additional parameter (**map** = *mapname*). The **map** option defines the rules for matching internal and external names.

The following configuration allows all users to connect to all databases using the SSL encryption and SSL client certificates:

```
# type  database  user      address         auth-method
hostssl all       all       0.0.0.0/0       cert
```

auth-options

The **auth-options** field specifies options for the given auth-method. Options should be added in the following format: **<name>=<value>**.

In addition to the method-specific options listed below, there is one **map** option available to all authentication methods. The **map** option is supported stored in the SYSID or in an external service.

The **peer**, **gss**, and **sspi** methods provide external authentication. As a result of successful authentication, ADPG/PostgreSQL receives the same specified during authentication PostgreSQL name, the name identifies the external software. Use the **map** field to specify an additional parameter (**map** = *mapname*). The **map** option defines the rules for matching internal and external names.

The following configuration allows all users to connect to all databases using the SSL encryption and SSL client certificates:

```
# type  database  user      address         auth-method
hostssl all       all       0.0.0.0/0       cert
```

auth-options

The **auth-options** field specifies options for the given auth-method. Options should be added in the following format: **<name>=<value>**.

In addition to the method-specific options listed below, there is one **map** option available to all authentication methods. The **map** option is supported stored in the SYSID or in an external service.

- **ldapurl** – requires the client to provide a valid Uniform[SSL certificate.
- **verify-full** – requires the client must provide a valid (trusted) SSL certificate and **CN** = *Common Name* in the certificate must match the user name or an applicable mapping.

This option works similarly to the **cert** authentication method with the verification of client certificates to be associated with any authentication method that supports **hostssl** records.

In any record that includes client certificate authentication (a record with the **cert** authentication method or a **clientcert** parameter), the optional **clientcome** parameter specifies which credential element in the client certificate should be matched during verification. This parameter can have one of two values:

- **CN** – the value must match against the **Common Name** (**CN**) field in the default value.
- **DN** – the value must match against the **Distinguished Name** (**DN**) field.

The **map** option is specified for methods that use name mapping. It defines the rules for matching internal and external names. To set the **map** option, you need to add the rules to the **ident_file** (**ident_file=/adpgdir/ident_conf**) configuration parameter dimensions its location. The default value is **hba_file('adpgdir')**. For example:

```
# pg_ident.conf
# MAPNAME       SYSTEM-USERNAME         ADPG-USERNAME
  omicron       ann                     admin_db

# type  database  user      address         auth-method  auth-options
host    all       admin_db  192.168.12.10/32  ident        map=omap_admin
```

Include external files

The records listed in the table below allow you to include the content of external files in the **PG_HBA** field.

Name	Description
include	The record will be replaced with the contents of the specified file
include_if_exists	The record will be replaced with the content of the given file if the file exists. Otherwise, a message is logged to indicate that the file has been skipped
include_dir	The record will be replaced by the contents of all the files found in the directory if they do not start with **.** and end with **.conf** processed in alphabetical order

Example:

```
include file1
```

Display pg_hba.conf records

You can use the pg_hba_file_rules view to display a summary of the pg_hba.conf contents. A row appears in this view for each non-empty, non-comment line in pg_hba.conf with an annotation indicating whether the rule could be applied successfully.

Note that this view reports the current contents of the file, not the content that was last loaded by the server. A new reflecting an incorrect entry will have values only for the **line_number** and **error** fields until the file is fixed and pg_hba.conf content is validated during the Reconfigure & Restart action. After the validation fails, the action will be terminated with an error and ADPG will not load the current content of pg_hba.conf.

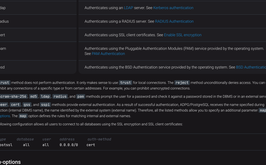
```
SELECT * FROM pg_hba_file_rules;
```

A possible result:

```
 rule_number |  file_name   | line_number | type |  database  |  user_name  |  address  |
netmask | auth_method | options | error
-------------+--------------+-------------+------+------------+-------------+-----------+
--------+-------------+---------+-------
           1 | /pg/data/adpg/pg_hba.conf |           2 | local | {all}      | {all}       |
       | peer |         |
           2 | /pg/data/adpg/pg_hba.conf |           4 | local | {replication} | {pg_replicator} |
       |      | trust   |         |
           3 | /pg/data/adpg/pg_hba.conf |           5 | local | {replication} | {pg_replicator} |
       |      | scram-sha-256 |         |
           4 | /pg/data/adpg/pg_hba.conf |           6 | host  | {all}      | {pg_replicator} | 10.92.40.185 | 255.255.
255.255 | scram-sha-256 |         |
           5 | /pg/data/adpg/pg_hba.conf |           7 | host  | {replication} | {pg_replicator} | 10.92.40.185 | 255.255.
255.255 | scram-sha-256 |         |
           6 | /pg/data/adpg/pg_hba.conf |           8 | host  | {all}      | {pg_rewind} | 10.92.40.185 | 255.255.
255.255 | scram-sha-256 |         |
           7 | /pg/data/adpg/pg_hba.conf |           9 | host  | {replication} | {pg_rewind} | 10.92.40.214 | 255.255.
255.255 | scram-sha-256 |         |
           8 | /pg/data/adpg/pg_hba.conf |          10 | host  | {all}      | {pg_rewind} | 10.92.40.214 | 255.255.
255.255 | scram-sha-256 |         |
           9 | /pg/data/adpg/pg_hba.conf |          11 | host  | {replication} | {pg_rewind} | 10.92.40.214 | 255.255.
255.255 | scram-sha-256 |         |
          10 | /pg/data/adpg/pg_hba.conf |          12 | host  | {all}      | {pg_rewind} | 10.92.40.214 | 255.255.
255.255 | scram-sha-256 |         |
          11 | /pg/data/adpg/pg_hba.conf |          13 | host  | {all}      | {postgres}  | 0.0.0.0      | 0.0.0.0
       | ident   | {map=test} |
          12 | /pg/data/adpg/trust       |           1 | host  | {all}      | {all}       | 0.0.0.0      | 0.0.0.0
       | trust   |         |
(12 rows)
```

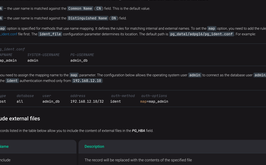
PG_HBA field

To edit the **PG_HBA** field, at 2ndQ UI, open the **Clusters** page, select an ADPG cluster, and switch to the **Services** tab. Select the **ADPG** service. Click the **PG_HBA** field in the **Configuration** tree on the **Primary configuration** tab.



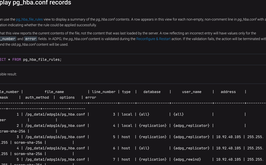
Edit the PG_HBA field

Enter a record in the pop-up window and click **Apply**.



Add a record

To save changes, click **Save**.



Save changes

Execute the Reconfigure & Restart action to apply changes.

Password authentication

Eugenia Kuzina

Contents

To Table of Contents

- Specify the authentication settings
  - The pg_hba.conf file
  - The password_encryption parameter
- Set and change a user password
- View passwords

ADPG supports several password authentication methods that differ in how a password is stored on the server and transmitted over the connection. The table below contains supported methods.

Password authentication methods

Name	Description
scram-sha-256	Performs the `scram-sha-256` authentication according to RFC 7677. This is a challenge-response scheme that prevents password sniffing on untrusted connections. Passwords are stored on the server in a cryptographically hashed form. `scram-sha-256` is the most secure method, but older client libraries do not support it
md5	Uses a custom secure challenge-response mechanism. It prevents password sniffing and stores passwords on the server in a hashed form but provides no protection if the password hash is stolen from the server. The md5 hash algorithm is not secure against collision attacks. The `md5` method does not support the db_user_namespace feature
password	Sends the password in the clear text format. This method is vulnerable to password sniffing attacks. You can use this method if the connection is protected by SSL encryption

Specify the authentication settings

The following options control the password authentication settings:

- The pg_hba.conf file defines client authentication settings.
- The password_encryption parameter specifies the algorithm used to encrypt user passwords.

The pg_hba.conf file

The *pg_hba.conf* file contains a set of records, one per line. Refer to the following link for the detailed field description: Overview of PG_HBA configuration.

Use the ADCM UI to edit *pg_hba.conf*. To do this, open the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** tab and add lines to the ***PG_HBA*** field. For example, the following record allows a connection established over TCP/IP to all databases, by all users, from all IPv4 addresses with the `scram-sha-256` authentication method:

```
host  all  all 0.0.0.0/0 scram-sha-256
```

Click ***Save*** on the ***Configuration*** tab and execute the Reconfigure & Restart action to apply changes. The following lines are added to *pg_data1/adpg16/pg_hba.conf*.

```
# BEGIN Customs from ADCM
host  all  all 0.0.0.0/0 scram-sha-256
# END Customs from ADCM
```

The password_encryption parameter

If you set a password in the CREATE ROLE or ALTER ROLE SQL command, the **password_encryption** parameter determines the password encryption algorithm. Possible values are `scram-sha-256` (the default value) and `md5`.

If `md5` is specified as an authentication method in *pg_hba.conf* but the user's password on the server is encrypted with the `scram-sha-256` algorithm, the `scram-sha-256` authentication is used as the most secure method.

You can change the `password_encryption` parameter via ADCM UI. To do this, open the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** tab, expand the ***ADPG configurations*** section, and set a new value for `password_encryption` in the ***postgresql.conf*** field (see Configuration parameters). For example, the following line sets the `md5` algorithm for the password encryption:

```
password_encryption=md5
```

You can also execute the following SQL statement to change the `password_encryption` parameter for the current session:

```
SET password_encryption = 'scram-sha-256';
```

Set and change a user password

If you do not set a password for a user, the user's password is NULL and the password authentication fails for this user.

Specify the `PASSWORD` parameter of the CREATE ROLE SQL command to set the password for a new user role.

```
CREATE ROLE user1 WITH LOGIN PASSWORD 'password1';
```

To change the password for an existing role, execute the ALTER ROLE command.

```
ALTER ROLE user1 WITH PASSWORD 'password2';
```

The `\password` psql meta-command also changes the password of the current role.

```
\password
```

After the `\password` command is executed, the command prompt asks you to enter a new password twice:

```
Enter new password for user "postgres":
Enter it again:
```

If the entered passwords match, the current password is changed.

View passwords

The password hash for each database user is stored in the pg_authid catalog. You can run the following SQL command to view roles and passwords:

```
SELECT rolname, rolpassword FROM pg_authid;
```

The command output:

```
         rolname          |                          rolpassword
--------------------------+-------------------------------------------------------
 pg_database_owner        |
 pg_read_all_data         |
 pg_write_all_data        |
 pg_monitor               |
 pg_read_all_settings     |
 pg_read_all_stats        |
 pg_stat_scan_tables      |
 pg_read_server_files     |
 pg_write_server_files    |
 pg_execute_server_program |
 pg_signal_backend        |
 postgres                 | SCRAM-SHA-256$4096:2yHlw+fjx7PWll0dScESNA==$SETiIdi
xLSGgtE/EEk/5kfRBsIpu+09XuCDPWIFmAuE=:Fyy6OPtQmm1+Ms8ryqww47NT1YhLJ6/zik4iskkm3M
o=
 user_md5                 | md5ea4a7de3c817a0c8cc2670dab9aecc47
(13 rows)
```

An encrypted password starts with the method name. In this example, the `postgres` role password is encrypted using the `scram-sha-256` method and the `user_md5` role password is encrypted with the `md5` algorithm.

Kerberos authentication

© Evgenii Kusina

AD.PG

Contents

To Table of Contents

Install the FreeIPA server

Install the FreeIPA client

Configure ADPG for Kerberos

Install the openldap-clients package

Configure ADPG to use LDAP

Kerberos is a protocol for authentication service requests between trusted hosts across an untrusted network. Kerberos manages credentials (authentication), while LDAP holds authoritative information about user accounts and their access rights (authorization).

We create a secure connection between two hosts (OS – CentOS):

- 10.92.6.36, adpg-host, adpg-host.domain.local;
- 10.92.6.201, freeipa-host, freeipa-host.domain.local.

ADPG is installed on adpg-host.

Install the FreeIPA server

We use Docker CE to deploy the FreeIPA server on freeipa-host. Follow the steps below to install FreeIPA.

1. Execute the command below to add host names and their IP addresses to the hosts file:

```
$ sudo tee -a /etc/hosts > /dev/null <<EOT
10.92.6.36  adpg-host.domain.local adpg-host
10.92.6.201 freeipa-host.domain.local freeipa-host
EOT
```

2. Use the hostnamectl tool to set the hostname:

```
$ sudo hostnamectl set-hostname freeipa-host.domain.local
```

3. Execute the following commands to disable SELinux:

```
$ sudo setenforce 0
$ sudo sed -i 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/selinux/config
$ sudo reboot
```

4. Install Docker CE:

```
$ sudo -i
$ yum remove docker \
              docker-client \
              docker-client-latest \
              docker-common \
              docker-latest \
              docker-latest-logrotate \
              docker-logrotate \
              docker-engine
$ yum install -y yum-utils
$ yum-config-manager \
    --add-repo \
    https://download.docker.com/linux/centos/docker-ce.repo
$ yum install docker-ce docker-ce-cli containerd.io docker-compose-plugin
$ systemctl start docker
$ systemctl enable docker
```

5. Create the ipa-data directory for the FreeIPA configuration files, specify FreeIPA installation options, and write them to the ipa-server-install-options file:

```
$ mkdir /var/lib/ipa-data
$ tee -a /var/lib/ipa-data/ipa-server-install-options > /dev/null <<EOT
--realm=DOMAIN.LOCAL
--ds-password=freeipaDSpass
--admin-password=freeipaADMINpass
EOT
```

6. Install the FreeIPA server:

```
$ docker run -it -h freeipa-host.domain.local --read-only \
    -v /run/fs/cgroup:/run/fs/cgroup:rw \
    --name freeipa \
    --network host \
    -v /var/lib/ipa-data:/data:z -v /etc/hosts:/etc/hosts \
    -e PASSWORD=Password123 \
    freeipa/freeipa-server:centos-7 ipa-server-install -U -r DOMAIN.LOCAL --no-ntp
```

Install the FreeIPA client

Install the FreeIPA client on adpg-host.

1. Add host names and their IP addresses to the hosts file:

```
$ sudo tee -a /etc/hosts > /dev/null <<EOT
10.92.6.36  adpg-host.domain.local adpg-host
10.92.6.201 freeipa-host.domain.local freeipa-host
EOT
```

2. Use the hostnamectl tool to set the hostname:

```
$ sudo hostnamectl set-hostname adpg-host.domain.local
```

3. Execute the following commands to disable SELinux:

```
$ sudo setenforce 0
$ sudo sed -i 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/selinux/config
$ sudo reboot
```

4. Install the FreeIPA client:

```
$ sudo -i

$ yum install freeipa-client
```

5. Utilize the ipa-client-install command to configure adpg-host to use IPA:

```
$ ipa-client-install --mkhomedir --domain=domain.local --server=adpg-host.domain.local
```

The terminal displays the following notification:

```
Autodiscovery of servers for failover cannot work with this configuration.
If you proceed with the installation, services will be configured to always access the discovered server for all operations and
will not fail over to other servers in case of failure.
Proceed with fixed values and no DNS discovery? [no]:
```

Type `yes` to continue with fixed values. The fixed values are displayed:

```
Client hostname: adpg-host.domain.local
Realm: DOMAIN.LOCAL
DNS Domain: domain.local
IPA Server: freeipa-host.domain.local
BaseDN: dc=domain,dc=local

Continue to configure the system with these values? [no]:
```

Type `yes` to confirm the settings.

Specify `admin` as the authorized user:

```
User authorized to enroll computers: admin
```

Enter the password that is passed as the `PASSWORD` parameter of the `docker run` command (in our example – `Password123`).

6. Use the `klist` command to obtain a new Kerberos ticket. Execute the `klist` command to check if a Kerberos ticket is created:

```
$ kinit admin
$ klist
```

7. Add a user and set their password:

```
$ ipa user-add pguser --first=pguser --last=PG --password
```

The result:

```
----------------------
Added user "pguser"
----------------------
  User login: pguser
  First name: pguser
  Last name: PG
  Full name: pguser PG
  Display name: pguser PG
  Initials: pP
  Home directory: /home/pguser
  GECOS: pguser PG
  Login shell: /bin/sh
  Principal name: pguser@DOMAIN.LOCAL
  Principal alias: pguser@DOMAIN.LOCAL
  User password expiration: 20220804154641Z
  Email address: pguser@domain.local
  UID: 139160001
  GID: 139160001
  Password: True
  Member of groups: ipausers
  Kerberos keys available: True
```

8. Services require **Service Principal Name (SPN)** to perform Kerberos authentication. Register the postgres service to obtain SPN:

```
$ ipa service-add postgres/adpg-host.domain.local@DOMAIN.LOCAL --force
```

The result:

```
-------------------------------------------------------------------
Added service "postgres/adpg-host.domain.local@DOMAIN.LOCAL"
-------------------------------------------------------------------
  Principal name: postgres/adpg-host.domain.local@DOMAIN.LOCAL
  Principal alias: postgres/adpg-host.domain.local@DOMAIN.LOCAL
  Managed by: adpg-host.domain.local
```

Configure ADPG for Kerberos

Create a keytab on adpg-host:

```
$ ipa-getkeytab --principal=postgres/adpg-host.domain.local@DOMAIN.LOCAL --keytab=/etc/krb5.keytab
```

Use the chown command to change the owner and the group of the krb5.keytab file:

```
$ chown postgres:postgres /etc/krb5.keytab
```

Utilize ADCM to change the ADPG configuration settings. Open the ***Clusters*** → ***ADPG cluster*** → ***Services*** → ***ADPG*** → ***Primary configuration*** tab, expand the **ADPG configurations** section, open the ***postgresql.conf*** field, and set `listen_addresses` to `*`.

Also, add the following line to ***postgresql.conf***:

```
krb_server_keyfile = '/etc/krb5.keytab'
```

Add the following line to the ***PG_HBA*** section:

```
host all pguser 0.0.0.0/0 gss  include_realm=0 krb_realm=DOMAIN.LOCAL
```

Click the **Save** button and execute the Reconfigure & Restart action to apply changes.

Create a user on adpg-host:

```
$ sudo su - postgres
$ psql -c "CREATE USER pguser"
```

Test the GSSAPI-encrypted connection:

```
$ kinit pguser
$ psql -h adpg-host.domain.local -U pguser postgres
```

The result:

```
psql (14.3)
GSSAPI-encrypted connection
Type "help" for help.

postgres=#
```

Install the openldap-clients package

The openldap-clients package contains command line tools to view and modify directories on an *LDAP server*. Install openldap-clients on *adpg-host*:

```
$ yum install openldap-clients -y
```

Execute the `klist` command to check if a current system user obtains a Kerberos ticket with IPA administrative privilege. If not, run the command `kinit admin`:

```
$ klist
$ kinit admin
```

Add a user:

```
$ ipa user-add pgldap --first=pgldap --last=PG --password
```

Test the connection:

```
$ ldapsearch -x -h freeipa-host.domain.local -b dc=domain,dc=local uid=pgldap
```

Configure ADPG to use LDAP

Create the `pgldap` user:

```
$ sudo su - postgres
$ psql -c "CREATE USER pgldap"
```

Add the following line to the ***PG_HBA*** section:

```
host all pgldap 0.0.0.0/0 ldap  ldapserver=freeipa-host.domain.local ldapbasedn="cn=users,cn=compat,dc=domain,dc=local"
ldapsearchattribute=uid
```

Click **Save** and execute the Reconfigure & Restart action to apply changes.

Test the GSSAPI-encrypted connection:

```
$ kinit pgldap
$ psql -h adpg-host.domain.local -U pgldap postgres
```

The result:

```
psql (14.3)
GSSAPI-encrypted connection
Type "help" for help.

postgres=#
```

LDAP authentication

Evgenia Kuzina

AD.PG

Contents

To Table of Contents

Overview of LDAP authentication
- Simple bind mode
- Search + bind mode

Set up a FreeIPA server

Configure ADPG to use LDAP.

Examples
- FreeIPA server
- MS Active Directory

Overview of LDAP authentication

You can use LDAP authentication to access an ADPG cluster. This authentication method operates similarly to Password authentication, except that it uses LDAP as the password verification method. To validate the user name/password pairs, ADPG connects to an LDAP server, such as Active Directory or 389 Directory Server. Note that the user must already exist in the PostgreSQL database before LDAP can be used for authentication.

LDAP authentication can operate in two modes:

- simple bind mode,
- search + bind mode.

The tables below list configuration options grouped by bind modes. All these options should be specified in the `auth-options` field of PG_HBA. See Configure ADPG to use LDAP.

Parameters used in both modes

Parameter name	Description
ldapserver	Names or IP addresses of LDAP servers to connect to. You can specify multiple servers separated by spaces
ldapport	Port number on LDAP server to connect to. If the port is not specified, the LDAP library's default port setting will be used (389)
ldapscheme	Set this parameter to `ldaps` to use LDAPS (LDAP over TLS/SSL) supported by some LDAP server implementations. See the `ldaptls` parameter for an alternative
ldaptls	Set the `ldaptls` value to `1` to make the connection between ADPG/PostgreSQL and the LDAP server use TLS encryption. This approach utilizes the StartTLS operation described in RFC 4513

The `ldapscheme` and `ldaptls` settings encrypt only the traffic between the ADPG/PostgreSQL and LDAP servers. The connection between the ADPG/PostgreSQL server and ADPG/PostgreSQL client is unencrypted.

Simple bind mode

In simple bind mode, the server binds to the distinguished name constructed as `prefix` + `user name` + `suffix`. Typically, the prefix is used to specify `cn=` or `DOMAIN\`, in an Active Directory environment. The suffix is used to specify the remaining part of DN.

Parameters used in simple bind mode

Parameter name	Description
ldapprefix	Specifies a string that is prepended to the user name when forming the bind DN for authentication
ldapsuffix	Specifies a string that is appended to the user name when forming the bind DN for authentication

Search + bind mode

In this mode, the ADPG/PostgreSQL server binds to the LDAP directory with a fixed user name and password, specified with the `ldapbinddn` and `ldapbindpasswd` parameters, and performs a search for a user. If no user name and password are configured, an anonymous bind will be attempted. The search is performed on the `ldapbasedn` subtree, checking for an exact match between the user name and `ldapsearchattribute` values. Once the search finds a user, the server disconnects and reconnects to the LDAP directory as that user, with the password supplied by the client, to ensure that the account is correct.

Parameters used in search + bind mode

Parameter name	Description
ldapbasedn	Root DN to begin the search for a user
ldapbinddn	User's DN to bind to the directory to perform the search
ldapbindpasswd	User's password to bind to the directory to perform the search
ldapsearchattribute	Attribute to match against the user name during search + bind authentication. If not specified, the `uid` attribute will be used
ldapsearchfilter	The search filter that is used when doing search+bind authentication. `$username` in the filter will be replaced with the user name. This option implements more flexible search filters than `ldapsearchattribute`. For example, `ldapsearchfilter="(|(uid=$username)(mail=$username))"`.
ldapurl	Specifies RFC 4516 LDAP URL. `ldap[s]://host[:port]/basedn[?attribute][?scope][?[filter]]]` `scope` is used to specify the scope of the search. The allowed values are: - `base` — a base object search; - `one` — a one level search; - `sub` — a subtree search.

Set up a FreeIPA server

This section contains instructions on how to set up and configure a FreeIPA server that is used in the examples below.

Perform the following steps to install a FreeIPA server on a host with the CentOS or RHEL operating system.

1. Install the FreeIPA server:

```
$ sudo yum install ipa-server
```

2. The examples below use host names in LDAP authentication settings. If your network does not include a DNS service to resolve host names, you can execute the command below to add host names and their IP addresses to the hosts file:

```
$ sudo tee -a /etc/hosts > /dev/null <<EOT
> 10.92.42.229 adpg.example.com adpg-host
> 10.92.43.11 ipa.example.com ipa-host
> EOT
```

Where:

- `10.92.42.229` — IP address of the host with ADPG;
- `10.92.43.11` — IP address of the host with the FreeIPA server.

3. Configure the FreeIPA server:

```
$ sudo ipa-server-install \
  --unattended \
  --hostname=ipa.example.com \
  --domain=example.com \
  --realm=EXAMPLE.COM \
  --ds-password=pass2345 \
  --admin-password=pass1234
```

4. Use the `ipactl status` command to check the status of services:

```
$ sudo ipactl status
```

The output should show that the `Directory` `krb5kdc` and `kadmin` services are running:

```
Directory Service: RUNNING
krb5kdc Service: RUNNING
kadmin Service: RUNNING
httpd Service: RUNNING
ipa-custodia Service: RUNNING
ntpd Service: RUNNING
pki-tomcatd Service: RUNNING
ipa-otpd Service: RUNNING
ipa: INFO: The ipactl command was successful
```

5. Authenticate as the `admin` user that is created automatically to perform administrative activities. To do this, run the `kinit` command:

```
$ kinit admin
```

When the following prompt is shown, enter the password specified in the `admin-password` option of the FreeIPA server configuration (in this example, `pass1234`):

```
Password for admin@EXAMPLE.COM:
```

6. Use the ipa `user-add` command to create a regular FreeIPA user:

```
$ ipa user-add john \
--first=John \
--last=Smith \
--password
```

Enter a new user's password twice:

```
Password:
Enter Password again to verify:
```

Result:

```
----------------------
Added user "john"
----------------------
  User login: john
  First name: John
  Last name: Smith
  Full name: John Smith
  Display name: John Smith
  Initials: JS
  Home directory: /home/john
  GECOS: John Smith
  Login shell: /bin/sh
  Principal name: john@EXAMPLE.COM
  Principal alias: john@EXAMPLE.COM
  User password expiration: 20250317131108Z
  Email address: john@example.com
  UID: 1414488803
  GID: 1414488803
  Password: True
  Member of groups: ipausers
  Kerberos keys available: True
```

Configure ADPG to use LDAP

If your network does not include a DNS service to resolve host names, execute the command below on the ADPG/PostgreSQL host to add host names and their IP addresses to the hosts file:

```
$ sudo tee -a /etc/hosts > /dev/null <<EOT
> 10.92.42.229 adpg.example.com adpg-host
> 10.92.42.11 ipa.example.com ipa-host
> EOT
```

Where:

- `10.92.42.229` — IP address of the host with ADPG;
- `10.92.43.11` — IP address of the host with the FreeIPA server.

As mentioned above, a user must already exist in the database before LDAP can be used for authentication.

Create a role for LDAP users to utilize it in a record of pg_hba.config format:

```
CREATE ROLE ldap_users LOGIN;
```

Create a role for a specific user in the `ldap_users` group. Note that an ADPG user name should match an LDAP user name.

```
CREATE ROLE john IN GROUP ldap_users;
```

To configure connection options, open the **Clusters** page, select an ADPG cluster, and switch to the **Services** tab. Select the **ADPG** service and click the PG_HBA field on the **Primary configuration** tab in the **Configuration** view. Add a new record in the format of the pg_hba.config file. The following record uses the simple binding mode to connect to the FreeIPA server:

```
host    all         +ldap_users     0.0.0.0/0       ldap        ldapserver=ipa.example.com ldapprefix="uid=" ldapsuffix",cn=users,cn=accounts,dc=example,dc=com"
```

Test the connection using the `psql` tool:

```
$ psql postgres -U john -h 10.92.42.229
```

When the following prompt is shown, enter the password specified for the `john` LDAP user:

```
Password for user john:
```

Examples

Since LDAP often uses commas and spaces to separate the different parts of DN, it is often necessary to use double-quoted parameter values when configuring LDAP options, as shown in the examples below.

FreeIPA server

The configuration below uses the search + bind mode. Since `ldapbinddn` is not specified, ADPG attempts to bind anonymously to the LDAP server and perform a search for a user in the specified base DN (the `ldapbasedn` parameter):

```
host    all     +ldap_users     0.0.0.0/0       ldap        ldapserver=ipa.example.com ldapbasedn="cn=users,cn=accounts,dc=example,dc=com"
```

An example of the same configuration for search + bind mode but written as a URL (the `ldapurl` attribute):

```
host    all     +ldap_users     0.0.0.0/0       ldap ldapurl="ldap://ipa.example.com/cn=users,cn=accounts,dc=example,dc=com?uid?sub"
```

Methods that use anonymous binding are not secure. It is better to specify the `ldapbinddn` and `ldapbindpasswd` options to perform the search as an authorized user:

```
host    all         +ldap_users     0.0.0.0/0       ldap        ldapserver=ipa.example.com ldapbasedn="cn=users,cn=accounts,dc=example,dc=com" ldapbinddn="uid=admin,cn=users,cn=accounts,dc=example,dc=com" ldapbindpasswd="pass1234"
```

MS Active Directory

The code below shows how to configure LDAP authentication in search + bind mode for Microsoft Active Directory:

```
host    all     +ldap_users     0.0.0.0/0       ldap        ldapserver=ad01.example.ic ldapbasedn="OU=Peoples,DC=ad,DC=ranger-test" ldapbinddn="cn=admin,dc=ad,dc=ranger-test" ldapbindpasswd="bind_pass123" ldapsearchattribute="sAMAccountName"
```

Active Directory configuration used in the example	
url	ldap://ad01.example.ic:389
binddn	cn=admin,dc=ad,dc=ranger-test
bind password	bind_pass123
user name attribute	sAMAccountName
searchBase	ou=Peoples,dc=ad,dc=ranger-test
group searchbase	ou=Groups,dc=ad,dc=ranger-test
user searchBase	ou=Peoples,dc=ad,dc=ranger-test

Enable SSL encryption

Contents

Enable SSL via ADCM
Use client certificates
Create self-signed SSL certificates
- Generate certificates for an ADPG cluster
- Generate an SSL certificate for PostgreSQL clients

To Table of Contents

ADPG supports SSL encryption of incoming traffic for all services in an ADPG cluster. To use this feature in an ADPG cluster, it is necessary to obtain certificates and [enable SSL via ADCM](#).

For test purposes, you can use self-signed certificates. Then creation is described in the [Create self-signed SSL certificates](#) section.

When SSL is enabled, all services of an ADPG cluster use SSL encryption and the HTTP protocol instead of HTTP.

> **important**
>
> - It is strongly recommended to use certificates signed by a trusted CA. This also applies to self-signed certificates if they are used in production environments.
> - It is necessary to generate SSL certificates with SAN (Subject Alternative Names). Short host names and fully qualified domain names (FQDN) should be present in the SAN section of certificates. IP addresses are not supported.
> - Keys encrypted with a passphrase are not supported.
> - When upgrading from ADPG v1.5.3 with enabled SSL, it is recommended to switch SSL off, then upgrade to ADPG 1.6.3 and use the **Manage SSL** cluster action to enable SSL.

Enable SSL via ADCM

To enable SSL, follow the steps below:

1. On the **Clusters** page, run the **Manage SSL** action.

2. In the window that opens, switch on the **Enable SSL** toggle button.

The 'Run an action: Manage SSL' window

SSL settings are displayed below the **Enable SSL** toggle button. Parameters highlighted in red are required.

SSL parameters

3. Specify SSL settings according to your environment. All paths to certificates and keys should be specified as absolute. It is not recommended to save certificate and key files in the ADPG data directory (the default path is /pgdata/) because the data directory can be changed during a major upgrade.

   The table below describes SSL parameters available for editing. The **File location** column describes where each file should be stored on the specified path.

Parameter	Description	File location
ADPG generic certificate	Path to a generic ADPG certificate that meets the requirements for all ADPG services	On each cluster host where the ADPG, Balancer, Etcd, or Monitoring services are installed
ADPG generic certificate key	Path to a generic ADPG certificate key that meets the requirements for all ADPG services. The permissions for the generic ADPG certificate key file should be set to **0600** — the file owner has read and write permissions, while the group and others have no permissions at all. If the file's owner service (YAML) is used, the file name should match the [service_name] parameter, e.g., for etcd.pem, postgresql.key	On each cluster host where the ADPG, Balancer, Etcd, or Monitoring services are installed
Annotated PostgreSQL certificate	Path to a separate certificate with special rights for the ADPG service's user. It is equivalent to the [ADCLIENT_CRT] parameter from the postgresql.conf file. The [ADCLIENT_CRT] user should have the permissions for the [ADCLIENT_KEY] file.	On each cluster host where the ADPG service is installed
Annotated PostgreSQL certificate key	Path to a key of a separate certificate for the ADPG service's user (e.g., it is equivalent to the [ADCLIENT_KEY] parameter from the postgresql.conf file). The permissions for the Annotated PostgreSQL certificate key file should be set to **0600** — the file owner has read and write permissions, while the group and others have no permissions at all. The [ADCLIENT_KEY] user should have the owner permissions on the certificate key file.	On each cluster host where the ADPG service is installed
CA file	Path to a file containing the SSL server certificate authority (CA) for verifying client certificates	On each cluster host where the ADPG, Balancer, Etcd, or Monitoring services are installed
CA backup file	Path to a file containing the SSL server certificate authority (CA) for all CA-signing certificates. Specify the parameter if [backup_postgresql](#) and [Main Type](#) parameter is set to **ALL**	On each cluster host where the ADPG service is installed

4. After you specify SSL parameters, click **Next**.

The 'Run an action: Manage SSL' window with specified SSL settings

5. On the **Confirmation** tab, click **Run** to execute the **Manage SSL** action.

The 'Confirmation' tab

After the **Manage SSL** action is executed, SSL settings are added to the **SSL configuration** section of the ADPG service configuration. It contains the following fields:

- **Enable SSL** — A read-only field that is set to **TRUE** if SSL is enabled.
- **postgresql.conf** — A read-only field that contains SSL settings.
- **PG_FILE** — A read-to-specify-path-for-SSL connections.

6. After the **Manage SSL** action is finished, it is necessary to allow SSL connections for the ADPG server. To do this, open the **Primary configuration** tab of the ADPG service, expand the **SSL configuration** section in the **Configuration** tree, and click the **PG_FILE** field.

The 'SSL configuration' section

Add a new record to allow SSL connections to the **PG_FILE** field of the **SSL configuration** section. For example:

```
hostssl   all     all     0.0.0.0/0     cert
```

The 'PG_FILE' field

This record allows all users (**all**) to utilize SSL connections (**hostssl**) to all databases (**all**) from all IPv4 addresses (**0.0.0.0/0**). Authentication is performed using SSL client certificates (**cert**). See [pg_hba.conf file format](#) documentation for more details.

7. Save the changes and run the [Reconfigure & Restart](#) action to apply new settings.

> **NOTE**
>
> The **Manage SSL** action overrides SSL parameters specified in the *postgresql.conf* and *Custom postgresql.conf* sections located in the **Configuration** tree of the ADPG service. Note that if you disable SSL using the **Manage SSL** action, the SSL settings from these sections will be restored. To avoid ambiguous situations, it is recommended to remove the SSL settings from these sections before running the **Manage SSL** cluster action.

Use client certificates

For a **hostssl** record in the **PG_HBA** section, you can set the **clientcertverify-ca** or **clientcertverify-full** authentication option.

For **clientcertverify-ca**, the server verifies that the certificate presented by a user is signed by one of the trusted certificate authorities. If **clientcertverify-full** is used, the server does not only verify the certificate, but it also checks whether the user name or its mapping matches the **CN** (Common Name) of the provided certificate. For example, if a user named *test_user* tries to log in, the name *test_user* or its mapping must be specified in the **CN** field.

The **clientcert** parameter can be used in addition to the **hostssl** entries with the verification-ca or verification-full authentication options. This parameter controls whether a client certificate is required by the server for SSL connections. For example, if **clientcert** is set to **verify-ca** or **verify-full**, the server will require a valid client certificate to be presented and the CA is configured.

There are two possible approaches to the **hostssl** entries with the verification-ca or verification-full authentication options:

- Use the **cert** authentication method for **hostssl** entries in the **PG_HBA** records. The certificate will be used for both authentication and security of the SSL connection. See [pg_hba.conf file format](#) documentation for more details. It is necessary to specify a valid **clientcert** value when using the **cert** authentication method in this case. **CN** (Common Name) provided in the certificate is checked against the user name or an applicable mapping.
- Use one authentication method in the **hostssl** entries with the verification-ca or verification-full authentication options and **clientcert** authentication option = **verify-ca** or **verify-full**. The former option is checked in this case, while the latter also ensures that **CN** (Common Name) in the certificate matches the user name or an applicable mapping.

For information on how to set up certificates on the client, see [SSL Support](#).

Create self-signed SSL certificates

Instead of CA certificates, you can also create your own self-signed certificates, which can be used for development and testing, but are not recommended for use in a production environment.

Generate certificates for an ADPG cluster

The next ADPG cluster for which the certificates will be created includes three hosts with the services described in the table below.

Host	IP	FQDN	Service
web-host-1	10.92.41.39	web-host-1.ru-central1.internal	ADPG
web-host-2	10.92.46.220	web-host-2.ru-central1.internal	Etcd
web-host-3	10.92.41.32	web-host-3.ru-central1.internal	Balancer

In this example, certificates are created on the **web-host-1** host with the ADPG service.

For the ADPG cluster, it is necessary to generate SSL certificates with the Unix Eastern-Hemisphere Names option. To do this, create a temporary configuration file with the **ADCLUSTER** section corresponding to the cluster topology. For example, you can place it in the following path: /tmp/cnf. For the next ADPG cluster, the file looks like this:

```
[ req ]
default_bits = 4096
prompt = no
default_md = sha256
req_extensions = req_ext
distinguished_name = dn

[ dn ]
C = RU
L = Moscow
O = Arenadata
CN = Cassini Demo

[ req_ext ]
basicConstraints = CA:FALSE
keyUsage = digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth, clientAuth
subjectAltName = @alt_names

[ alt_names ]
IP.1 = 127.0.0.1
IP.2 = 10.92.41.39
IP.3 = 10.92.46.220
IP.4 = 10.92.41.32
DNS.1 = localhost
DNS.2 = web-host-1.ru-central1.internal
DNS.3 = web-host-2.ru-central1.internal
DNS.4 = web-host-3.ru-central1.internal
DNS.5 = web-host-1
DNS.6 = web-host-2
DNS.7 = web-host-3

[ ADCLUSTER ]
ADPG = web-host-1.ru-central1.internal
```

To generate SSL certificates, follow the steps below:

1. Create a directory where you want to generate certificates: grant full permissions, and navigate to it. For example:

```
$ sudo mkdir /etc/certificates
$ sudo chmod 777 /etc/certificates
$ cd /etc/certificates
```

2. To create a server certificate that clients can verify, create a certificate request (Certificate Signing Request, CSR) and a public/private key files. Replace **<fqdn_of_server>** with the fully qualified domain name of the server. Ubuntu/Debian **hostfqdn** is also necessary to set the required permissions:

```
$ openssl req -new -nodes -text -out root.csr -keyout root.key -subj "/CN=<root_yourdomain_com>"
$ chmod 600 root.key
```

3. Sign the request with the key to create a root certificate authority. Note that the path to the openssl.cnf file (etc/ssl/openssl.cnf) can be different in your environment:

```
$ openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

Set the required permissions for the root.crt file:

```
$ chmod 644 root.crt
```

4. Create the following files: one executable and one text and add text into:

```
$ touch serial root.srl index.txt
$ echo 1000 > serial
```

5. Generate the following files: one executable, 1 pc. postgres_1.csr and postgres_1.key

```
$ openssl req -new -nodes -text -out postgres_1.csr -keyout postgres_1.key -subj "/CN=ClusterUsers"
$ chmod 600 postgres_1.key
$ openssl x509 -req -in postgres_1.csr -text -days 365 -CA root.crt -CAkey root.key -CAcreateserial -out postgres_1.crt -extfile /tmp/san.cnf -extensions v3_req
```

The following files are created: generic_1.csr, generic_1.crt and generic_1.crtkey

```
$ openssl req -new -nodes -text -out generic_1.csr -keyout generic_1.key -subj "/CN=ClusterUsers" -config /tmp/san.cnf
$ chmod 600 generic_1.key
$ openssl x509 -req -in generic_1.csr -text -days 365 -CA root.crt -CAkey root.key -CAcreateserial -out generic_1.crt -extfile /tmp/san.cnf -extensions v3_req
```

The following files are created: generic_1.csr, generic_1.crt and generic_1.crtkey.

6. Verify certificates. The commands below should return **OK**:

```
$ openssl verify -CAfile root.crt generic_1.crt
$ openssl verify -CAfile root.crt postgres_1.crt
```

7. Change permissions for Annotated PostgreSQL certificates and its key:

```
$ sudo chown postgres:postgres /etc/certificates/postgres_1.*
```

8. Copy the CA certificates to the standard directory for storing CA certificates. By default, this directory is /usr/local/share/ca-certificates/ and run the **update-ca-certificates** command to update the trusted certificates cache. This step should be performed on all hosts with the ADPG service after you copy the certificates to them:

```
$ sudo cp generic_1.crt /usr/local/share/ca-certificates/
$ sudo cp root.crt /usr/local/share/ca-certificates/
$ sudo cp root.crt /usr/local/share/ca-certificates/
$ sudo update-ca-certificates
```

9. Copy the generic_1.crt and generic_1.crtkey files to other cluster hosts using the same path (in this example — /etc/certificates) and set the permissions for generic_1.crtkey:

```
$ sudo chmod 644 generic_1.crt key
```

Paths to these files should also be specified in the **ADPG generic certificate** key and **ADPG generic certificate** parameters of the **Manage SSL** action.

10. Copy the files to other cluster hosts using the same path (in this example, /etc/certificates) and set the permissions for postgres_1.*:

```
sudo chown postgres:postgres /etc/certificates/postgres_1.*
```

Paths to these files should also be specified in **Annotated PostgreSQL certificate key** parameters of the **Manage SSL** action.

11. Copy the root.crt file to other cluster hosts using the same path (in this example, /etc/certificates) and set the permissions for root.crt:

```
$ chmod 644 root.crt
```

The path to this file should be specified in the **CA file** parameter of the **Manage SSL** action.

12. Perform step 8 on all hosts with the ADPG service.

Generate an SSL certificate for PostgreSQL clients

Create a certificate request (CSR) where **CN** is a database user. For example:

```
$ openssl req -new -nodes -text -out client.csr -keyout client.key -subj "/CN=postgres_username"
```

For example:

```
$ openssl req -new -nodes -text -out client.csr -keyout client.key -subj "/CN=postgres"
```

Set the following permissions for client.key:

```
$ chmod 600 client.key
```

Create a signed certificate for the client using the root certificate generated in the example above:

```
$ openssl x509 -req -in client.csr -text -days 365 -CA root.crt -CAkey root.key -out client.crt
```

Remove the CSR file:

```
$ rm client.csr
```

The client.crt and client.key files should be stored on a client. See [SSL Client Settings](#).

All certificates are generated and placed on cluster hosts, specify the required ADPG service settings via ADCM UI:

1. Add a record that allows SSL connections to the **PG_HBA** field of the **SSL configuration** section. For example:

```
hostssl all all 0.0.0.0/0 cert clientcert=1
```

2. Set the **SSL_CA_FILE** parameter in the *postgresql.conf* of the **ADPG configuration** section. In this example, it has the following value:

```
ssl_ca_file='/etc/certificates/root.crt'
```

3. Execute the [Reconfigure & Restart](#) action of the ADPG service.

Database roles and privileges

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

Create a role

Role attributes

Role membership

- The SET ROLE command
- The INHERIT attribute

Drop a role

ADPG uses roles to manage database access permissions. A role represents a database user or a group of database users. Roles can own database objects (for example, tables and functions) and allow access and actions on these objects (assign privileges). It is also possible to grant membership in a role to another role to use its privileges.

Database roles are separate from operating system users, but you can create roles corresponding to operating system users. Roles are global to the entire ADPG cluster, not to a single database.

The ADPG cluster contains a predefined superuser role `postgres`. Connect as `postgres` to create additional roles.

Create a role

To create a role, use the CREATE ROLE SQL command. You can run commands in the psql command line.

```
CREATE ROLE user1 WITH LOGIN PASSWORD 'password1';
```

The command above creates the role `user1` with the password `password1`. The LOGIN parameter allows the role to log in: `user1` can be an initial session authorization name during client connection.

If you need to use special characters in a role name, enclose the name in double quotes. For additional information about SQL identifiers, refer to Identifiers and keywords.

You can also execute the createuser command from the shell command line to create a role. `createuser` is a wrapper for the `CREATE ROLE` SQL command. Connect to the ADPG host as a superuser or a user with the CREATEROLE privilege to run the `createuser` command.

```
$ createuser user2
```

The pg_roles view provides access to information about database roles. You can execute the following query to check if a role is added:

```
SELECT rolname FROM pg_roles;
```

The `user1` and `user2` items should be in the result with PostgreSQL predefined roles:

```
pg_database_owner
pg_read_all_data
pg_write_all_data
pg_monitor
pg_read_all_settings
pg_read_all_stats
pg_stat_scan_tables
pg_read_server_files
pg_write_server_files
pg_execute_server_program
pg_signal_backend
postgres
user1
user2
```

The `\du` psql meta-command also lists existing roles.

NOTE

Only a user with superuser privileges can create a new superuser role or grant superuser privileges.

Role attributes

A database role can have attributes that define its privileges and interact with the client authentication system.

Role attributes

Name	Description	Example
LOGIN	The role can be used as the initial role name for a database connection (an equivalent of the CREATE USER command)	`CREATE ROLE name LOGIN;`
SUPERUSER	A database superuser bypasses all permission checks, except the right to log in. Since this role has high-level privileges, minimize the use of this role for security reasons	`CREATE ROLE name SUPERUSER;`
CREATEDB	The role has a permission to create databases	`CREATE ROLE name CREATEDB;`
CREATEROLE	The role has a permission to create roles. A role with CREATEROLE privilege can alter and drop other roles, grant or revoke membership in them	`CREATE ROLE name CREATEROLE;`
REPLICATION	A role has a permission to initiate streaming replication. This role must also have the LOGIN permission	`CREATE ROLE name REPLICATION LOGIN;`
PASSWORD	Use this attribute if the client authentication method requires a password. For more information, see Password authentication	`CREATE ROLE name PASSWORD 'password_string';`

NOTE

We recommend you to create a role that has the CREATEDB or CREATEROLE privileges instead of a superuser role and use this role for routine operations. This approach restricts superuser access and reduces security risks.

Use the ALTER ROLE command to modify role attributes after creation. The following command changes the password of the `user1` role:

```
ALTER ROLE user1 WITH PASSWORD 'password2';
```

You can also add the CREATEDB or CREATEROLE privileges to the `user2` role:

```
ALTER ROLE user2 CREATEROLE CREATEDB;
```

Role membership

You can group users to manage their privileges. To do this, create a group role:

```
CREATE ROLE group1 CREATEROLE CREATEDB;
```

There is no difference between a group role and individual user role, except the way of use. Group roles usually do not have the LOGIN attribute.

Execute the GRANT command to add members to the group role. The command below adds `user1` and `user2` to `group1`.

```
GRANT group1 TO user1, user2;
```

To remove a group member, utilize the REVOKE command.

```
REVOKE group1 FROM user1;
```

To use group privileges, call the SET ROLE command or add the `INHERIT` attribute to a group member role.

The SET ROLE command

Log in as a group member and execute the SET ROLE command.

```
SET ROLE 'group1';
```

As a result, the group member temporarily 'becomes' `group1`. The database session has access to group role privileges and creates database objects on behalf of a group role: the owner of all created objects is `group1`.

The INHERIT attribute

Group member roles that have the `INHERIT` attribute automatically use group privileges. In the example below, `user3` can utilize the `group1` privileges after logging in.

```
CREATE ROLE user3 LOGIN INHERIT;
GRANT group1 TO user3;
```

Drop a role

To drop a role, drop or reassign all objects that the role owns. Granted permissions also must be revoked.

You can use the ALTER command to reassign an object. The command below assigns the database table `table1` to the role `user2`.

```
ALTER TABLE table1 OWNER TO user2;
```

Alternatively, execute the REASSIGN OWNED command to reassign an entire role ownership to another role. The REASSIGN OWNED command transfers database objects to a new owner, but it does not affect granted privileges for objects that do not belong to the role. Call the DROP OWNED command to drop these privileges.

```
REASSIGN OWNED BY user1 TO user2;
DROP OWNED BY user1;
```

Since the commands above cannot access objects in other databases, run them for each database that contains objects owned by the role.

After all dependencies are removed, call the DROP ROLE command to remove a role:

```
DROP ROLE user1;
```

You can also execute the dropuser command from the shell command line to drop a role. `dropuser` is a wrapper for the `DROP ROLE` SQL command. Connect to the ADPG host as a superuser or a user with the CREATEROLE privilege to run the `dropuser` command.

```
$ dropuser user1
```

Column and row level security

Eugenia Kuzina

To Table of Contents

In addition to the SQL-standard privilege system, PostgreSQL provides row security policies and additional options to set up column security.

This technique is useful if you need to delimit user access to certain rows or columns in a table. Refer to the following articles for details:

- Row security policies
- Column-level security

Row security policies

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

Enable row security

Manage row security policies

Disable row security

You can apply **row security policies** to a table. These policies determine, on a per-user basis, which rows can be processed. You can create a row security policy for specific commands like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, or specify it for `ALL` commands. This feature is also known as **row-level security**.

Connect to the database as the `postgres` user and create a table for examples. You can use psql for this:

```
$ psql -h 10.92.6.36 -p 5432 -U postgres postgres
```

In the code above, `10.92.6.36` is the server address, `5432` is the port for connections, `-U postgres` is a user name, and `postgres` is the database name.

```
CREATE TABLE users (
user_id serial PRIMARY KEY,
user_name  VARCHAR NOT NULL,
pwhash VARCHAR,
real_name  VARCHAR NOT NULL
);
```

Insert data to the table:

```
INSERT INTO users (user_name, pwhash, real_name)
    VALUES
    ('james', 'xxx', 'James Brown'),
    ('mary', 'xxx', 'Mary Smith'),
    ('alice', 'xxx', 'Alice Gray');
```

Create roles and use the GRANT command to define their access privileges on the `users` table:

```
CREATE ROLE james WITH LOGIN PASSWORD 'password1';
GRANT ALL ON users TO james;

CREATE ROLE mary WITH LOGIN PASSWORD 'password2';
GRANT ALL ON users TO mary;
```

Now users `james` and `mary` have `ALL` privileges on the `users` table.

Enable row security

To enable row security, execute the ALTER TABLE command with the `ENABLE ROW LEVEL SECURITY` parameter.

```
ALTER TABLE users ENABLE ROW LEVEL SECURITY;
```

When row security is enabled on a table, access to the table for selecting and modifying rows must be allowed by a row security policy. If no policy is defined for a table, the default deny policy is applied: no rows are visible or can be modified. Operations that are applied to the whole table, for example, `TRUNCATE`, `REFERENCES`, are not subject to row security.

Superusers and roles with the `BYPASSRLS` attribute always bypass the row security system. Table owners are also not affected by row security, but they can enable it for themselves with the `ALTER TABLE … FORCE ROW LEVEL SECURITY` command.

Reconnect to the database as the user `james`:

```
$ psql -h 10.92.6.36 -p 5432 -U james postgres
```

If you try to select rows from the `users` table, you will get the following result:

```
SELECT * FROM users;
 user_id | user_name | pwhash | real_name
---------+-----------+--------+-----------
(0 rows)
```

The user `james` cannot access table data because row security is enabled and no row security policy is created.

> **NOTE**
>
> Only a table owner can enable and disable row security and add policies to a table.

Manage row security policies

Row security policies can be specific to commands, or to roles, or both. A policy can be specified to apply to `ALL` commands, or to `SELECT`, `INSERT`, `UPDATE`, or `DELETE`. Multiple roles can be assigned to a current policy. See CREATE POLICY for details.

PostgreSQL also applies standard role membership and inheritance rules to row security policies.

Each policy has a name. Multiple policies can be defined for a single table. Policies are table-specific, each policy for a table must have a unique name. Different tables can have policies with the same name.

Reconnect to the database as the `postgres` user and define the following row security policy:

```
$ psql -h 10.92.6.36 -p 5432 -U postgres postgres
```

```
CREATE POLICY policy1 ON users FOR ALL TO PUBLIC USING (user_name=current_user);
```

This policy allows all roles to execute all commands if the particular value from the `user_name` column of the `users` table is equal to the current user name.

`PUBLIC` is used instead of a role name and applies the policy to all roles.

The `USING` expression is any SQL conditional expression returning a boolean value.

According to this policy, the users `james` and `mary` created above can manipulate rows that contain information about each of them.

Reconnect to the database as `james` and select data:

```
$ psql -h 10.92.6.36 -p 5432 -U james postgres
```

```
SELECT * FROM users;
```

The result:

```
user_id | user_name | pwhash |  real_name
---------+-----------+--------+-------------
       1 | james     | xxx    | James Brown
```

Reconnect to the database as `mary` and select data:

```
$ psql -h 10.92.6.36 -p 5432 -U mary postgres
```

```
SELECT * FROM users;
```

The result:

```
user_id | user_name | pwhash | real_name
--------+-----------+--------+-----------
      2 | mary      | xxx    | Mary Smith
```

Reconnect to the database as the `postgres` user, create a new user `alice` and add a new policy.

```
$ psql -h 10.92.6.36 -p 5432 -U postgres postgres
```

```
CREATE ROLE alice WITH LOGIN PASSWORD 'password3';
GRANT ALL ON users TO alice;

CREATE POLICY policy2 ON users TO alice USING (true) WITH CHECK (true);
```

The `WITH CHECK` expression should return `true` to allow a user to insert and update rows.

This policy allows `alice` all operations on all rows from the table.

Reconnect to the database as `alice` and check the result:

```
$ psql -h 10.92.6.36 -p 5432 -U alice postgres
```

```
DELETE FROM users WHERE user_id=1;

SELECT * FROM users;
```

The result:

```
user_id | user_name | pwhash |  real_name
---------+-----------+--------+-------------
       2 | mary      | xxx    | Mary Smith
       3 | alice     | xxx    | Alice Gray
```

You can also use the ALTER POLICY command to update a policy. The following code renames `policy1` to `policy_for_users`:

```
ALTER POLICY policy1 ON users RENAME TO policy_for_users;
```

The DROP POLICY command deletes a policy.

```
DROP POLICY policy_for_users ON users;
```

Disable row security

To disable row security, execute the `ALTER TABLE` command with the `DISABLE ROW LEVEL SECURITY` parameter.

```
ALTER TABLE users DISABLE ROW LEVEL SECURITY;
```

Column-level security

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

Create a table view

Grant access to specific columns

Use column encryption

PostgreSQL allows you to restrict access to a particular column or set of columns for a user. This way, the user cannot view or modify column data.

You can use the following options to implement column-level security:

- Create a table view
- Grant access to specific columns
- Use column encryption

Connect to the database as the `postgres` user and create a table for examples. You can use psql for this:

```
$ psql -h 10.92.6.36 -p 5432 -U postgres postgres
```

In the code above, `10.92.6.36` is the server address, `5432` is the port for connections, `-U postgres` is a user name, and `postgres` is the database name.

```
CREATE TABLE users (
user_id serial PRIMARY KEY,
user_name VARCHAR NOT NULL,
real_name VARCHAR NOT NULL,
address VARCHAR,
salary INT
);
```

Insert data to the table:

```
INSERT INTO users (user_name, real_name, address, salary)
    VALUES
    ('james', 'James Brown', '123 2nd Street', 5000),
    ('mary', 'Mary Smith', '456 13th Street', 6000),
    ('alice', 'Alice Gray', '789 19th Street', 5700);
```

Create roles:

```
CREATE ROLE user1 WITH LOGIN PASSWORD 'password1';
CREATE ROLE user2 WITH LOGIN PASSWORD 'password2';
```

Create a table view

You can create a view that includes only the allowed columns and grant the user access to this view, and not to the underlying table. For example, you can hide the `salary` column from `user1`. Create a view with the `user_name`, `real_name`, and `address` columns and grant `user1` privileges on this view:

```
CREATE VIEW users_info AS SELECT user_name, real_name, address FROM users;
GRANT ALL ON users_info TO user1;
```

Reconnect to the database as `user1`:

```
$ psql -h 10.92.6.36 -p 5432 -U user1 postgres
```

Select data from the `users_info` view:

```
SELECT * FROM users_info;
```

Since `user1` has permissions to this view, the result is the following:

```
 user_name |  real_name  |     address
-----------+-------------+-----------------
 james     | James Brown | 123 2nd Street
 mary      | Mary Smith  | 456 13th Street
 alice     | Alice Gray  | 789 19th Street
```

If you try to select all data from the `users` table on behalf of `user1`, an error occurs, because `user1` does not have access rights to the entire table:

```
SELECT * FROM users;
```

The result:

```
ERROR: permission denied for table users
```

Grant access to specific columns

You can grant access to a user only to certain columns of a table. For example, you can hide the `salary` column from `user2`. Reconnect to the database as the `postgres` user and add permissions to `user2` for all columns of the `users` table except `salary`:

```
$ psql -h 10.92.6.36 -p 5432 -U postgres postgres
```

```
GRANT SELECT (user_name, real_name, address) ON users TO user2;
```

Reconnect to the database as `user2`:

```
$ psql -h 10.92.6.36 -p 5432 -U user2 postgres
```

Select data from allowed columns of the `users` table:

```
SELECT user_name, real_name, address FROM users;
```

The result:

```
 user_name |  real_name  |     address
-----------+-------------+-----------------
 james     | James Brown | 123 2nd Street
 mary      | Mary Smith  | 456 13th Street
 alice     | Alice Gray  | 789 19th Street
```

Note, if `user2` tries to select all columns from the `users` table, an error occurs:

```
SELECT * FROM users;
```

The result:

```
ERROR: permission denied for table users
```

Use column encryption

It is also possible to encrypt specific database columns to protect data. You can use an additional pgcrypto module for this. For information on how to install and use this module, see Encryption.

The `pgp_sym_encrypt` and `pgp_sym_decrypt` function from the pgcrypto module are used in the example below.

Reconnect to the database as `user1` and create the `users_encryption` table:

```
$ psql -h 10.92.6.36 -p 5432 -U user1 postgres
```

```
CREATE TABLE users_encryption (
user_id serial PRIMARY KEY,
user_name VARCHAR NOT NULL,
real_name VARCHAR NOT NULL,
address VARCHAR,
salary TEXT
);
```

Add data and encrypt the `salary` column using the `pgp_sym_encrypt` function:

```
INSERT INTO users_encryption (user_name, real_name, address, salary)
    VALUES
    ('james', 'James Brown', '123 2nd Street', pgp_sym_encrypt('5000','secret_encryption_key')),
    ('mary', 'Mary Smith', '456 13th Street', pgp_sym_encrypt('6000','secret_encryption_key')),
    ('alice', 'Alice Gray', '789 19th Street', pgp_sym_encrypt('5700','secret_encryption_key'));
```

Select data to check if the `salary` column is encrypted:

```
SELECT * FROM users_encryption;
```

The result:

```
user_id | user_name |  real_name  |     address     |           salary
--------+-----------+-------------+-----------------+-----------------------------------------------------
      1 | james     | James Brown | 123 2nd Street  | \xc30d04070302ec4106d9f16a58767ed23501d9679...
      2 | mary      | Mary Smith  | 456 13th Street | \xc30d04070302a972f63c44aa0b8270d23501ce968...
      3 | alice     | Alice Gray  | 789 19th Street | \xc30d04070302edf9f8629c0fb48669d23501560fe...
```

Reconnect to the database as the `postgres` user and add permissions to `user2` for the `users_encryption` table:

```
$ psql -h 10.92.6.36 -p 5432 -U postgres postgres
```

```
GRANT ALL ON users_encryption TO user2;
```

Reconnect as `user2`, and select data:

```
$ psql -h 10.92.6.36 -p 5432 -U user2 postgres
```

```
SELECT * FROM users_encryption;
```

The result is the same:

```
user_id | user_name |  real_name  |     address     |           salary
--------+-----------+-------------+-----------------+-----------------------------------------------------
      1 | james     | James Brown | 123 2nd Street  | \xc30d04070302ec4106d9f16a58767ed23501d9679...
      2 | mary      | Mary Smith  | 456 13th Street | \xc30d04070302a972f63c44aa0b8270d23501ce968...
      3 | alice     | Alice Gray  | 789 19th Street | \xc30d04070302edf9f8629c0fb48669d23501560fe...
```

Execute the `SELECT` statement with the `pgp_sym_decrypt` function to view decrypted data from the `salary` column:

```
SELECT user_id, user_name, real_name, address, pgp_sym_decrypt(salary::bytea, 'secret_encryption_key') AS salary FROM
users_encryption;
```

The result:

```
user_id | user_name |  real_name  |     address     | salary
--------+-----------+-------------+-----------------+--------
      1 | james     | James Brown | 123 2nd Street  | 5000
      2 | mary      | Mary Smith  | 456 13th Street | 6000
      3 | alice     | Alice Gray  | 789 19th Street | 5700
```

Note, any user that has the encryption key can decrypt the column data. In the example above, `user1` encrypts data and `user2` decrypts data using the encryption key.

ADPG cluster management via ADCM

Eugenia Kuzina

To Table of Contents

You can use ADCM to manage the ADPG cluster.

The following actions are available in the ADCM UI:

- Cluster actions that allow you to manage the entire cluster with all services.
- Service actions that manage a single service.

Cluster actions

Eugenia Kuzina

Contents

To Table of Contents

Overview

Check and Precheck

Manage SSL

Reinstall

Reinstall statuschecker

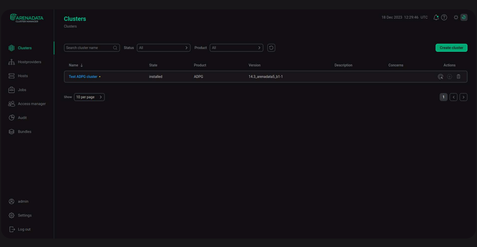
Start

Stop

This article describes the actions available for the ADPG cluster in the ADCM UI.

Overview

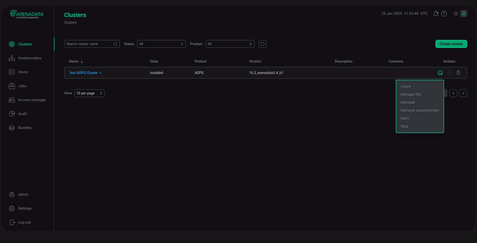
You can find cluster actions on the ***Clusters*** page.



The "Clusters" page

Refer to ADCM documentation for an overview of the ***Clusters*** page and common actions.

Click ⋯ in the ***Actions*** column to display a drop-down list with the available actions and select an action to run it.



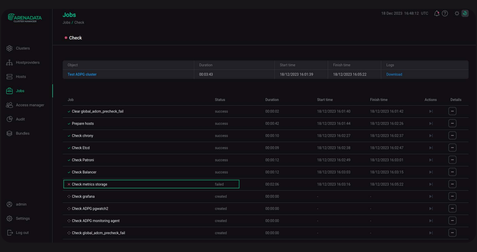
Actions

Check and Precheck

The ***Precheck*** action verifies if hosts have correct settings and all components and services are configured according to the ADPG cluster requirements. ***Precheck*** is available before the cluster installation.

The ***Check*** action includes ***Precheck*** and also checks if the installed cluster components operate correctly. ***Check*** is available after the cluster installation.

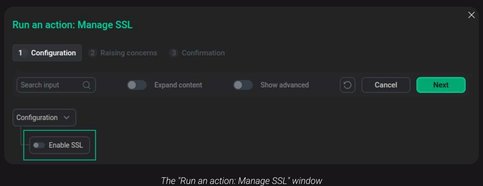
You can find the result on the ***Jobs*** page. If the action completes successfully, the corresponding task changes its status from `running` to `success`. If the job fails, it switches to the `failed` status. The actions also store the result in the `global_adcm_precheck_fail` ADCM variable to implement a postponed failure. The result is a boolean value. It is `true` if the last ***Check*** or ***Precheck*** action is successful. The `false` value indicates that an error occurred during checking. In this case, the status of the ***Check global_adcm_precheck_fail*** operation is changed to `failed`. In some cases, when ***Check global_adcm_precheck_fail*** is failed, you need to review inner steps of the ***Check*** or ***Precheck*** action that are marked as successful to determine the cause of the problem.



An error occurs while performing the action

Manage SSL

The ***Manage SSL*** action enables or disables SSL in an ADPG cluster. To enable SSL, switch on the ***Enable SSL*** toggle button in the ***Run an action: Manage SSL*** window and follow the instructions from Enable SSL encryption.

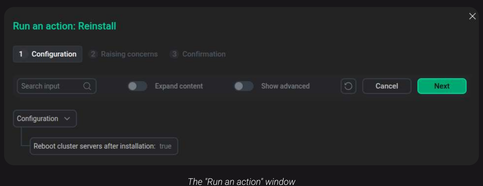


The "Run an action: Manage SSL" window

To disable SSL, switch off the ***Enable SSL*** toggle button and click ***Run*** on the ***Confirmation*** tab to execute the ***Manage SSL*** action.

Reinstall

The ***Reinstall*** action reinstalls the ADPG cluster. In its ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation.



The "Run an action" window

Reinstall statuschecker

The ***Reinstall statuschecker*** action reconfigures and restarts the status checker for all cluster services.

Start

The ***Start*** action starts all cluster services.

Stop

The ***Stop*** action stops all cluster services.

Upgrade a cluster to a new ADPG version

By Adrien Obernesser

Contents

Prepare cluster for upgrade
Upgrade of a minor version
Upgrade of a major version

To Table of Contents

CAUTION

- It is highly recommended to create a backup before an upgrade, especially if you intend to upgrade ADPG to a new major version.
- Before an upgrade, run the `Check` cluster action and ensure that during the action execution there are no jobs in the `Failed` status. If errors are found, fix them before you proceed to the steps below.
- When upgrading from v16.3.3 with enabled SSL, it is recommended to switch SSL off, then upgrade to v16.3.4 and use the **Manage SSL** cluster action to enable SSL.

At present, ADPG supports upgrades of minor versions and the upgrade of the major ADPG version from v14.3.8 to v15.3-1/v16.3-3. ADPG v16.3.x is based on PostgreSQL 16.3. The following options for upgrading of Enterprise Edition (EE) and Community Edition (CE) are possible:

- from ADPG EE v14.3.8 to ADPG EE v15.3-1/v16.3-3;
- from ADPG CE v14.3.8 to ADPG CE v15.3-1/v16.3-3.

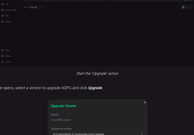
If you need to upgrade CE v14.3.7 and lower to EE v15.3-1/v16.3-3, first upgrade CE v14.x to EE v14.3.8 and then upgrade EE v14.3.8 to EE v15.3-1/v16.3-3. If you need to upgrade EE v14.3.8 to EE v14.3.4 or higher, first upgrade EE v14.3.8 to EE v16.3.3 and then upgrade EE v16.3.3 to EE v16.3.4 or higher.

Prepare cluster for upgrade

Upload a new bundle. To do this, go to the **Bundles** page, click **Upload bundle**, and select a new bundle file in your environment.

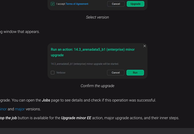
The Bundles page

When a new bundle is loaded, open the **Clusters** page. Click the icon ⋮ that becomes active for the cluster in the **Actions** column.



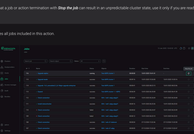
Start the "Upgrade" action

In the **Upgrade Cluster** window that opens, select a version to upgrade ADPG and click **Upgrade**.



Select version

Click **Run** in the confirmation dialog window that appears.



Confirm the upgrade

The cluster will be prepared for upgrade. You can open the **Jobs** page to see details and check if this operation was successful.
Other steps are different for the minor and major versions.
Note that on the **Jobs** page, the **Stop the job** button is available for the **Upgrade minor EE** action, major upgrade actions, and their inner steps.

CAUTION

It is important to remember that a job or action termination with **Stop the job** can result in an unpredictable cluster state, use it only if you are ready to restore the cluster from a backup.

Stop the job of an action terminates all jobs included in this action.

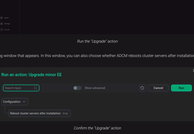
Stop the job" of an action

Stop the job of an inner action job terminates the current job and the action continues to run starting with the job that follows the terminated one.

Stop the job" of an inner action job

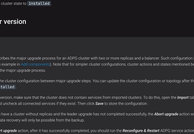
Upgrade of a minor version

As a result of the previous action, ADCM changes the cluster state to `ready for upgrade` and fills the drop-down list in the **Actions** column with two items: **Precheck** and **Upgrade**. Click the icon ⋮ in the **Actions** column and run the **Precheck** action to make sure that hosts have correct settings and all components and services are configured according to the ADPG cluster requirements.
If **Precheck** is successful, you can find the result on the **Jobs** page, execute the **Upgrade** action. The action name includes versions CE/EE. For example, **Upgrade minor EE** means that the cluster will be upgraded to the next minor Enterprise Edition.



Run the "Upgrade" action

Click **Run** in the confirmation dialog window that appears. In this window, you can also choose whether ADCM reboots cluster servers after installation.



Confirm the "Upgrade" action

After upgrade, ADCM changes the cluster state to `installed`.

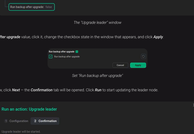
Upgrade of a major version

CAUTION

- The following section describes the major upgrade process for an ADPG cluster with two or more replicas and a balancer. Such configuration is recommended for high availability (see an example in App components). Note that for simpler cluster configurations, cluster actions and states mentioned below may be absent or different during the major upgrade.
- It is prohibited to change the cluster configuration between major upgrade steps. You can update the cluster configuration or topology after the cluster status has been changed to `installed`.
- Before upgrading a major version, make sure that the cluster does not contain services from important clusters. To do this, open the **Import** tab of the ADPG cluster being upgraded and ensure that all connected services if they exist. Then click **Save** to store the configuration.
- Also, remember that if you have a balancer without replicas and the balancer upgrade has not completed successfully, the **Abort upgrade** action will not be available and the cluster data recovery will only be possible from the backup.
- If you need to call the **Abort upgrade** action, after it has successfully completed, you should run the **Reconfigure & Restart** ADPG service action and then the **Reconfigure and restart** Monitoring service action (in case, Monitoring is used). Otherwise, the **Check** action may fail.

The **Upgrade Cluster** action prepares the cluster for upgrade. You can open the **Jobs** page to see details and check if this operation was successful.
ADCM changes the cluster state to `ready for major upgrade` and fills the drop-down list in the **Actions** column with two items: **Abort upgrade** and **Upgrade leader**.

1. Click the icon ⋮ in the **Actions** column and run the **Upgrade leader** to continue. At this step, you can also roll back the upgrade if some error occurs when the cluster was prepared for upgrading. To do this, click **Abort upgrade**.

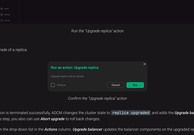


Cluster actions for major upgrade

2. On the **Configuration** tab of the opened window, the **Run backup after upgrade** option is available. If its value is set to `True` and creating backups is enabled in the cluster, after a successful cluster upgrade a backup will be created according to the settings specified in the **Enable backups** section of the ADPG configuration parameters. If **Run backup after upgrade** is set to `True`, but **Enable backups** is disabled, an entry indicating that the backup cannot be created will be added to the log.

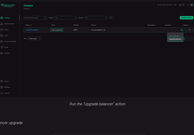
The "Upgrade leader" window

To change the **Run backup after upgrade** value, click it, change the checkbox state in the window that appears, and click **Apply**.



Set "Run backup after upgrade"

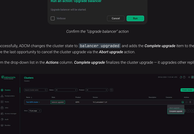
In the **Upgrade leader** window, click **Next** → the **Confirmation** tab will be opened. Click **Run** to start updating the leader node.



The "Confirmation" tab

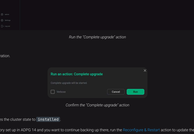
If **Upgrade leader** is successful, ADCM changes the cluster state to `leader upgraded` and adds the **Upgrade replica** item to the drop-down list in the **Actions** column. If **Upgrade leader** fails, run the **Abort upgrade** action to roll back changes in the cluster and investigate the **Jobs** page to find out the cause of the problem.

3. Click the icon ⋮ in the **Actions** column and run **Upgrade replica**. **Upgrade replica** upgrades one replica from a cluster. This replica is selected randomly among the cluster replicas. Changes are made only on one replica, so that if problems arise during the upgrade, working replicas will the previous version remain in the cluster.



Run the "Upgrade replica" action

4. Click **Run** to confirm the upgrade of a replica.



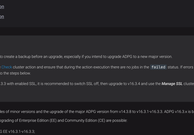
Confirm the "Upgrade replica" action

If the **Upgrade replica** operation is terminated successfully, ADCM changes the cluster state to `replica upgraded` and adds the **Upgrade balancer** item to the drop-down list in the **Actions** column. At this step, you also can click **Abort upgrade** to roll back changes.

5. Select **Upgrade balancer** from the drop-down list in the **Actions** column. **Upgrade balancer** updates the balancer components on the upgraded replica.

Run the "Upgrade balancer" action

6. Click **Run** to confirm the balancer upgrade.



Confirm the "Upgrade balancer" action

If the operation is finished successfully, ADCM changes the cluster state to `balancer upgraded` and adds the **Complete upgrade** item to the drop-down list in the **Actions** column. At this step, you have the last opportunity to cancel the cluster upgrade via the **Abort upgrade** action.

7. Select **Complete upgrade** from the drop-down list in the **Actions** column. **Complete upgrade** finalizes the cluster upgrade – it upgrades other replicas, services and components.

Run the "Complete upgrade" action

8. Click **Run** to confirm the operation.

Confirm the "Complete upgrade" action

After upgrade, ADCM changes the cluster state to `installed`.

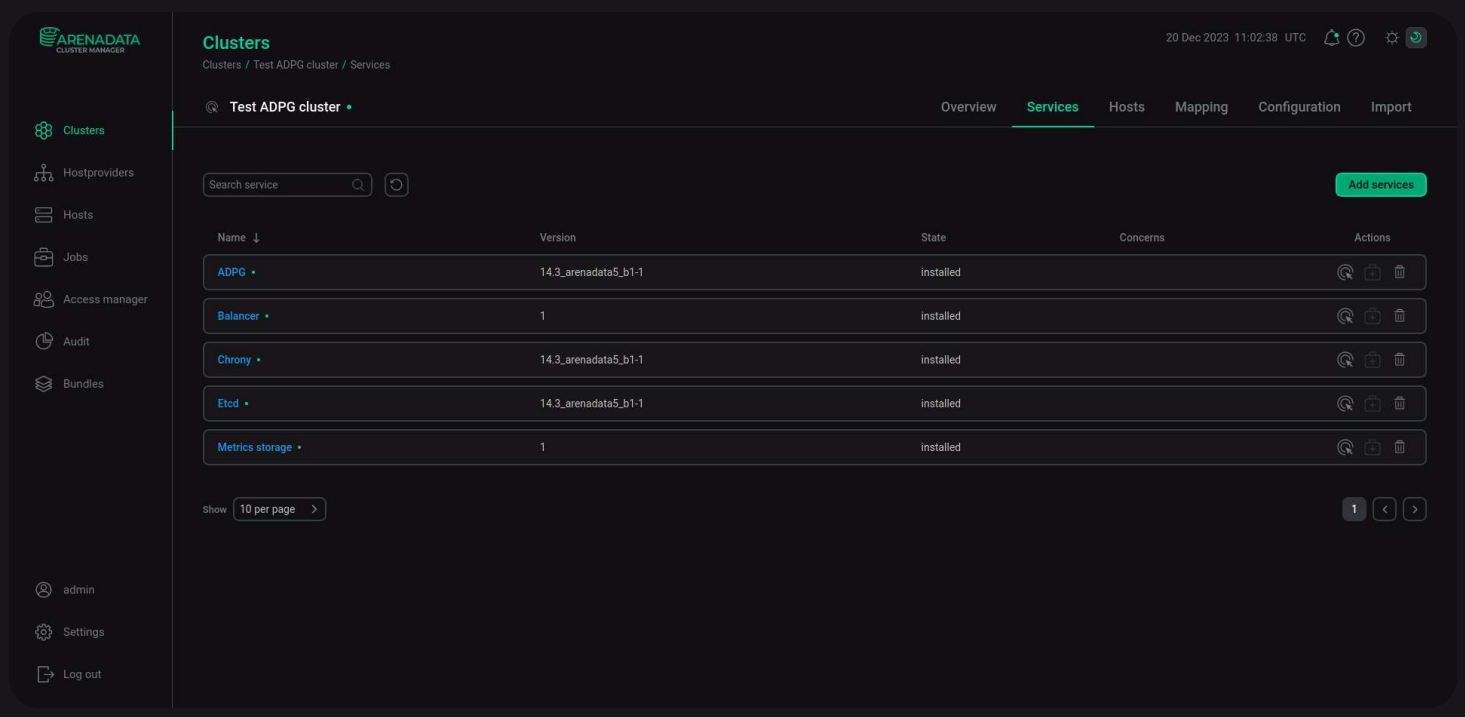
9. If you have a backup repository set up in ADPG 14 and you want to continue backing up there, run the Reconfigure & Restart action to update the backup stanza prefix to `pg16`. Otherwise, you won't be able to make backups in the upgraded ADPG 16 cluster.

Service actions

Eugenia Kuzina

To Table of Contents

ADCM UI provides actions to manage services. The actions are available on the ***Services*** tab. To open this tab, click an ADPG cluster name on the ***Clusters*** page and switch to the ***Services*** tab.



ADPG cluster services in the ADCM interface

Refer to ADCM documentation for an overview of the ***Services*** tab and common actions.

The following articles describe actions for each service of an ADPG cluster:

- ADPG
- ADPG Control Agents
- Balancer
- Chrony
- Etcd
- Monitoring

ADPG service actions

J. Eugene Kuzma

Contents

To Table of Contents

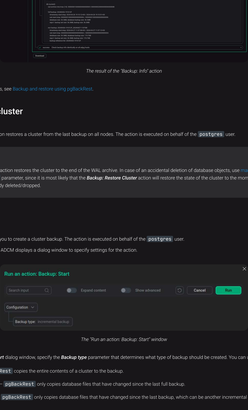
Backup: Info
Backup: Restore cluster
Backup: Start
Check
Resume Cluster
Pause Cluster
Expand
Reconfigure & Restart
Reinstall
Start
Stop

This article describes the actions available for the ADPG service.

Backup: Info

The ***Backup: Info*** action displays a list of cluster backups. The action is executed on behalf of the `postgres` user.

To view the result, open the ***Jobs*** page, click the ***Backup: Info*** row in the task list, switch to the ***Ansible [check]*** tab of the job, find the ***Check valid backup*** row, and click it to expand. Backups are grouped by node:



The result of the 'Backup: info' action

For more information on backups, see Backup and restore using pgBackRest.

Backup: Restore cluster

The ***Backup: Restore cluster*** action restores a cluster from the last backup on all nodes. The action is executed on behalf of the `postgres` user.

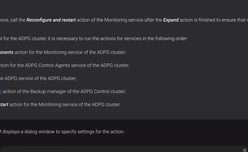
IMPORTANT

The ***Backup: Restore cluster*** action restores the cluster to the end of the WAL archive. In case of an accidental deletion of database objects, use manual PITR restore with the `--target` parameter, since it is most likely that the ***Backup: Restore Cluster*** action will restore the state of the cluster to the moment when the database objects were already deleted/dropped.

Backup: Start

The ***Backup: Start*** action allows you to create a cluster backup. The action is executed on behalf of the `postgres` user.

When you choose ***Backup: Start***, ADCM displays a dialog window to specify settings for the action.



The 'Run an action: Backup: Start' window

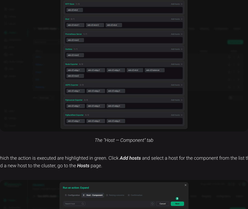
In the ***Run an action: Backup: Start*** dialog window, specify the ***Backup type*** parameter that determines what type of backup should be created. You can use the following values:

- `full backup` – `pgBackRest` copies the entire contents of a cluster to the backup.
- `differential backup` – `pgBackRest` only copies database files that have changed since the last full backup.
- `incremental backup` – `pgBackRest` only copies database files that have changed since the last backup, which can be another incremental backup, a differential backup, or a full backup.

Check

The ***Check*** action verifies whether the settings of hosts, components, and related services match the requirements of the service, and whether the service itself works correctly.

You can find the action result and the process of its execution on the Jobs page:



The "Check" action result on the "Jobs" page

Resume Cluster

The ***Resume Cluster*** action switches Patroni from the pause mode, brings the cluster to the working state, returns the control over the cluster to Patroni. The action also resolves the situation with several leaders if it occurs.

Use the Pause Cluster action to turn on Patroni pause mode.

Pause Cluster

The ***Pause Cluster*** action switches Patroni to the pause mode. In the pause mode, Patroni does not manage the cluster, but still saves the cluster state in DCS. The pause mode can be helpful during major version upgrades or corruption recovery. These actions often start and stop nodes for reasons unknown to Patroni. Some nodes can be even temporary promoted, breaking the assumption of running only one master. Therefore, Patroni needs to be able to stop controlling a running cluster.

Use the Resume Cluster action to switch off Patroni pause mode.

Expand

The ***Expand*** action adds new hosts to a service.

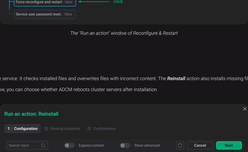
IMPORTANT

If you use the Monitoring service, call the ***Reconfigure and restart*** action of the Monitoring service after the ***Expand*** action is finished to ensure that metrics are collected from added hosts.

If you are using ADPG Control for the ADPG object, it is necessary to run the actions for services in the following order:

1. the ***Add/Remove components*** action for the Monitoring service of the ADPG cluster;
2. the ***Add components*** action for the ADPG Control Agents service of the ADPG cluster;
3. the ***Expand*** action for the ADPG service of the ADPG cluster;
4. the Apply backup config action of the Backup manager of the ADPG Control cluster;
5. the ***Reconfigure and restart*** action for the Monitoring service of the ADPG cluster.

When you choose ***Expand***, ADCM displays a dialog window to specify settings for the action.

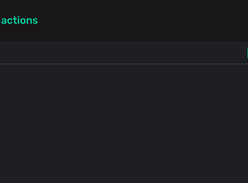


The 'Run an action' window

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation. Click ***Next*** to open the ***Host – Component*** tab where you can add cluster hosts to the component.

The 'Host – Component' tab

Components of the service for which the action is executed are highlighted in green. Click ***Add hosts*** and select a host for the component from the list that appears. The list displays hosts added to the cluster. To add a new host to the cluster, go to the ***Hosts*** page.



Add a host to a component

After the host is added, click ***Next***.

Reconfigure & Restart

The ***Reconfigure & Restart*** action updates service configuration files with the configuration settings specified on the service ***Primary configuration*** tab. If a changed parameter requires a restart, the action restarts the service. If not, the ***Reconfigure & Restart*** action only reloads configuration.

If you make changes directly to the postgresql.conf or pg_hba.conf file, these changes will be lost.

In the action's confirmation window, you can enable the ***Service user password reset*** option. This option allows you to regenerate passwords for the `adpg_replicator` and `adpg_rewind` system Patroni users. It can be useful if you need to update passwords for security reasons, for example, in case of password leaks.

You can also activate the ***Show advanced*** toggle button to display the ***Force reconfigure and restart*** checkbox that disables validation of the configuration settings before restarting ADPG.

The ***Force reconfigure and restart*** option should be used when:

- The ADPG service is stopped and cannot be started due to misconfiguration. In this case, configuration settings will be replaced with the default values.
- You have changed parameters of the ***Patroni ADPG configurations*** section that require the restart of the Patroni service, for example, `patroni_log_dir`.

The 'Run an action" window of Reconfigure & Restart

Reinstall

The ***Reinstall*** action reinstalls the service. It checks installed files and overwrites files with incorrect content. The ***Reinstall*** action also installs missing files.

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation.

The 'Run an action" window

Start

The ***Start*** action starts a service.

Stop

The ***Stop*** action stops a service.

Service actions of the ADPG Control Agents service

Eugenia Kuzina

Contents

To Table of Contents

Add components

Check

Reinstall

Restart

Start

Stop

Uninstall

This article describes the actions available for the ADPG Control Agents service.

Add components

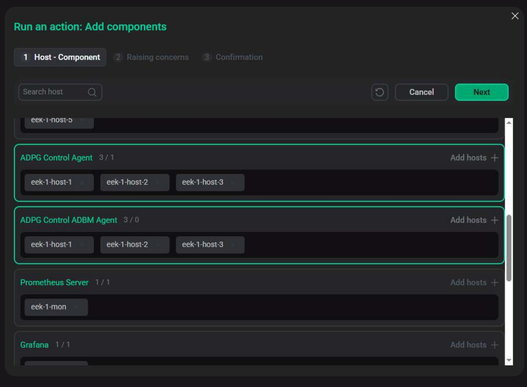
The ***Add components*** action installs components of the ADPG Control Agents service on new hosts.

IMPORTANT

If you are using ADPG Control for the ADPG cluster, it is necessary to run the actions for services in the following order:

1. the ***Add/Remove components*** action for the Monitoring service of the ADPG cluster;
2. the ***Add components*** action for the ADPG Control Agents service of the ADPG cluster;
3. the ***Expand*** action for the ADPG service of the ADPG cluster;
4. the Apply backup config action of the Backup manager of the ADPG Control cluster;
5. the ***Reconfigure and restart*** action for the Monitoring service of the ADPG cluster.

When you select this action, ADCM opens the ***Run an action*** window, where you can add new hosts on which you need to install ADPG Control Agents components.



The "Add components" action's window

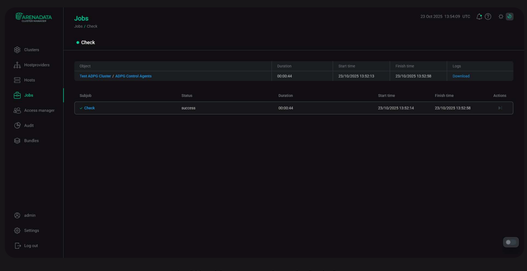
Click ***Add hosts*** to select a host to install the corresponding component. Note that the host must belong to the current cluster. You can add new hosts to the cluster on the Hosts page.

After you add components, click ***Next*** located at the top of the window.

Check

The ***Check*** action verifies whether the settings of hosts, components, and related services match the requirements of the service, and whether the service itself works correctly.

You can find the action result and the process of its execution on the Jobs page.



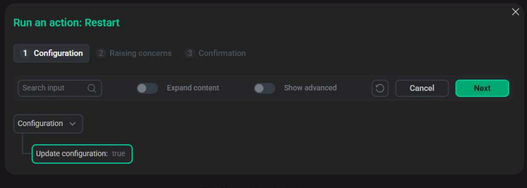
The "Check" action result on the "Jobs" page

Reinstall

The ***Reinstall*** action reinstalls the service. It checks installed files and overwrites files with incorrect content. The ***Reinstall*** action also installs missing files.

Restart

The ***Restart*** action restarts a service. If you set `true` to the ***Update configuration*** field in the confirmation window, service configuration parameters will be updated with values specified in the ADCM UI.



The "Restart" action's window

If ***Update configuration*** is set to `false`, the service will only restart with the parameters set previously.

Start

The ***Start*** action starts a service.

Stop

The ***Stop*** action stops a service.

Uninstall

The ***Uninstall*** action uninstalls the service.

Balancer service actions

Eugenia Kuzina

Contents

To Table of Contents

Check

Expand

Reconfigure & Restart

Reinstall

Start

Stop

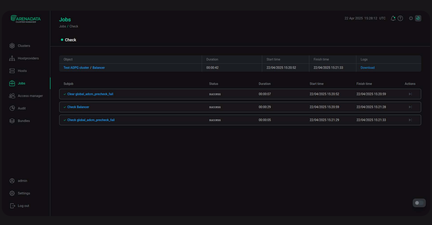
Uninstall

This article describes the actions available for the Balancer service.

Check

The ***Check*** action verifies whether the settings of hosts, components, and related services match the requirements of the service, and whether the service itself works correctly.

You can find the action result and the process of its execution on the Jobs page.



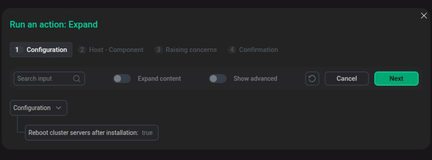
The "Check" action result on the "Jobs" page

Expand

The ***Expand*** action adds new hosts to a service.

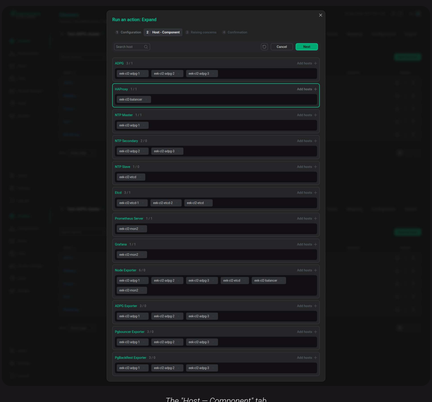
> **NOTE**
>
> If you use the Monitoring service, call the ***Reconfigure and restart*** action of the Monitoring service after the ***Expand*** action is finished to ensure that metrics are collected from added hosts.

When you choose ***Expand***, ADCM displays a dialog window to specify settings for the action.



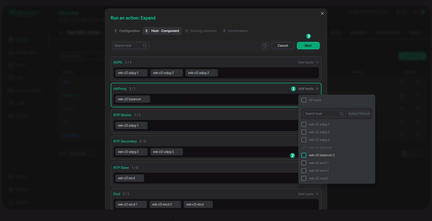
The "Run an action" window

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation. Click ***Next*** to open the ***Host — Component*** tab where you can add cluster hosts to the component.



The "Host — Component" tab

Components of the service for which the action is executed are highlighted in green. Click ***Add hosts*** and select a host for the component from the list that appears. The list displays hosts added to the cluster. To add a new host to the cluster, go to the ***Hosts*** page.



Add a host to a component

After the host is added, click ***Next***.

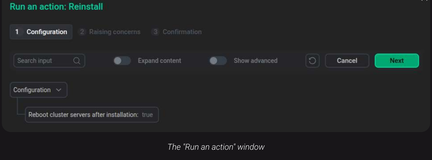
Reconfigure & Restart

The ***Reconfigure & Restart*** action updates service configuration files with the configuration settings specified on the service ***Primary configuration*** tab. If a changed parameter requires a restart, the action restarts the service. If not, the ***Reconfigure & Restart*** action only reloads configuration.

Reinstall

The ***Reinstall*** action reinstalls the service. It checks installed files and overwrites files with incorrect content. The ***Reinstall*** action also installs missing files.

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation.



The "Run an action" window

Start

The ***Start*** action starts a service.

Stop

The ***Stop*** action stops a service.

Uninstall

The ***Uninstall*** action uninstalls the service.

Chrony service actions

Eugenia Kuzina

Contents

To Table of Contents

Check

Expand

Reconfigure & Restart

Reinstall

Start

Stop

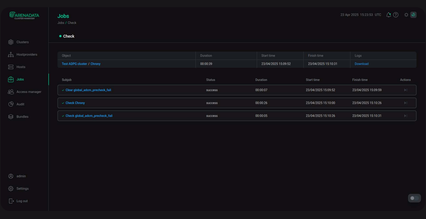
Uninstall

This article describes the actions available for the Chrony service.

Check

The ***Check*** action verifies whether the settings of hosts, components, and related services match the requirements of the service, and whether the service itself works correctly.

You can find the action result and the process of its execution on the Jobs page.



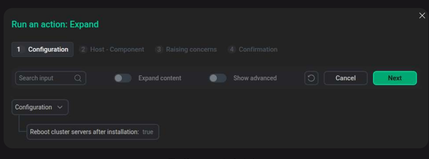
The "Check" action result on the "Jobs" page

Expand

The ***Expand*** action adds new hosts to a service.

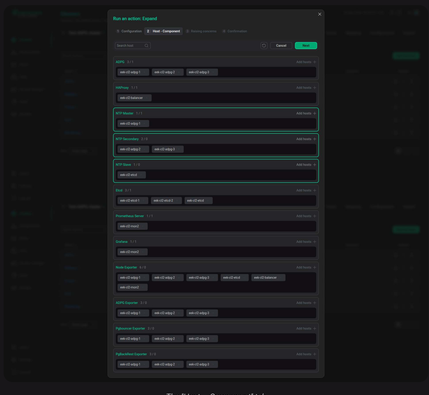
> **NOTE**
>
> If you use the Monitoring service, call the ***Reconfigure and restart*** action of the Monitoring service after the ***Expand*** action is finished to ensure that metrics are collected from added hosts.

When you choose ***Expand***, ADCM displays a dialog window to specify settings for the action.



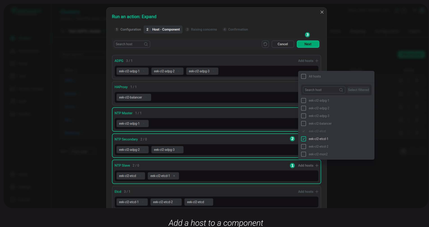
The "Run an action" window

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation. Click ***Next*** to open the ***Host — Component*** tab where you can add cluster hosts to the component.



The "Host — Component" tab

Components of the service for which the action is executed are highlighted in green. Click ***Add hosts*** and select a host for the component from the list that appears. The list displays hosts added to the cluster. To add a new host to the cluster, go to the ***Hosts*** page.



Add a host to a component

After the host is added, click ***Next***.

Reconfigure & Restart

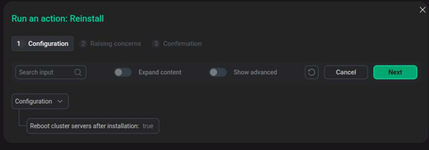
The ***Reconfigure & Restart*** action updates service configuration files with the configuration settings specified on the service ***Primary configuration*** tab. If a changed parameter requires a restart, the action restarts the service. If not, the ***Reconfigure & Restart*** action only reloads configuration.

If you make changes directly to the *chrony.conf* file, these changes will be lost.

Reinstall

The ***Reinstall*** action reinstalls the service. It checks installed files and overwrites files with incorrect content. The ***Reinstall*** action also installs missing files.

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation.



The "Run an action" window

Start

The ***Start*** action starts a service.

Stop

The ***Stop*** action stops a service.

Uninstall

The ***Uninstall*** action uninstalls the service.

Etcd service actions

Eugenia Kuzina

Contents

To Table of Contents

Check

Expand

Reconfigure & Restart

Reinstall

Start

Stop

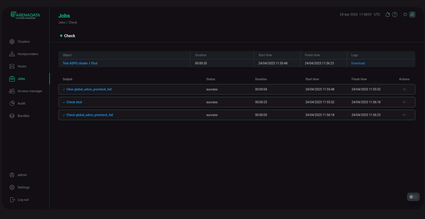
Uninstall

This article describes the actions available for the Etcd service.

Check

The ***Check*** action verifies whether the settings of hosts, components, and related services match the requirements of the service, and whether the service itself works correctly.

You can find the action result and the process of its execution on the Jobs page.



The "Check" action result on the "Jobs" page

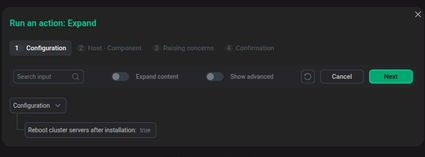
Expand

The ***Expand*** action adds new hosts to a service. The number of hosts with the Etcd service should remain odd.

NOTE

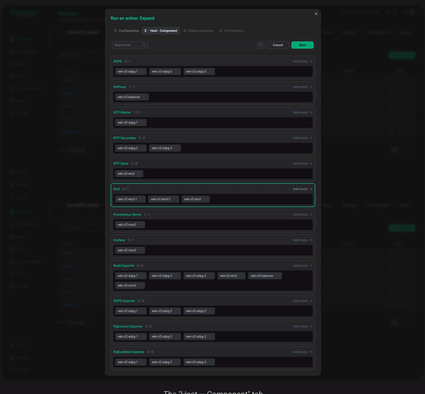
- After you add nodes to expand the Etcd service, it is necessary to execute the ***Reconfigure & Restart*** action of the ADPG and Balancer services.
- If you use the Monitoring service, call the ***Reconfigure and restart*** action of the Monitoring service after the ***Expand*** action is finished to ensure that metrics are collected from added hosts.

When you choose ***Expand***, ADCM displays a dialog window to specify settings for the action.



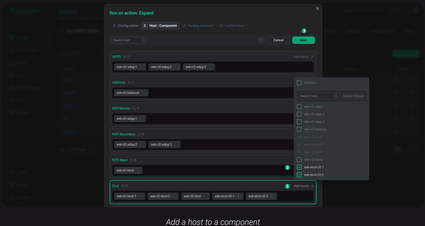
The "Run an action" window

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation. Click ***Next*** to open the ***Host — Component*** tab where you can add cluster hosts to the component.



The "Host — Component" tab

Components of the service for which the action is executed are highlighted in green. Click ***Add hosts*** and select a host for the component from the list that appears. The list displays hosts added to the cluster. To add a new host to the cluster, go to the ***Hosts*** page.



Add a host to a component

After the host is added, click ***Next***.

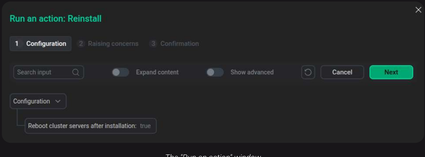
Reconfigure & Restart

The ***Reconfigure & Restart*** action updates service configuration files with the configuration settings specified on the service ***Primary configuration*** tab. If a changed parameter requires a restart, the action restarts the service. If not, the ***Reconfigure & Restart*** action only reloads configuration.

Reinstall

The ***Reinstall*** action reinstalls the service. It checks installed files and overwrites files with incorrect content. The ***Reinstall*** action also installs missing files.

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation.



The "Run an action" window

Start

The ***Start*** action starts a service.

Stop

The ***Stop*** action stops a service.

Uninstall

The ***Uninstall*** action uninstalls the service.

Monitoring service actions

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

Add/Remove components

Check

Reconfigure and restart

Reinstall

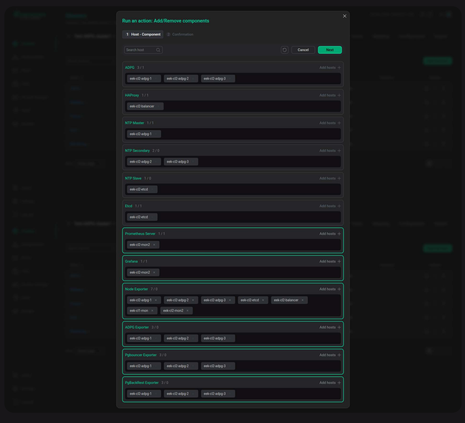
Start

Stop

This article describes the actions available for the Monitoring service.

Add/Remove components

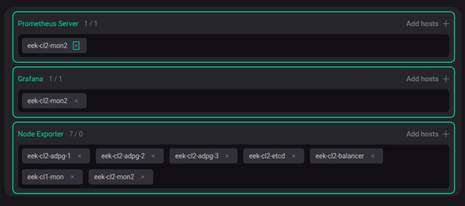
This action opens the component-host mapping interface, where you can add and remove components of the Monitoring service.



The component-host mapping interface

Only components of the Monitoring service are available for editing.

You can click ***Add hosts*** and select a host for the component from the list that appears. It is also possible to remove hosts from a component if this component is not required. If the host can be deleted, it will display a gray cross on the right side.



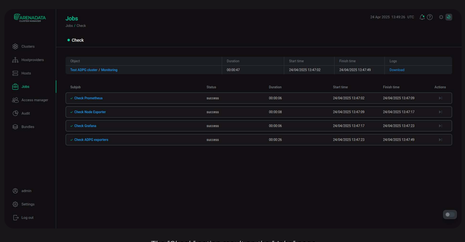
Components that can be removed

After you redistribute components, click ***Next*** located at the top of the window.

Check

The ***Check*** action verifies whether the settings of hosts, components, and related services match the requirements of the service, and whether the service itself works correctly.

You can find the action result and the process of its execution on the Jobs page.



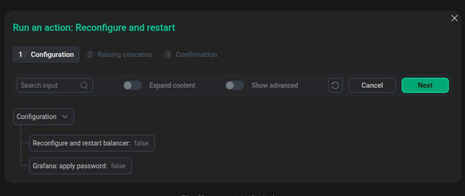
The "Check" action result on the "Jobs" page

Reconfigure and restart

The ***Reconfigure and restart*** action updates service configuration files with the configuration settings specified on the service ***Primary configuration*** tab. If a changed parameter requires a restart, the action restarts the service. If not, the ***Reconfigure and restart*** action only reloads configuration.

When the action is executed for the Monitoring service, you can set the following options:

- ***Reconfigure and restart balancer*** — whether to reconfigure and restart the Balancer service.
- ***Grafana: apply password*** — whether the Grafana administrator password should be reset using the value of the ***Grafana settings → Grafana administrator's password*** configuration parameter located on the ***Primary configuration*** tab of the Monitoring service.

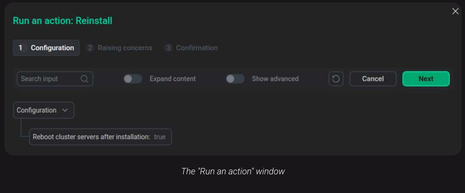


The "Run an action" window

Reinstall

The ***Reinstall*** action reinstalls the service. It checks installed files and overwrites files with incorrect content. The ***Reinstall*** action also installs missing files.

In the ***Run an action*** dialog window, you can choose whether ADCM reboots cluster servers after installation.



The "Run an action" window

Start

The ***Start*** action starts a service.

Stop

The ***Stop*** action stops a service.

ADPG monitoring metrics

Contents

To Table of Contents

- ADPG Checkpointer / Bgwriter / Block IO Stats
- ADPG DB overview
- ADPG Health check
- ADPG Replication
- ADPG Sessions overview
- Elvid Cluster overview
- Global ADPG dashboard
- PgBackRest statistics
- Pgbouncer statistics
- Postgres statistics
- Postgres SQL Patroni overview

ADPG monitoring metrics are collected by [Telegraf](https://www.influxdata.com/time-series-platform/telegraf/) and stored in [InfluxDB](https://www.influxdata.com/products/influxdb-overview/) time series database. [Grafana](https://grafana.com/) is used to create the dashboards. The current metrics cover the stats of the most used tools in the [ADPG](https://adpg.pigsty.io/) ecosystem and [Pigsty](https://pigsty.io/) distribution. Each dashboard is designed to provide a specific view of the PostgreSQL database, with detailed explanations of each panel and row, making it easy to understand and monitor your PostgreSQL cluster effectively.

Grafana Welcome screenshot

To monitor Grafana, the [Grafana](https://grafana.com/) dashboard is used.

Grafana Dashboard screenshot

The page with the dashboard list and a dropdown shows the **ADPG Dashboard**. We can find quickly a display scrambled dashboard.

ADPG Dashboard screenshot

You can find the following dashboards in Grafana:

- [ADPG Checkpointer / Bgwriter / Block IO Stats](#adpg-checkpointer--bgwriter--block-io-stats) - Statistics of checkpoints and bgwriter
- [ADPG DB overview](#adpg-db-overview) - Graphs for ADPG main characteristics
- [ADPG Health check](#adpg-health-check) - Main information of a status check for ADPG
- [ADPG Replication](#adpg-replication) - Statistics replication and streams
- [ADPG Sessions overview](#adpg-sessions-overview) - Statistics and graphs for sessions
- [Elvid Cluster overview](#elvid-cluster-overview)
- [Global ADPG dashboard](#global-adpg-dashboard) - Statistics and charts
- [PgBackRest statistics](#pgbackrest-statistics) - Statistics and graphs for the system views and is supported to the first version using the pgBackRest client commands.
- [Pgbouncer statistics](#pgbouncer-statistics) - Statistics and graphs for Pgbouncer
- [Postgres statistics](#postgres-statistics) - Statistics and graphs for the system views and the Postgres Exporter component.
- [Postgres SQL Patroni overview](#postgresql-patroni-overview) - Patroni cluster info.
- [Pgbouncer statistics](#pgbouncer-statistics) - Pgbouncer statistics.
- [Postgres statistics](#postgres-statistics) - Statistics and graphs.

ADPG Checkpointer / Bgwriter / Block IO Stats

This dashboard contains statistics of checkpoints and bgwriter.

ADPG Checkpointer / Bgwriter dashboard

ADPG Checkpointer / Bgwriter / Block IO Stats includes the following graphs:

- **Checkpoints** - Represents the number of checkpoints and the aggregated period.
- **Buffers** - Represents the **Total Buffers**.
- **Buffers Clean** - Represents the **BUFFERS_CHECKPOINT**, **BUFFERS_CLEAN** and **BUFFERS_BACKEND** values.
- **Block IO Read** - Represents the **BLK_READ_TIME** and **BLK_WRITE_TIME** values and the **TOTAL_BLK_TIME** system vue table in **GB**.
- **Buffers Alloc** - Represents the **Buffers Allocated** value.

ADPG DB overview

This dashboard contains graph graphs for ADPG main characteristics and helps to analyze the weaknesses of a particular node.

The ADPG DB overview dashboard includes the following charts:

- **Instance stats** - Instance stat info.
- **Instance uptime**
- **TPS** - Transactions per second.
- **QPS** - Queries per second.
- **Active backends** - Average query quantity.
- **DB size on TPS** - TPS size is calculated for each hour.
- **Active / Idle Ratio**
- **Active / Idle Count**
- **TPS / QPS Avg**
- **Idle / Idle Avg / Idle Max**
- **Temporal file (total Avg)**
- **Temp. file / Query / CONN (Max)**
- **Temp bytes** - It appears when large groups and sorting operations require more memory than the **WORK_MEM** value.

ADPG Health check

This dashboard displays the health check of a specific database, which can be selected in the **database** filter at the top of the page.

The ADPG Health check dashboard includes the following charts:

- **Instance stats**
- **Instance uptime**
- **Is in recovery system**
- **Longest query runtime**
- **Active connections**
- **Max connections**
- **Max prepared xacts**
- **Is the database hot stand**
- **Is the database up (bool)**
- **XIDAge**
- **MXIDAge**
- **Age of the oldest**
- **Is the database connection allowed**
- **Autovacuum count**
- **Checkpoints requested**
- **Backend - Max time**
- **Wal_sync_time(Avg)**
- **Temp. bytes per second(Avg)**
- **Longest wal (Total WAL bytes)**
- **Wal_write_time (Avg / Max)**
- **WRITE/s per second (Avg)**
- **WRITE/s per second (Max)**
- **Wal replication lag**
- **Max_wal_size / WAL Size**
- **Disk - WRITE/s per sec**
- **Checkpoint age**
- **Checkpoint - Max time**
- **Checkpoint - Max time**
- **Checkpoint - Max time

ADPG high availability overview

Eugenia Kuzina

Contents

To Table of Contents

Replication modes
- Asynchronous mode
- Synchronous mode

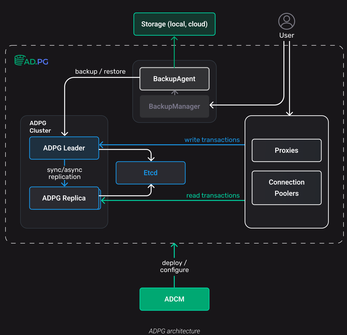
The etcd cluster

NOTE

The high availability (HA) feature is available for ADPG Enterprise Edition.

Database servers can work together to be able to switch quickly to another server in case of a primary server failure. This behavior is called **high availability (HA)**. ADPG uses Patroni to build the cluster based on the PostgreSQL streaming replication and implement HA. An ADPG cluster that supports HA should contain several ADPG nodes.

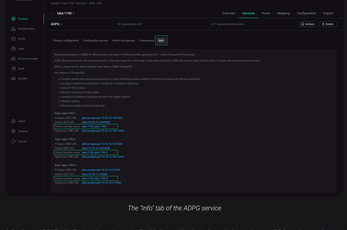
Each node consists of a PostgreSQL server and a Patroni agent that serves the local PostgreSQL instance. One ADPG node is the **leader**, others are **replica** instances. The leader serves read-write transactions (unless otherwise specified in load balancing settings), and replicas process only read-only requests. PostgreSQL streaming replication is a method that transfers WAL records from a leader to replicas. In case of the leader failure, Patroni elects a new leader from replicas and the ADPG cluster continues to operate.



ADPG architecture

The Patroni settings available from ADCM are described in Configuration settings.

The **Info** tab of the ADPG service displays the names of the Patroni cluster members, as well as links: `Postgres JDBC URL`, `Patroni REST API`, and `Pgbouncer JDBC URL`.



The "info" tab of the ADPG service

You can enable ***Patroni basic auth*** for Patroni REST API. To do this, activate the ***Enable Patroni basic auth*** toggle button in the ADPG service configuration settings and run the ***Reconfigure & Restart*** action of the ADPG service.

After enabling ***Patroni basic auth***, unsafe (POST) calls to Patroni are authenticated, but safe (GET) calls will be processed without authentication. Examples:

Auth is disabled, no password is required:

```
$ curl -X POST http://10.92.43.184:8008/reload
```

```
reload scheduled
```

Auth is enabled, no password is specified:

```
$ curl -X POST http://10.92.43.184:8008/reload
```

```
no auth header received
```

Auth is enabled, the password is incorrect:

```
$ curl -X POST -u "admin:wrongpassword" http://10.92.43.184:8008/reload
```

```
not authenticatedpostgres
```

Auth is enabled, the password is correct:

```
$ curl -X POST -u "admin:12345678" http://10.92.43.184:8008/reload
```

```
reload scheduled
```

Replication modes

The streaming replication has two modes: the asynchronous mode, which is used by default, and the synchronous mode.

Asynchronous mode

In asynchronous mode, a leader executes requests immediately, while changes from WAL are transmitted to replicas separately. If the leader crashes, uncommitted transactions may not be replicated, causing data loss.

The `maximum_lag_on_failover` parameter sets the amount of transaction log that can be lost in bytes. Exceeding this parameter does not affect the asynchronous replication process, but replicas with a lag greater than `maximum_lag_on_failover` do not participate in the leader election. You can change `maximum_lag_on_failover` on the ***Primary configuration*** tab of the ADPG service. To specify it, open ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** as described in Configure services and expand the ***Patroni ADPG configurations*** node, which contains Patroni settings.

The amount of lost data can be greater. In the worst case, it can be calculated as `maximum_lag_on_failover` bytes of the transaction log plus the amount that is written in the last `ttl` seconds. However, a typical replication delay is less than a second.

If the leader fails or becomes unavailable for any other reason, Patroni automatically promotes a replica to the leader. After this operation, it is impossible to recover any not replicated transactions.

The advantage of the asynchronous replication mode is the fast transaction commit, you do not need to wait for all replicas to apply the changes.

Synchronous mode

The synchronous replication ensures that all changes made by a transaction are transferred to one or more replica nodes. Each commit of a write transaction waits for a confirmation that the commit is written to WAL on the disk of the leader and replica hosts. The synchronous replication provides a higher level of durability, but it also increases the response time for the requesting transaction. The minimum wait time is the round-trip time between a leader and a replica.

The following transactions do not wait for a response from replicas:

- read-only transactions;
- transaction rollbacks;
- subtransaction commits.

Long-running actions such as data loading or index building do not wait for the final commit message.

To turn on Patroni synchronous mode, select the `synchronous_mode` checkbox on the ***Primary configuration*** tab of the ADPG service. To do this, open ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** as described in Configure services.

In this mode, Patroni can promote a replica only if the replica contains all transactions that returned a successful commit status to a client. However, Patroni synchronous mode does not guarantee that commits will be stored on multiple nodes under all circumstances. When no suitable replica is available, the leader still provides write access but does not guarantee replication. If the leader fails, no replica is promoted. When the host that used to be the leader comes back, Patroni promotes it to the leader automatically, unless a system administrator changes the leader manually. This behavior allows you to use the synchronous mode with clusters that consist of two ADPG nodes.

If a replica crashes, commits are blocked until Patroni runs and switches the leader to standalone mode. In the worst case, the delay is `ttl` seconds. Manually shutting down or restarting a replica does not cause a commit service interruption. The replica signals the leader to release itself from synchronization before the PostgreSQL shutdown.

To guarantee that each record is stored on at least two nodes, enable `synchronous_mode_strict` in addition to `synchronous_mode`. You can enable this option on the ***Primary configuration*** tab of the ADPG service. This parameter prevents Patroni from switching off the synchronous replication on the leader when no replicas are available. As a result, the leader does not provide write access and blocks all client write requests until at least one replica becomes available.

NOTE

Because of the way synchronous replication is implemented in PostgreSQL it is still possible to lose transactions even when the `synchronous_mode_strict` is used. If the PostgreSQL backend is canceled while waiting to acknowledge a replication (as a result of a packet cancellation due to client timeout or backend failure) transaction changes become visible for other backends. Such changes are not yet replicated and may be lost in case of replica promotion.

Patroni uses the `synchronous_node_count` parameter to manage the number of synchronous standby databases. Its default value is `1`. This parameter has no effect when `synchronous_mode` is disabled. When `synchronous_mode` is enabled, Patroni manages the precise number of synchronous standby databases based on `synchronous_node_count`. For details, see replication modes.

You can store Patroni log files to a custom directory. To set up custom logging, open the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** tab, activate the ***Show advanced*** toggle button, and expand the ***Patroni ADPG configurations*** node. Set the ***use_custom_patroni_log_dir*** parameter to `true` and specify a path to a custom directory.

The etcd cluster

Patroni saves the cluster configuration in the Distributed Configuration Storage (DCS). ADPG requires an etcd cluster to use it as Patroni DCS. The `listen_peer_urls_port` and `listen_client_urls_port` etcd settings must be specified during installation, after installation they cannot be changed.

You can use the built-in ADPG Etcd service or an external etcd cluster. If you wish to utilize the built-in Etcd service, refer to the Add services article that describes how to add services. You must use an odd number of nodes for the Etcd service. Three or more nodes are recommended. Nodes for Etcd must not contain the ADPG service. Note, after you add nodes to expand the working Etcd service, it is necessary to execute the ***Reconfigure & Restart*** action of the ADPG and Balancer services.

Etcd keeps the whole log on the disk until the compaction procedure is performed. This may result in the lack of the disk space. To avoid this problem, the autocompaction is enabled in ADPG. The default `Autocompaction mode` value is `periodic`, `Autocompaction retention` – `10h`. The max number of bytes the etcd db file can consume (the `Space quota` parameter) is set to `2147483648` bytes. You can change these settings on the ***Clusters → ADPG cluster → Services → Etcd → Primary configuration*** page.

The defragmentation of the disk space is required to return the disk space from the etcd backend to the host file system. It is recommended to schedule the defragmentation (e.g. with cron). Note, that the defragmentation blocks the cluster nodes, and they become unavailable.

The defragmentation command for the whole cluster (it blocks the whole cluster for a period of time):

```
$ etcdctl defrag --cluster
```

For the local node (it blocks only the local node for a period of time):

```
$ etcdctl defrag
```

If SSL is enabled:

```
$ /usr/lib/adpg16/usr/bin/etcdctl defrag --cluster --cacert="/etc/adpg/ssl/etcd/ca.crt" \
--cert="/etc/adpg/ssl/etcd/client.crt" --key="/etc/adpg/ssl/etcd/client.key"
```

If you want to use an external etcd cluster, set external Etcd settings on the ***Primary configuration*** tab of the ADPG service. To specify them, open ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** as described in Configure services.

TIP

You can use the ***Expand*** action to add new nodes to the ADPG cluster and built-in Etcd service.

Logical replication

Eugenia Kuzina

Contents

To Table of Contents

- Overview
- Configuration settings
- Publication
- Subscription
- Conflicts
- Security

Overview

Logical replication is a replication of data objects and their changes based on their replication identity (usually a primary key). We use the term "logical replication" as opposed to physical replication, which uses block addresses and byte-by-byte replication. Logical replication allows you to control both data replication and security.

The logical replication uses a **publish/subscribe** model with one or more **subscribers** that subscribe to one or more **publications** on the **publishing node**. Subscribers receive data from the publications and can republish the data for cascading replication or more complex configurations.

The logical replication of a table starts with taking a snapshot of the data in the publisher database and copying it to the subscriber. For more information, see Architecture. After that, the changes on the publisher are sent to the subscriber in real-time as they occur. The subscriber applies changes in the same order as the publishing node. Publications within the same subscription are guaranteed transactional integrity. The subscriber applies data changes in the same order as the publisher so that transactional consistency is guaranteed for publications within a single subscription. This method of data replication is also known as transactional replication.

The typical use cases for logical replication are:

- sending incremental changes in a single database or a subset of databases to subscribers;
- firing triggers for individual changes when a subscriber receives them;
- combining multiple databases into one (for example, for analysis purposes);
- replicating between different PostgreSQL major versions;
- replicating between PostgreSQL instances on different platforms;
- providing access to replicated data to different groups of users;
- sharing a database subset across multiple databases.

The subscriber database behaves in the same way as any other PostgreSQL instance and can be a publisher if you create publications in it. When a subscriber acts as a read-only application, there are no conflicts with a single subscription. If any applications or other subscribers write to the same set of tables, conflicts can occur.

For detailed information about restrictions of the logical replication, refer to Restrictions.

To set up a logical replication, perform the following steps:

- Configure the publishing and subscriber nodes.
- Create publication.
- Create subscription.

Configuration settings

The logical replication requires several configuration options to be set. Use the ***ADPG configurations*** section on the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** tab in ADCM UI to specify these options. See Configuration parameters for details.

Specify the following parameters on the publisher side:

- wal_level must be set to `logical`.
- max_replication_slots must be set to at least the number of subscriptions expected to connect, plus some reserve for table synchronization.
- max_wal_senders should be set to at least the `max_replication_slots` value plus the number of physical replicas that are connected at the same time.

Set the following parameters on the subscriber side:

- max_replication_slots should be set to at least the number of subscriptions added to the subscriber, plus some reserve for table synchronization.
- max_logical_replication_workers should be also set to at least the number of subscriptions added to the subscriber, plus some reserve for table synchronization.
- max_worker_processes should include additional worker processes for the replication, at least `max_logical_replication_workers` plus 1. Note, some extensions and parallel queries also occupy slots from `max_worker_processes`. Unlike other parameters, you can set this parameter in the ***ADPG configurations*** section on the ***Primary configuration*** tab. See Configuration parameters.

You also need to check if the settings specified in the ***PG_HBA*** section on the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** tab in ADCM UI allow the replication. These settings depend on your network configuration and the user you want to utilize for the connection.

Publication

A publication can be defined on any physical replication primary. A node that contains a publication is called a **publisher**. A publication is a set of changes generated from a table or a group of tables. It is also can be described as a **replication set**. Each publication exists in only one database.

Publications are different from schemas and do not affect the table access. You can include each table in multiple publications. Currently, publications can only contain tables. Objects must be explicitly added to them unless the publication is created for all tables with the `ALL TABLES` specification.

Publications can limit the set of changes they contain. It is possible to choose any combination of the `INSERT`, `UPDATE`, `DELETE`, and `TRUNCATE` operations, similar to how triggers can fire on different event types. By default, all types of operations are replicated.

To replicate the `UPDATE` and `DELETE` operations, the published table must have a **replica identity**. It allows the system to find appropriate rows to update or delete them on the subscriber side. By default, this is the primary key. You can also specify another unique index for a replica identity. If the table does not have a suitable key, the entire row can be used as a key. To do this, set the replica identity to `full`. However, this approach is very inefficient and should be used if there is no other solution. If a replication identifier other than `full` is selected on the publisher side, an identifier consisting of the same or fewer columns must also be defined on the subscriber side. For more information about assigning a replica identity, see REPLICA IDENTITY. If a table without a replica identity is added to a publication, the replicated `UPDATE` or `DELETE` operation causes an error on the publisher. The `INSERT` operations can be processed without the specified replica identity.

Every publication can have multiple subscribers.

Use the CREATE PUBLICATION command to create a publication. The following command creates a publication that publishes all changes from the `books` and `orders` tables:

```
CREATE PUBLICATION publication1 FOR TABLE books, orders;
```

The code below creates a publication that publishes all changes from all tables:

```
CREATE PUBLICATION all_tables FOR ALL TABLES;
```

The following publication publishes the `DELETE` operations performed on the `books` table:

```
CREATE PUBLICATION delete_publication FOR TABLE books WITH (publish = 'delete');
```

You can use the ALTER PUBLICATION command to change the definition of a publication.

The command below changes the `delete_publication` publication to publish deletes and updates:

```
ALTER PUBLICATION delete_publication SET (publish = 'update, delete');
```

The following code adds the `authors` table to `publication1`:

```
ALTER PUBLICATION publication1 ADD TABLE authors;
```

Both the `ADD TABLE` and `DROP TABLE` operations are transactional. A table replication only starts or stops after the transaction is committed.

To remove a publication, call the DROP PUBLICATION command:

```
DROP PUBLICATION all_tables;
```

The pg_publication catalog contains all publications created in a database. The `pg_publication_tables` view provides information about the mapping between publications and the tables they contain. To retrieve this data, use the following commands:

```
SELECT * FROM pg_publication;

SELECT * FROM pg_publication_tables;
```

Subscription

A **subscription** is the downstream side of logical replication. The node on which the subscription is defined is called the **subscriber**. A subscription defines the connection to another database and set of publications to which the subscriber wants to subscribe.

A subscriber database works as any other PostgreSQL instance and can be used as a publisher for other databases if its own publications are defined.

A subscriber node can have multiple subscriptions. It is possible to define multiple subscriptions between a single publisher/subscriber pair. It is necessary to ensure that the publication objects do not overlap.

Each subscription receives changes via one replication slot. Additional replication slots may be required for the initial data synchronization. They are dropped after the data synchronization is finished. For more information, see Replication slot management.

A logical replication subscription can be a standby for synchronous replication.

Subscriptions can be archived when a dump-file is created. The `pg_dump` utility adds subscriptions to an SQL dump file, if it is run on behalf of a superuser. Otherwise, subscriptions are skipped, because non-superusers cannot read all subscription information from the pg_subscription catalog.

Use the CREATE SUBSCRIPTION command to add a subscription:

```
CREATE SUBSCRIPTION subscription1
CONNECTION 'host=10.92.6.56 port=5432 user=postgres dbname=book_store password=postgres'
PUBLICATION publication1, delete_publication;
```

Utilize the ALTER SUBSCRIPTION command to change a subscription. The following code changes the subscribed publication to `all_tables`:

```
ALTER SUBSCRIPTION subscription1 SET PUBLICATION all_tables;
```

The command below disables (stops) the subscription:

```
ALTER SUBSCRIPTION subscription1 DISABLE;
```

To remove a subscription, use DROP SUBSCRIPTION:

```
DROP SUBSCRIPTION subscription1;
```

When you drop and recreate a subscription, the synchronization information is lost. You need to resynchronize data after the recreation.

Only tables can be the target of replication. For example, you cannot replicate to a view.

The tables are matched between the publisher and the subscriber using the fully qualified table name. The replication to a table with a different name is not supported.

Table columns are also matched by name. The order of columns does not have to match. The data types of the columns do not need to match if a text representation of the data can be converted to the target type. For example, you can replicate from a column of type `integer` to a column of type `bigint`. The target table can also have additional columns that the published table does not contain. These columns are filled with default values specified in the target table definition.

Conflicts

The logical replication behaves similarly to DML operations, the data is updated even if it was changed locally on the subscriber node. If incoming data violates any constraints, the replication stops. This is considered as a **conflict**.

A conflict produces an error and stops the replication. You have to resolve it manually. You can find details about the conflict in the subscriber server log. The default path is */pg_data1/adcg16/log*.

Example:

```
2023-01-26 17:29:06.242 UTC [14525] LOG:  background worker "logical replication worker" (PID 1389) exited with exit code 1
2023-01-26 17:29:06.274 UTC [1390] ERROR:  duplicate key value violates unique constraint "test1_pkey"
2023-01-26 17:29:06.274 UTC [1390] DETAIL:  Key (employee_id)=(1) already exists.
2023-01-26 17:29:06.274 UTC [1390] CONTEXT:  COPY test1, line 1
```

To resolve the conflict, change the data on the subscriber to match the incoming change, or skip the problematic transaction. To omit the transaction, call the pg_replication_origin_advance function and pass a node name corresponding to the subscription name and a position as parameters. You can find the current position of origins in the pg_replication_origin_status system view.

When replicating an `UPDATE` or `DELETE` operation, missing data does not produce a conflict, and such an operation is skipped.

Security

If users can modify the schema of tables on the subscriber side, they also can execute arbitrary code as a superuser. Use roles to limit ownership and the `TRIGGER` privilege on these tables. Moreover, if untrusted users can create tables, use publications that list tables explicitly. Create the `FOR ALL TABLES` subscription only when you trust every user permitted to create a table on the publisher or subscriber.

The role used for the replication connection must have the `REPLICATION` attribute or be a superuser. If a role does not have the `SUPERUSER` or `BYPASSRLS` attribute, the publisher row security policies can execute. If a role does not trust all table owners, include `options=-crow_security=off` in the connection string. If a table owner adds a row security policy, this setting causes replication to halt rather than execute the policy. Use the ***PG_HBA*** section on the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** tab in ADCM UI to configure role access. A role for replication must have the `LOGIN` attribute.

To copy the initial table data, the role used for the replication connection must have the `SELECT` privilege on a published table or be a superuser.

To create a publication, the user must have the `CREATE` privilege in the database.

To add tables to a publication, the user must have ownership rights on the table. To create a publication that publishes all tables, the user must be a superuser.

To create a subscription, the user must be a superuser.

The process of applying subscription changes is performed in the local database with superuser rights.

Configure load balancing

Eugenia Kuzina

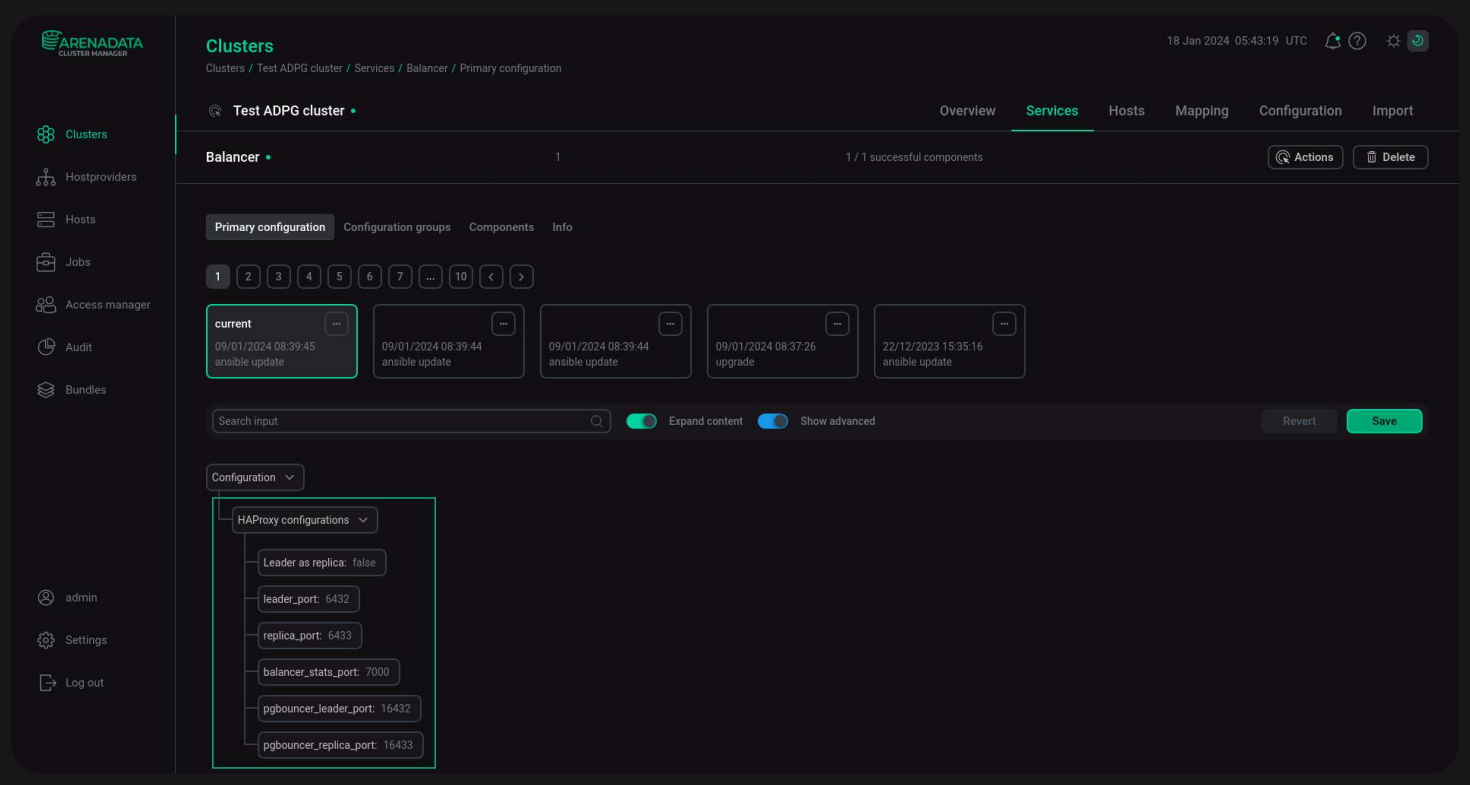
To Table of Contents

NOTE

The load balancing feature is available for ADPG Enterprise Edition.

The ADPG cluster implements load balancing, which distributes client requests across multiple database hosts. For more information on this functionality, see Load balancing.

The **Balancer** service performs balancing. To configure it, go to the ***Clusters*** page, select an ADPG cluster, and switch to the ***Services*** tab. Click the ***Balancer*** service and expand ***HAProxy configurations*** node on the ***Primary configuration*** tab (see Configure services).



Settings of the Balancer service

With the default settings, only the leader node processes write requests, and read only requests are passed to replicas. You can also allow the leader to handle read only transactions. To do this, activate the ***leader as replica*** checkbox.

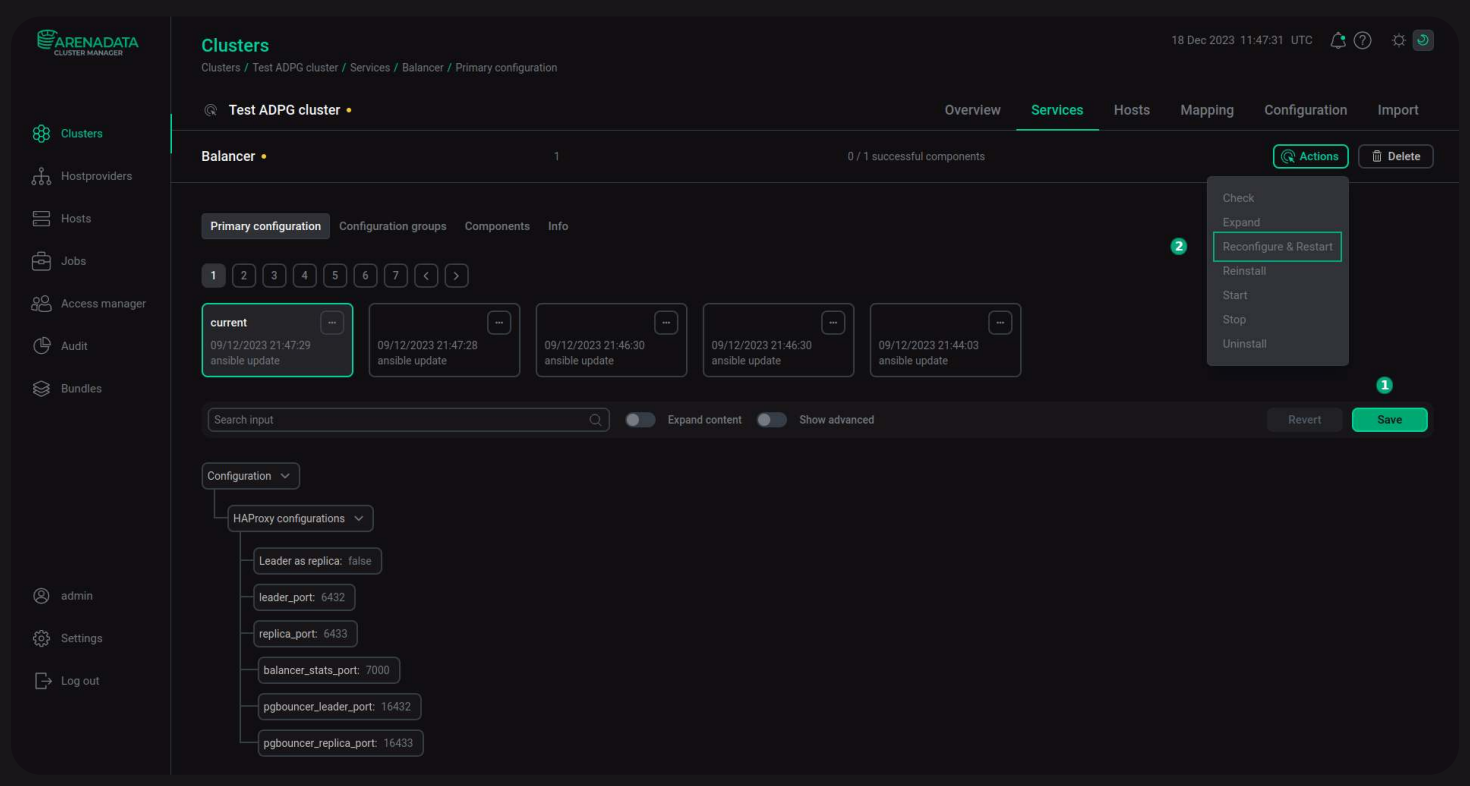
The ***leader_port*** option specifies a port on the host with the HAProxy component for write transactions. The default value is `6432`.

The ***replica_port*** option sets a port on the host with the HAProxy component for read only transactions. The default value is `6433`.

The ***balancer_stats_port*** option defines a port on which a web page with a HAProxy statistics report is available. The default value is `7000`.

The ***pgbouncer_leader_port*** and ***pgbouncer_replica_port*** options are used if PgBouncer is enabled. All connections to `pgbouncer_leader_port` are transferred to a PgBouncer instance of an ADPG leader node. The default `pgbouncer_leader_port` value is `16432`. All connections to `pgbouncer_replica_port` are transferred to PgBouncer instances of ADPG replica nodes. The default `pgbouncer_replica_port` value is `16433`. See Configure PgBouncer.

After you change the settings, click ***Save*** and execute the Reconfigure & Restart action. The ***Reconfigure & Restart*** action updates the service configuration files according to the configuration settings and restarts the service.



Run Reconfigure & Restart

Configure PgBouncer

Eugenia Kuzina

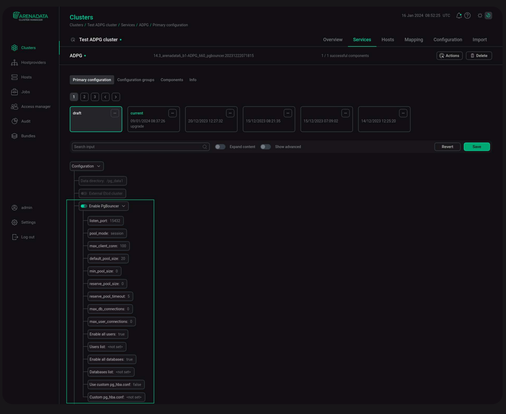
To Table of Contents

NOTE

The PgBouncer feature is available for ADPG Enterprise Edition.

ADPG includes the PgBouncer tool that is a lightweight connection pooler for PostgreSQL. An application can connect to PgBouncer as an ADPG server, and PgBouncer creates a connection to the ADPG server, or reuses one of its existing connections. PgBouncer is designed to reduce the performance impact of opening new PostgreSQL connections. For more information, see PgBouncer overview.

To enable PgBouncer and set its configuration parameters, open the ***Services*** tab of an ADPG cluster and select the ***ADPG*** service. On the ***Primary configuration*** tab, switch on the ***Enable PgBouncer*** toggle button. When this toggle button is enabled, you can expand the corresponding node and specify required settings.



PgBouncer settings

All available settings are listed in the table below.

Parameter name	Description	Default value
listen_port	The port that PgBouncer listens on. This port cannot be changed for a running cluster	15432
pool_mode	Specifies when a server connection can be reused by other clients. It can contain the following values: ■ `session` — a server is released back to the pool after a client disconnects. ■ `transaction` — a server is released back to the pool after a transaction finishes. ■ `statement` — a server is released back to the pool after a query finishes. Transactions that include multiple statements are prohibited in this mode.	session
max_client_conn	The maximum number of client connections allowed	100
default_pool_size	The number of connections to the server that are allowed for each user/database pair	20
min_pool_size	The minimum number of connections to the server to keep in the pool	0
reserve_pool_size	The number of additional connections allowed to the pool if a client connection exceeds the time specified in the `reserve_pool_timeout` option	0
reserve_pool_timeout	If a client needs to wait longer than the specified time in seconds, additional connections from `reserve_pool_size` are used	5
max_db_connections	The maximum number of connections to a server for a database	0
max_user_connections	The maximum number of connections to a server for a user	0
Enable all users	When enabled, this option allows access to all users with password authentication. If the option is disabled, utilize `Users list` to allow specified users to access ADPG with PgBouncer	Enabled
Users list	List of users who are allowed to access ADPG using PgBouncer, when the `Enable all users` option is disabled. For example: `postgres,user1`	Empty
Enable all databases	When enabled, this option allows access to all existing databases with PgBouncer. If the option is disabled, utilize `Databases list` to allow access to specified databases using PgBouncer	
Databases list	List of databases that are allowed to access using PgBouncer when the `Enable all databases` option is disabled. It has the same syntax as the databases PgBouncer section. For example, `postgres = host=localhost port=5432`	Empty
Use custom pg_hba.conf	When disabled, the general PG_HBA section of the ADPG service will be used for PgBouncer. If the option is enabled, it is necessary to add required settings to ***Custom pg_hba.conf***. In this case, PgBouncer will use settings from ***Custom pg_hba.conf***	Disabled
Custom pg_hba.conf	Custom pg_hba.conf section for PgBouncer	Empty

To apply changes, click ***Save***, and execute the Reconfigure & Restart action of the ***ADPG*** service.

In ADPG, PgBouncer utilizes the `hba` client authentication. If ***Use custom pg_hba.conf*** is disabled, the actual authentication type is loaded from the *pg_hba.conf* file. If ***Use custom pg_hba.conf*** is enabled, the settings from ***Custom pg_hba.conf*** is used.

By default, the ***Enable all users*** option is enabled. It allows access to all users with password authentication. You can disable this option and specify a list of users who are allowed to access ADPG using PgBouncer in the ***Users list*** parameter.

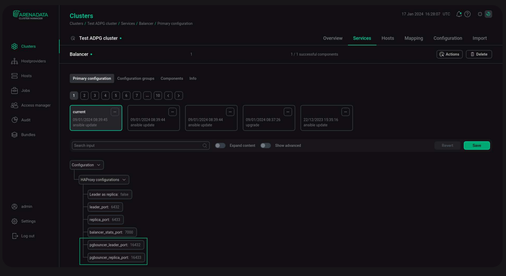
The ***Enable all databases*** option allows access to all existing databases with PgBouncer. You can disable this option and specify a list of databases that are allowed to access using PgBouncer in the ***Databases list*** parameter.

If the ***Balancer*** service is not added to the cluster, all the client requests should be sent to a PgBouncer `listen_port` on both leader and replicas nodes.

If your ADPG cluster uses the ***Balancer*** service, connections to the leader node should be sent to the balancer port specified in the `pgbouncer_leader_port` parameter of the ***Balancer*** service. All connections to this port are transferred to a PgBouncer instance of an ADPG leader node. The default `pgbouncer_leader_port` value is `16432`.

Connections to replica nodes should be sent to the balancer port specified in the `pgbouncer_replica_port` parameter of the ***Balancer*** service. All connections to this port are transferred to PgBouncer instances of ADPG replica nodes. The default `pgbouncer_replica_port` value is `16433`.

To change the `pgbouncer_leader_port` and `pgbouncer_replica_port` parameters, open the ***Services*** tab of an ADPG cluster and select the ***Balancer*** service. Specify the parameters on the ***Primary configuration*** tab.



Balancer settings

After you specify parameters, click ***Save*** and execute the Reconfigure & Restart action of the ***Balancer*** service to apply changes.

Switchover from leader to replica in ADPG

Eugenia Kuzina

To Table of Contents

As mentioned in ADPG architecture overview, ADPG uses Patroni to build a cluster based on the PostgreSQL streaming replication and implement High Availability (HA). If you need to switchover from the leader node of an ADPG cluster to a replica, you can do it manually via the patronictl switchover command.

This command can only be used in a healthy cluster that has:

- leader node;
- synchronized replica.

Follow the steps below to switchover a leader:

1. Run the Check cluster action to ensure that the cluster is healthy. If no errors are found, you can proceed to the steps below.
2. Connect to one of the ADPG service nodes under the `postgres` user:

```
$ sudo su - postgres
```

3. Execute the patronictl list command to display cluster members:

```
$ patronictl -c /etc/adpg16-patroni/adpg16-patroni.yml list
```

where `/etc/adpg16-patroni/adpg16-patroni.yml` is the path to the Patroni configuration file.

Result:

```
+ Cluster: test_adpg_cluster (7397708120470393282) ---------+---------+----+-----------+
| Member                           | Host         | Role    | State   | TL | Lag in MB |
+----------------------------------+--------------+---------+---------+----+-----------+
| test_adpg_cluster_eek-adpg-adpg1 | 10.92.40.49  | Leader  | running | 10 |           |
| test_adpg_cluster_eek-adpg-adpg2 | 10.92.42.110 | Replica | running | 10 |         0 |
| test_adpg_cluster_eek-adpg-adpg3 | 10.92.42.217 | Replica | running | 10 |         0 |
+----------------------------------+--------------+---------+---------+----+-----------+
```

4. Run the `patronictl switchover` command:

```
$ patronictl -c <path to patroni.yml> switchover --leader <current leader> --candidate <new leader>
```

Use the names of the cluster members from the list above as parameters:

```
$ patronictl -c /etc/adpg16-patroni/adpg16-patroni.yml switchover --leader test_adpg_cluster_eek-adpg-adpg1 --candidate
test_adpg_cluster_eek-adpg-adpg3
```

The following output will be displayed:

```
When should the switchover take place (e.g. 2024-08-02T15:28 )  [now]:
```

Press `Enter` to confirm that the leader should be switched immediately.

Type `y` in response to the following message:

```
Are you sure you want to switchover cluster test_adpg_cluster, demoting current master test_adpg_cluster_eek-adpg-adpg1? [y/N]:
```

Result:

```
2024-08-02 14:29:02.22919 Successfully switched over to "test_adpg_cluster_eek-adpg-adpg3"
+ Cluster: test_adpg_cluster (7397708120470393282) ---------+---------+----+-----------+
| Member                           | Host         | Role    | State   | TL | Lag in MB |
+----------------------------------+--------------+---------+---------+----+-----------+
| test_adpg_cluster_eek-adpg-adpg1 | 10.92.40.49  | Replica | stopped |    |   unknown |
| test_adpg_cluster_eek-adpg-adpg2 | 10.92.42.110 | Replica | running | 10 |         0 |
| test_adpg_cluster_eek-adpg-adpg3 | 10.92.42.217 | Leader  | running | 10 |           |
+----------------------------------+--------------+---------+---------+----+-----------+
```

5. Run the following command to make sure that the leader has been changed:

```
$ patronictl -c /etc/adpg16-patroni/adpg16-patroni.yml list
```

The `Role` column displays if a node is a leader or a replica:

```
+ Cluster: test_adpg_cluster (7397708120470393282) ---------+---------+----+-----------+
| Member                           | Host         | Role    | State   | TL | Lag in MB |
+----------------------------------+--------------+---------+---------+----+-----------+
| test_adpg_cluster_eek-adpg-adpg1 | 10.92.40.49  | Replica | running | 11 |         0 |
| test_adpg_cluster_eek-adpg-adpg2 | 10.92.42.110 | Replica | running | 11 |         0 |
| test_adpg_cluster_eek-adpg-adpg3 | 10.92.42.217 | Leader  | running | 11 |           |
+----------------------------------+--------------+---------+---------+----+-----------+
```

When the switchover command is executed, the timeline (the `TL` column) grows. Its increase shows that the operation was successful.

6. Run the Check cluster action to ensure that the cluster works correctly.

SQL dump

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

- Create a backup
- Restore the dump
- Dump a database cluster
- Backup large databases
  - Use compressed dumps
  - Use split
  - Use custom dump format
  - Use parallel dump feature

The dump technique allows you to create a file with SQL commands that recreates a database in the same state as it was at the time the dump was created.

Create a backup

PostgreSQL includes the pg_dump utility that implements this functionality. It does not require blocking access of other users and makes consistent backups when the database is in use.

The basic use of `pg_dump` is:

```
$ pg_dump <dbname> > <dumpfile>
```

Where:

- `<dbname>` is the database name to be backed up,
- `<dumpfile>` is a name of the backup file that will be created.

The `pg_dump` utility writes its result to the standard output. Dumps can be output in script or archive file formats (`custom`, `directory`, or `tar`). See the Options section of the pg_dump article for details. The command above creates a script text file.

Script dumps are plain text files that contain SQL commands. To restore a database from a script file, use `psql`. You can utilize script files to create backups and reconstruct the database on a machine with another architecture. It is also possible to manually modify script files to restore the database in another DBMS.

The `pg_dump` application backs up a single database. To back up an entire cluster or global objects that are common to all databases in a cluster (roles and tablespaces), use pg_dumpall.

Since `pg_dump` is a client application, you can create a backup from any remote host that has access to the database. It must have the read access to all tables that you want to back up. To back up an entire database, you almost always have to run it as a database superuser. If you do not have this privilege, you can back up database parts to which you have the read access. Use options `-n` (schema) or `-t` (table) as described in the following article: pg_dump.

The example below backs up the `books_store` database to the *books_store20221215.sql* file:

```
$ pg_dump books_store > books_store20221215.sql
```

The following example creates a backup of the `books` table from the `books_store` database:

```
$ pg_dump -d books_store -t books > books_20221215.sql
```

To specify which database server `pg_dump` should contact, use the command line options `-h` (host) and `-p` (port). The default host is localhost or the host specified in the `PGHOST` environment variable. The default port is set in the `PGPORT` environment variable. If `PGPORT` is not assigned, `pg_dump` uses the value specified during compilation. The default port value for the ADPG service is `5432`. The `pg_dump` program connects with the database user name that is equal to the current operating system user name. To override this, specify the `-U` option or set the `PGUSER` environment variable.

The following example backs up the `books_store` database located on the host `10.92.6.26`. `pg_dump` establishes the connection on behalf of the `postgres` user:

```
$ pg_dump -h 10.92.6.26 -U postgres books_store > books_store20221215.sql
```

The archive file formats are designed to be portable across architectures. See pg_dump for details.

> **NOTE**
>
> The `pg_dump` output can be re-loaded into newer versions of PostgreSQL, whereas file-level backups and continuous archiving are both version-specific.

Restore the dump

Use the psql program to restore dump files. The basic command syntax to restore a dump is:

```
$ psql <dbname> < <dumpfile>
```

Where:

- `<dbname>` is the database name to be restored,
- `<dumpfile>` is a name of the backup file.

The command above does not create the `dbname` database. You must create it from template0 before executing the command.

```
$ createdb -T template0 books_store
```

> **NOTE**
>
> The `pg_dump` utility creates dumps that are relative to `template0`. Languages, procedures, and others added via `template1` are also be dumped. If you utilize a customized `template1`, create an empty database from `template0`, as in the example above.

The `psql` utility continues the script execution after an SQL error occurs. You can use the `ON_ERROR_STOP` variable to alter this behavior and force the `psql` exit with the exit status `3` in case of SQL errors.

```
$ psql --set ON_ERROR_STOP=on books_store < books_store20221215.sql
```

Before restoring an SQL dump, all the users who own objects or have permissions on objects in the dumped database must already exist. If they do not exist, errors occur when restoring objects with the original ownership or permissions.

After restoring a backup, run ANALYZE on each database to provide the query optimizer with correct statistics.

For archive file dumps, use the pg_restore utility.

> **TIP**
>
> To improve the performance of dump file restoring, pay attention to the recommendations in the Improve pg_dump performance section.

Dump a database cluster

The `pg_dump` program backs up a single database. It does not dump information about roles or tablespaces because they are specified for a cluster rather than for a database. The `pg_dumpall` utility supports dumping of the database cluster contents. `pg_dumpall` backs up each database in a given cluster, and preserves cluster-wide data (role and tablespace definitions). It has the following syntax:

```
$ pg_dumpall > <dumpfile>
```

Where `<dumpfile>` is a name of the backup file that will be created.

The resulting dump can be restored with `psql`.

```
$ psql -f dumpfile postgres
```

The `-f` option means that `psql` reads commands from the specified file (`dumpfile`). `postgres` is a database name. You can specify any existing database name to start from instead of `postgres`, but if you restore the backup of multiple databases into an empty cluster, `postgres` should be used. It is also necessary to have the database superuser access to restore a `pg_dumpall` dump. If you use tablespaces, make sure that the tablespace paths in the dump are appropriate for the new installation.

The `pg_dumpall` program adds to the dump file commands to re-create roles, tablespaces, and empty databases, after that invokes `pg_dump` for each database. This means that while each database is internally consistent, the snapshots of different databases are not synchronized.

To dump only cluster-wide data, use the `pg_dumpall --globals-only` option.

Backup large databases

Operating systems may have maximum file size limits that cause problems when creating large `pg_dump` output files. Since `pg_dump` writes to the standard output, you can use standard Unix tools to solve this problem.

Use compressed dumps

You can use a compression program, for example gzip:

```
$ pg_dump <dbname> | gzip > <filename.gz>
```

You can restore the dump as follows:

```
$ gunzip -c <filename.gz> | psql <dbname>
```

or

```
$ cat <filename.gz> | gunzip | psql <dbname>
```

Use split

The split command allows you to split the output into smaller files. The following example splits a file into 2 GB chunks:

```
$ pg_dump <dbname> | split -b 2G - <filename>
```

You can restore the dump as follows:

```
$ cat <filename>* | psql <dbname>
```

If you utilize GNU split, it is possible to use it and gzip together:

```
$ pg_dump <dbname> | split -b 2G --filter='gzip > $FILE.gz'
```

Use the zcat utility to restore it:

Use custom dump format

Since ADPG includes the zlib library, the `custom` pg_dump format compresses data. In this format, the size of the dump file is close to the size obtained using gzip, but it allows you to restore tables selectively. The following command dumps the database in the `custom` format:

```
$ pg_dump -Fc <dbname> > <filename>
```

A dump file of the `custom` format is not a script, use pg_restore utility to restore it:

```
$ pg_restore -d <dbname> <filename>
```

For very large databases, you might need to combine using the split utility with the `custom` dump format or compressed dump files.

Use parallel dump feature

To speed up the dump of a large database, you can use the `pg_dump` parallel mode. It dumps multiple tables at the same time. You can control the degree of parallelism with the `-j` parameter. Parallel dumps are only supported for the `directory` archive format.

```
$ pg_dump -j <number> -F d -f <out_directory> <dbname>
```

You can use `pg_restore -j` to restore a dump in parallel. This works for any archive in the `custom` or `directory` format, even if the archive is not created with `pg_dump -j`.

File system level backup

Eugenia Kuzina

Contents

To Table of Contents

Use pg_basebackup

Create an archive file

Make a snapshot

Use rsync

An alternative backup strategy is to directly copy the files that PostgreSQL uses to store data in the database. By default, ADPG places the database files at the following path: */pg_data1/adpg16*.

You can use any method that you prefer to back up the file system. Possible options are described below.

Use pg_basebackup

You can use the pg_basebackup tool to take a base backup of a running PostgreSQL database cluster. The `pg_basebackup` tool creates backups without affecting other database clients.

The `pg_basebackup` utility creates an exact copy of the cluster files, automatically enabling and disabling backup mode. Backups are created for the entire cluster. You cannot make a copy of individual databases or database objects. For selective backups, use `pg_dump` as described in the SQL dump article.

The `pg_basebackup` tool makes backups over a regular PostgreSQL connection that uses the replication protocol. The connection must be made on behalf of a superuser or a user that has a permission to initiate a replication, and *pg_hba.conf* must allow the replication connection.

The following example creates backup of the server at the `10.92.6.36` host and stores it in the local directory `/usr/local/pgsql/data`:

```
$ pg_basebackup -h 10.92.6.36 -D /usr/local/pgsql/data
```

The code below backs up the local server with one compressed tar file for each tablespace and stores it in the directory `backup2022-12-22`:

```
$ pg_basebackup -D backup2022-12-22 -Ft
```

For more information on the `pg_basebackup` options, see pg_basebackup.

Create an archive file

It is also possible to create an archive file:

```
$ tar -cf backup20221215.tar /pg_data1/adpg16
```

This method has two restrictions:

- The database server must be shut down. Use the Stop action for this. You also need to shut down the server before restoring the data.
- It is impossible to back up or restore only certain individual tables or databases from their respective files or directories. This does not work because the information contained in these files is not usable without the commit log files, *pg_xact/**, which contain the commit status of all transactions. A table file is only usable with this information. File system backups must be used for complete backup and restoration of an entire database cluster.

Make a snapshot

An alternative file-system backup approach is to make a consistent snapshot of the data directory, if the file system supports that functionality. The typical procedure is to make a "frozen snapshot" of the volume containing the database, then copy the whole data directory from the snapshot to a backup device, then release the "frozen snapshot". This works even while the database server is running. However, a backup created in this way saves the database files in a state as if the database server is not properly shut down. When you start the database server on the backed-up data, it operates as if the previous server instance crashes and replays the WAL log. Be sure to include the WAL files in your backup. You can execute the CHECKPOINT command before you take the snapshot to reduce recovery time.

If your database is spread across multiple file systems, there might not be any way to obtain exactly-simultaneous "frozen snapshots" of all the volumes. For example, if your data files and WAL log are on different disks, or if tablespaces are on different file systems, it might not be possible to use snapshot backup because the snapshots must be simultaneous. You can shut down the database server and make all the "frozen snapshots".

Use rsync

Another option is to use the rsync utility to perform a file system backup. Run `rsync` while the database server is running, then shut down the database server to execute `rsync --checksum` (`--checksum` is necessary because `rsync` only has the file modification-time granularity of one second). The second `rsync` is faster than the first because it needs to transfer less data. The result is consistent because the server is down. This method allows you to back up your file system with minimal downtime.

NOTE

Note that a file system backup is usually larger than an SQL dump. For example, `pg_dump` does not need to dump the contents of indexes, only the commands to recreate them. However, backing up the file system might be faster.

Continuous archiving and point-in-time recovery (PITR)

Eugenia Kuzina

Contents

To Table of Contents

Overview

Set up WAL archiving

Make a base backup

Use the low level API to make a base backup

Recover from a continuous archive backup

Overview

PostgreSQL writes every change made to the database data files to the write-ahead log (WAL). WAL files are located in the *pg_wal* directory. By default, ADPG places this directory at the following path: */pg_data1/adpg16/pg_wal*, where *pg_data1* is the `Data directory` parameter value. See Configuration parameters for details. If the system crashes, PostgreSQL "replays" WAL entries made since the last checkpoint.

It is possible to combine a file system level backup with WAL files for backup creation. The file system backup should be restored first, and after that WAL entries should be applied to a database to bring the system to a current state. This approach has the following advantages:

- It does not require a perfectly consistent file system backup as the starting point. When PostgreSQL applies WAL entries, it corrects any inconsistency in the backup. You do not need a file system snapshot capability and can use tar or a similar archiving tool.
- Since we can combine an indefinitely long sequence of WAL files for replay, continuous backup can be obtained by simply continuing to archive the WAL files. This is useful for large databases that require a significant amount of resources to create full backups frequently.
- It is not necessary to repeat operations from all WAL entries to the end. You can stop at any point and have a consistent point-in-time snapshot of the database. Therefore, this technique supports point-in-time recovery: it is possible to restore the database to its state at any point in time since the base backup was taken.
- If the server is continuously transferring WAL files to another machine that is loaded with the same base backup file, this machine has a current copy of the database and can replace the server in case of failure.

NOTE

The `pg_dump` and `pg_dumpall` tools do not produce file system level backups and cannot be used as part of a continuous-archiving solution.

Note that PITR supports restoring an entire database cluster, not a part of it. It also requires additional storage for the base backup and archived WAL files.

To successfully use continuous archiving, you need a continuous sequence of archived WAL files that starts no later than the backup start. You should set up and test the procedure for archiving WAL files before you take the first base backup.

Set up WAL archiving

PostgreSQL produces an indefinitely long sequence of WAL records. The system divides this sequence into WAL segment files, which are typically 16 MB in size. The segment files have numeric names that reflect their position in the abstract WAL sequence. If the server does not use WAL archiving, the system creates a few segment files and then "recycles" them by renaming segment files to higher segment numbers. It is assumed that segment files whose contents precede the last checkpoint can be recycled.

To archive WAL data, you need to capture the contents of each segment file once it is filled and save it somewhere before the segment file is recycled. For example, you can copy the segment files to an NFS-mounted directory on another machine or write them to any external drive. PostgreSQL lets the administrator specify a shell command for making a copy of a completed segment file. The command can be simple or invokes a complex shell script.

To enable WAL archiving, set the wal_level configuration parameter to `replica` or higher, archive_mode to `on`, and specify the shell command for the archive_command configuration parameter. Use the ***ADPG configurations*** section on the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** tab in ADCM UI to specify these parameters. See Configuration parameters for details.

In `archive_command`, use `%p` instead of a path name of the file to archive, and `%f` instead of a file name. The path name is relative to the cluster data directory. For example, the following `archive_command` copies WAL segments to the */mnt/server/archivedir* directory:

```
archive_command : 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'
```

During execution the server replaces the `%p` and `%f` parameters with values and the command looks as follows:

```
test ! -f /mnt/server/archivedir/000000060000000000000040 && cp pg_wal/000000060000000000000040
/mnt/server/archivedir/000000060000000000000040
```

A similar command is generated for each new file to be archived. The archive command is executed on behalf of the same user that runs ADPG.

The archive command should not overwrite any existing archive file. This is an important security feature to preserve the archive integrity in case of administrator errors (for example, when sending the output of two different servers to the same archive directory). In the example above, the archive command includes the `test` command for this purpose.

An example of setting the `archive_command` parameter with a shell script:

```
archive_command : 'local_backup_script.sh "%p" "%f"'
```

Since archived WAL files contain information from your database, make sure that the directory with these files has appropriate access permissions.

The archive command should only return a zero exit status if it succeeds. When PostgreSQL gets a zero result, it indicates that the file has been successfully archived and can be recycled. If the command returns a non-zero status, PostgreSQL periodically tries to archive the file until it succeeds.

When a signal (other than `SIGTERM`, that is used as part of a server shutdown) or a shell error terminates an archive command execution, the archive process is aborted and restarted. In such cases, the failure is not reported in pg_stat_archiver.

PostgreSQL invokes the archive command for completed WAL segments. If the server generates little WAL traffic, it can cause a long delay between the completion of a transaction and its recording in the archive storage. You can set the archive_timeout parameter to limit this delay. This parameter forces the server to switch to a new WAL segment file after the specified period of time.

You can also force the server to switch to a new WAL file with the `pg_switch_wal` function if you want to archive a completed transaction as soon as possible. Other utility functions related to WAL management are listed in Backup Control Functions.

Make a base backup

You can use the pg_basebackup tool as described in File system level backup to create a base backup.

To make use of the backup, you need to keep all the WAL segment files generated during and after the file system backup. The base backup process creates a backup history file that is stored in the WAL archive directory. This file is named after the first WAL segment file that you need for the file system backup. For example, if the starting WAL file is *000000010000123400000055CD* the backup history file is named as *000000010000123400000055CD.007C9330.backup*. The second part of the filename indicates the exact position in the WAL file and can be ignored.

It is necessary to keep all archived WAL files since the last base backup. That is why the base backup interval is usually chosen based on the size of storage for the WAL file archive. You should also consider the time required for recovering data. The system should replay all WAL segments that are archived since the last base backup to be restored.

Use the low level API to make a base backup

Low level base backups can be made in a non-exclusive or an exclusive way. The non-exclusive method is recommended and the exclusive is deprecated. Non-exclusive backup allows you to run other concurrent backup processes that can use the same backup API or the `pg_basebackup` tool.

The base backup creation with the low level API contains more steps than the method using `pg_basebackup`. It is very important that these steps are performed in order, and the next step is only executed if the previous one is successful.

1. Ensure that WAL archiving is enabled and working.
2. Connect to the server as a user with rights to run the pg_start_backup function and execute the command:

```
SELECT pg_start_backup('label', false, false);
```

   The `pg_start_backup` function has the following syntax:

```
pg_start_backup(<unique_identifier>, <quick_backup>, <exclusive>)
```

   Where:

   - `<unique_identifier>` is a string that is a unique identifier for the backup operation.
   - `<quick_backup>` By default, `pg_start_backup` can take a long time to complete. The `pg_start_backup` function performs a checkpoint, and I/O operations required for the checkpoint are spread out over a significant period of time that equals to half the inter-checkpoint interval. See the checkpoint_completion_target configuration parameter. It minimizes the impact on query processing. If you want to start the backup as soon as possible, change the second parameter to `true`. It starts an immediate checkpoint using as much I/O as available.
   - `<exclusive>` specifies if `pg_start_backup` initiates an exclusive base backup. In our example, a non-exclusive base backup is performed.
     The connection that calls `pg_start_backup` must be maintained until the backup is completed, otherwise, the process is automatically aborted.
3. Use any file system backup tool to create a backup. See Backing up the data directory for more information on what directories must be copied.
4. In the same connection as in step 2, execute pg_stop_backup:

```
SELECT * FROM pg_stop_backup(false, true);
```

   The `pg_stop_backup` function has the following syntax:

```
pg_stop_backup(<exclusive>, <wait_for_archive>)
```

   Where:

   - `<exclusive>` specifies if the backup is created in the exclusive mode. This parameter must match the value from the `pg_start_backup` call.
   - `<wait_for_archive>` specifies whether `pg_stop_backup` should wait until the WAL files are archived.
     This command terminates the backup mode. On a primary, it also performs an automatic switch to the next WAL segment. On a standby, it is not possible to automatically switch WAL segments, you can run `pg_switch_wal` on the primary to perform a manual switch. This operation makes the last WAL segment file written during the backup ready to archive.

     The `pg_stop_backup` function returns one row with three values. Write the second value to the *backup_label* file in the backup root directory. The third value should be written to the *tablespace_map* file unless the value field is empty. These files are critical and must be written byte by byte unchanged, which may require a file to be opened in the binary mode.
5. The WAL segment files used during the backup should be archived. The file identified by the `pg_stop_backup` first returned value is the last segment that is required to form a complete set of backup files. On a primary, if `archive_mode` is enabled and the `wait_for_archive` parameter of the `pg_stop_backup` function is `true`, `pg_stop_backup` does not return values until the last segment is archived. On a standby, `archive_mode` must be `always` to force `pg_stop_backup` to wait for archiving. Archiving WAL files occurs automatically if you configure `archive_command`.

Recover from a continuous archive backup

To recover from the backup, perform the following steps:

1. Stop the server.
2. Copy the cluster data directory and any tablespaces to a temporary location. This operation requires that you have enough free space on your system to hold two copies of your database. If you do not have enough space to save all the data, you must save the contents of the *pg_wal* directory, because it might contain logs that are not archived before the system failure.
3. Remove all existing files and subdirectories under the cluster data directory and root directories of any tablespaces.
4. Restore the database files from your file system backup. Make sure that they are restored with the correct ownership (the database system user, not `root`) and with the correct permissions. If you are using tablespaces, verify that the symbolic links in *pg_tblspc/* are restored properly.
5. Remove any files present in *pg_wal/*. These files are from the file system backup and are probably obsolete. If you did not use *pg_wal/*, create this directory with proper permissions. If you use a symbolic link, recreate it.
6. If you have unarchived WAL segment files that you saved in step 2, copy them into *pg_wal/*. Do not move them, so you still have the unmodified files if a problem occurs, and you have to start all over again.
7. Set recovery configuration settings in ***ADPG configurations*** and create the *recovery.signal* file in the cluster data directory (the default data directory in ADPG is */pg_data1/adpg16*). You can also temporarily modify the ***PG_HBA*** section in ADCM UI to prevent user connections until you recover the system.
8. Start the server. The server goes into recovery mode and reads the required archived WAL files. If the recovery is terminated because of an external error, the server can be restarted and the recovery will continue. When the restore process is complete, the server deletes *recovery.signal* to prevent an accidental re-entry into recovery mode and begins standard database operations.
9. Inspect the database contents to ensure that the desired state is restored. If so, edit the ***PG_HBA*** section to allow users to connect to the restored database. If not, return to step 1.

In step 7, you need to set up the recovery configuration that describes how the restore is performed and to what point. Use the restore_command parameter that specifies how to retrieve archived WAL file segments. This parameter is similar to `archive_command`. It contains a shell command string with `%p` instead of a path name to copy the log file to and `%f` instead of a log file name. The path name is relative to the cluster data directory.

Example:

```
restore_command : 'cp /mnt/server/archivedir/%f %p'
```

The `restore_command` parameter can also contain a shell script.

If the server cannot find WAL segments in the archive, it searches for them in the *pg_wal/* directory. It allows using recent non-archived segments. Segments from the archive have a higher priority than files from *pg_wal/*.

The recovery process handles all available WAL segments and restores the database to the current point in time or as close as possible given the available WAL segments. Therefore, a recovery ends with a "file not found" message, the exact text of the error message depends on the current `restore_command`. You may also see an error message at the start of recovery for a file named similar to *00000001 history*. This also does not indicate a problem in most cases. See Timelines for details.

If you need to recover to some previous point in time, set a stopping point. You can specify a stopping point ("recovery target") by date/time, a named restore point, or by completion of a specific transaction ID. At present, only the date/time and named restore point options are usable since there are no tools to help you identify which transaction ID to use.

If the recovery process finds corrupted WAL data, the recovery halts at that point and the server does not start. In this case, run the recovery process from the beginning and specify a "recovery target" before the point of corruption. The recovery can complete successfully. If the recovery fails for an external reason, such as a system crash or if the WAL archive becomes inaccessible, the recovery can be restarted, and it restarts from the point of the failure. The recovery restart works in the same way as checkpoints: the server periodically saves the current state to the disk and updates the *pg_control* file. This way, already processed WAL data is not rescanned.

Backup and restore using pgBackRest

By Evgeniy Rybkin

Contents

To Table of Contents

Overview

Backup configuration

- Backup storages
- The backup retention
- The backup compression
- Logging
- Custom pgBackRest configuration

Create a backup

Restore an ADPG cluster

Get a list of backups

Manual PITR restore of ADPG cluster

Overview

IMPORTANT

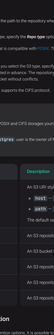
- This functionality is deprecated and will be removed after a major ADPG upgrade supporting PostgreSQL 17. Instead, use the part of ADPG Control — backup manager.
- Note that migration of previously created backups to the storage of the ADPG Control backup manager is not supported.

ADPG uses the functionality of the `pgbackRest` utility to create backups of the cluster and restore it from the backups. This solution easily scales for large databases and significant workloads. All backup actions are executed on behalf of the `postgres` user.

- **Backup** is a consistent copy of a database cluster that can be used to recover the cluster from a failure, to perform Point-In-Time Recovery (PITR), or to bring up a new standby. The `pgbackRest` utility can create the following types of backups:
- **Full backup** — `pgbackRest` copies the entire contents of a cluster to the backup. The first backup of the database cluster is always a full backup. `pgbackRest` can restore a cluster directly from a full backup. The full backups do not depend on any files outside it.
- **Differential backup** — `pgbackRest` only copies database files that have changed since the last full backup. To restore a differential backup, `pgbackRest` copies all files from the current differential backup and appropriate unchanged files from the previous full backup. A differential backup takes up less disk space but requires a valid full backup to be restored.
- **Incremental backup** — `pgbackRest` only copies database files that have changed since the last backup, which can be another incremental backup, a differential backup, or a full backup. Due to the fact that incremental backups only include files changed since the prior backup, they are generally much smaller than full or differential backups. The incremental backup size, depends on other backups. All prior incremental, differential, and full backups must be valid to perform a restore of the incremental backup. The backup sequence may not include differential backups.

Backup configuration

To enable creating backups, switch on the **Enable backups** toggle button displayed on the **Primary configuration** tab of the **ADPG** service (**Clusters** → **ADPG cluster** → **Services** → **ADPG** → **Primary configuration**). When this toggle button is enabled, you can expand the corresponding node and specify the `pgbackRest` settings described below.



Enable backups

NOTE

After you set all required settings on the **Primary configuration** tab, click **Save** to save the current configuration and run the Reconfigure & Restart action of the ADPG service.

In `pgbackRest` settings, each PostgreSQL/ADPG cluster to back up has its configuration called **stanza**. A stanza defines where a cluster is located, how it should be backed up, archiving options, and others. In ADPG, a stanza is created with the name specified in the **Backup stanza** field and the added prefix `adppreviewint`. For example, if the specified name is `mystanza`, the result — `adppg14-mystanza`.

You can also set the **Archive timeout(s)** parameter that specifies the amount of time (in seconds) to wait before forcing a switch to the next WAL file. The default value is `3600`. It is helpful when the database is lightly loaded and the WALs are slowly filling up, to ensure that WAL archiving does not stay for a long period.

Backup storages

Use **Backup repo** to specify the path to the repository where `pgbackRest` stores backups and archives WAL segments. The repository must exist and be accessible to all ADPG nodes in the cluster.

To determine the storage type, specify the **Repo type** option. You can use the following types of storage to save backups:

- `posix` — a storage that is compatible with POSIX. This is the default value. Directories on all nodes must be synchronized with each other, for example, using network file system tools.
- `s3` — an S3 storage. If you select the S3 type, specify settings located in the **S3 configuration** section. The S3 settings are listed in the table below. The bucket used to store the repository must be created in advance. The repository can be located in the bucket root (`/`) but it is usually best to place it in a subfolder. So, object store logs or other data can also be stored in the bucket without conflicts.
- `cifs` — a storage that supports the CIFS protocol.

NOTE

You need to configure POSIX and CIFS storages yourself for the cluster to work correctly. ADPG does not check the POSIX and CIFS configuration and mounted directories.

Make sure that the `postgres` user is the owner of POSIX folders or has read/write access to them on each host. If not, recreate the POSIX folders under the `postgres` user.

Parameter	Description
S3 URI style	An S3 URI style. The following URI styles are supported: - `host` - <bucket>.<endpoint>.<host>. For example, mybucket.us-east-1.amazonaws.com - `path` - <endpoint>.<host>/<bucket>. For example, us-east-1.amazonaws.com/mybucket The default value is `path`.
S3 Region	An S3 repository region where the bucket was created
S3 Bucket	An S3 bucket used to store the repository with backups
S3 Endpoint	An S3 repository endpoint. The endpoint should be valid for the specified region
S3 Key	An S3 repository access key used to access the bucket
S3 Key Secret	An S3 repository secret access key used to access the bucket

The backup retention

You can specify backup retention options. It is possible to choose whether a time (days) or a number will be used to determine which backups are expired. **Retention full type** allows specifying whether the **Retention full** parameter represents a time period (days) or a number of full backups to keep. The following **Retention full type** values are available:

- `count` — the number of backups to keep. The default value.
- `time` — full backups older than the **Retention full** value (in days) will be removed from the repository if there is at least one backup that expires in a number of days equal to or greater than **Retention full**. If **Retention full** is 30 (days) and there are two full backups over 29 days old and one 25 days old, no full backups will expire because once the 30-day backup expires, only the 25-day backup will remain, which would violate the retention policy of having at least one backup 30 days old before an older one can be expired. Where a full backup expires, all differential and incremental backups associated with the full backup will also expire.

You can use the **Retention diff** option to specify the number of differential backups to retain. When a differential backup expires, all incremental backups associated with the differential backup will also expire. If this option is not specified, then all differentials will be retained until the full backups on which they depend have expired.

The backup compression

To enable the backup compression, check the **Enable compression** checkbox and specify the **Compression type** option. Use the one of the following values: `bz2`, `gz`, `lz4`, and `zst`. The default value is `gz`.

You can also set **Compress level** — a file compression level (from `0` to `9`) used when **Compression type** does not equal `none`. If this option is not specified, the following levels are used based on the **Compression type** value:

- `bz2` — `9`;
- `gz` — `6`;
- `lz4` — `1`;
- `zst` — `3`.

Logging

By default, ADPG places `pgbackRest` log files at the following path: /var/log/adpg14-pgbackrest. Use the **Log path** option to change the log file directory.

You can also set a `pgbackRest` log level in the **Log level** field. Possible values are: `off`, `error`, `warn`, `info`, `detail`, `debug`, and `trace`. The default value is `info`.

Custom pgBackRest configuration

You can set `pgbackRest` settings that are not available in UI. For this, check the **Use custom config** checkbox and specify the required settings in the **Global options** and **Custom options** fields. In this case, all the `pgbackRest` settings except for **Backup stanza** parameter are applied from these fields. Settings from **Global options** are added to the `[global]` section of the `/pgbackrest` configuration file. The configuration file is located at the following path: etc/adpg14-pgbackrest/adpg14-pgbackrest.conf.

NOTE

If you check **Use custom config** and specify custom `pgbackRest` settings, ADPG writes these settings to the `pgbackRest` configuration file without checking. In case of incorrect settings, an error message will not be displayed in the ADCM UI. You need to verify the settings and check if the directory for storing backups exists and is accessible by yourself.

Create a backup

After you enable backups and set all the required settings as described above, use the Backup Start action of the ADPG service to create a backup. When you run this action, ADCM displays a dialog window where you can select a backup type from a drop-down list.

Change the backup type

In the opened window, select a backup type.

Select a backup type

By default, `pgbackRest` creates incremental backups. However, an incremental backup must be based on a full backup. If there are no full backups, `pgbackRest` creates a full backup, regardless of which backup type is selected.

After the backup type is selected, click **Run** to confirm the action. A backup will be created in the specified repository (**Backup repo**). If errors occur during backup creation, you can find information about them on the **Jobs** page.

Restore an ADPG cluster

Use the Backup: Restore cluster action of the ADPG service to restore a cluster from the last backup on all nodes. You can find information about the last backup in the output of the Backup Info action. If errors occur during the operation, information about them will be displayed on the **Jobs** page.

IMPORTANT

The Backup: Restore cluster action restores the cluster to the end of the WAL archive. In case of an accidental deletion of database objects, use manual PITR restore with the `--target` parameter, since it is most likely that the **Backup: Restore Cluster** action will restore the state of the cluster in the moment when the database objects were already deleted/dropped.

Get a list of backups

Run the Backup: Info action of the ADPG service to get a list of created backups. You can find the action result on the **Jobs** page. To view the backup list, click the **Backup: Info** row in the task list, switch to the **Ansible [check]** tab of the job, find the **Show backups information** row at the end of the list, and click it to expand.

The result of the 'Backup: Info' action

If errors occur during the operation, information about them will also be displayed on the **Jobs** page.

Manual PITR restore of ADPG cluster

In the automatic mode, you can restore cluster only from the last backup. To restore a cluster to a specific point in time, use the steps below:

1. Make sure that an appropriate backup exists. You can use the Backup: Info action for this.
2. Set the `${TIME}` variable to a time you want to restore the cluster.

```
$ TIME="2024-04-23 09:00:00+00"
```

3. Use the Stop action to stop the ADPG service. Make sure that all ADPG instances are stopped. You can check this on the **Jobs** page.
4. You can optionally delete the contents of the /pg_data/adpg14 directory on each ADPG node and run recovery without the `--delta` option. Alternatively, you can start the recovery with the `--delta` option, as shown below. In this case, it is not necessary to delete the contents, but you can copy them to another directory, which can be useful if the recovery fails.
5. Remove the cluster `/initialized` key on any etcd node using the following command:

```
$ /usr/lib/adpg14/usr/bin/etcdctl del --prefix=true "/areneatera/adpg14/cluster_name>/<cluster_uuid>/initialized"
```

When `<cluster_uuid>` is a cluster UUID (see **General configuration parameters** → **Cluster UUID** in Configure a cluster), and `<cluster_name>` is a cluster name in a snake case.

For example:

```
$ /usr/lib/adpg14/usr/bin/etcdctl del --prefix=true "/areneatera/adpg14/test_adpg_cluster_1/91cdeace-aeeb-4daa-bb1a-d181f7f9ecaa/initialized"
```

6. Use `pgbackRest` to restore the cluster on each ADPG node to the time specified by the `${TIME}` variable:

```
$ /usr/lib/adpg14/usr/bin/pgbackrest --config=/etc/adpg14-pgbackrest/adpg14-pgbackrest.conf --stanza=<stanza-name> --delta --type=time "--target=${TIME}" --target-action=promote restore
```

When `<stanza-name>` is the **Backup stanza** field value with the `adppreviewint-` prefix.

For example:

```
$ /usr/lib/adpg14/usr/bin/pgbackrest --config=/etc/adpg14-pgbackrest/adpg14-pgbackrest.conf --stanza=adpg14-mystanza --delta --type=time "--target=${TIME}" --target-action=promote restore
```

7. Make sure that the postgresql.auto.conf and recovery.signal files exist in the /pg_data/adpg14 directory.
8. Start PostgreSQL on the leader node on behalf of the `postgres` user using the following command:

```
$ /usr/lib/adpg14/bin/pg_ctl start -D /pg_data/adpg14
```

9. Search for the `starting point-in-time recovery` string entry in the ADPG log files (the default path is /pg_data/adpg14/log) to make sure that point-in-time-recovery has been started. See more log files for details.

```
2024-04-23 14:14:11.478 UTC [29516] LOG: starting point-in-time recovery to 2024-04-23 09:00:00+00
...
```

10. Stop PostgreSQL on the leader node on behalf of the `postgres` user using the following command:

```
$ /usr/lib/adpg14/bin/pg_ctl stop -D /pg_data/adpg14
```

11. Use the Start action to start the ADPG service.

Vacuuming

A reprint from

Contents

Overview
VACUUM and VACUUM FULL commands
Recover disk space
Update planner statistics
Update the visibility map
Prevent transaction ID wraparound failures
Multixacts and wraparound
Autovacuum

Overview

ADP's PostgreSQL maintains data consistency by using a multi-version model (Multiversion Concurrency Control, MVCC). Unless **both read/write** locking - data is not changed and it is not read while it is being changed - is used, each SQL statement sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This prevents statements from viewing inconsistent data produced by concurrent transactions performing updates on the same data rows, thus maintaining data integrity for each transaction (ACID). However, this model has some drawbacks:

- An [UPDATE](#) or [DELETE](#) of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multi-version concurrency - the old version of the row can be seen by transactions that started before the update or deletion command - but the old version of the row will eventually need to be removed from the table or its space will be consumed by dead rows that will grow larger and larger.
- A [SELECT](#) command can only see rows created by transactions that committed before the command started. Rows created by transactions that are still in progress or aborted are not visible to other commands. Information about which transactions have committed or aborted is stored in [pg_xact](#), and consulted frequently.
- [INSERT](#) and [UPDATE](#) commands usually add new rows to the end of the table. In a busy environment, many processes may be inserting rows at almost the same time, sometimes causing contention for the end of the table.

These issues are not unique to PostgreSQL, but they do manifest differently than in some other systems, so this chapter will explain how ADP's PostgreSQL handles them.

VACUUM and VACUUM FULL commands

The [VACUUM](#) command is the housekeeping tool for PostgreSQL's multi-version model. It can be run manually whenever you want, but you don't have to, because PostgreSQL has an "autovacuum" facility that will run VACUUM operations when appropriate. The autovacuum facility automates the execution of the [VACUUM](#) command.

Create a table and populate it with some data to see how VACUUM performs its tasks:

```
CREATE TABLE test2 (
    id integer,
    val text
);
INSERT INTO test2 VALUES (generate_series(1,1024), 'x');
```

Delete a row from the table and update another row:

```
DELETE FROM test2 WHERE id = 1;
UPDATE test2 SET val = 'y' WHERE id = 2;
```

Two rows are displayed, and the second row has a new number of the inserting transaction:

```
SELECT xmin, xmax, ctid, id, val FROM test2 WHERE id <= 2;
```

```
 xmin | xmax | ctid  | id | val
------+------+-------+----+-----
  733 |    0 | (0,2) |  2 | y
  732 |  733 | (0,1) |  1 | x
(2 rows)
```

The following table fields are shown:

- [xmin](#) — the identity (transaction ID) of the inserting transaction for this row version.
- [xmax](#) — the identity (transaction ID) of the deleting transaction, or zero for an undeleted row version.
- [ctid](#) — the physical location of the row version within its table.
- [xmax](#) — the command's number within the deleting transaction, or zero.
- [ctid](#) — the physical location of the new row version if the row has been updated. For the most current version of a row, [ctid](#) refers to the version itself. If the following [pageinspect](#) function [tuple_data_split](#) is used, the [ctid](#) field can be decoded (in the default 8 kB block size) to show the block number (in the first hexadecimal byte) and index (in the second hexadecimal byte) within the block.

To view dead rows, create the [pageinspect](#) extension:

```
CREATE EXTENSION pageinspect;
```

Run the following query that uses the [heap_page_items](#) function to display all tuples:

```
SELECT lp as tuple, t_xmin, t_xmax, t_field3 FROM heap_page_items(get_raw_page('test2',0));
```

Result:

```
 tuple | t_xmin | t_xmax | t_field3
-------+--------+--------+----------
     1 |    733 |      0 |        0
     2 |    733 |    733 |        0
     3 |    732 |    733 |        0
     4 |    733 |      0 |        0
(4 rows)
```

The [flags](#) field contains two dead tuples ([2 and 3](#)).

Then, run the [VACUUM](#) command on the table and view the [VERBOSE](#) key word to print a detailed vacuum activity report:

```
VACUUM VERBOSE test2;
```

Result:

```
INFO:  vacuuming "public.test2"
INFO:  "test2": removed 2 row versions in 1 pages
INFO:  "test2": found 2 removable, 3 nonremovable row versions in 5 out of 7 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 734
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Run the query again:

```
SELECT lp as tuple, t_xmin, t_xmax, t_field3 FROM heap_page_items(get_raw_page('test2',0));
```

Result:

```
 tuple | t_xmin | t_xmax | t_field3
-------+--------+--------+----------
     1 |        |        |
     2 |    733 |      0 |        0
     3 |        |        |
     4 |    733 |      0 |        0
(4 rows)
```

From the result, you can see that [VACUUM](#) removed the dead tuples, but does not free up the space. However, [VACUUM](#) makes it available for re-use, and new tuples will be written in place of the deleted dead tuples.

You can also use the query function [pg_relation_size](#) to obtain table statistics:

```
SELECT relname, n_live_tup, n_dead_tup
  FROM pg_stat_user_tables WHERE relname = 'test2';
```

Result:

```
 relname | n_live_tup | n_dead_tup
---------+------------+------------
 test2   |          3 |          0
(1 row)
```

If tables and columns are specified in the [VACUUM](#) command, it processes every table and materialized view in the current database. Also, [VACUUM](#) can be executed on a single table or on specific columns of a table.

The [VACUUM](#) command can be executed with one of options in the [FULL](#) key word:

```
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table_and_columns [, ...] ]
```

Execute the query mentioned above to view all tuples in the [test2](#) table:

```
SELECT lp as tuple, t_xmin, t_xmax, t_field3 FROM heap_page_items(get_raw_page('test2',0));
```

Result:

```
 tuple | t_xmin | t_xmax | t_field3
-------+--------+--------+----------
     1 |        |        |
     2 |    733 |      0 |        0
     3 |        |        |
     4 |    733 |      0 |        0
(4 rows)
```

Run the [VACUUM](#) command on the table with the [FULL](#) option and view the query again:

```
VACUUM FULL VERBOSE test2;
SELECT lp as tuple, t_xmin, t_xmax, t_field3 FROM heap_page_items(get_raw_page('test2',0));
```

Result:

```
INFO:  vacuuming "public.test2"
INFO:  "test2": found 0 removable, 2 nonremovable row versions in 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 734
There were 0 unused item identifiers.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
 tuple | t_xmin | t_xmax | t_field3
-------+--------+--------+----------
     1 |        |        |
     2 |    733 |      0 |        0
     3 |        |        |
     4 |    733 |      0 |        0
(4 rows)
```

The [VACUUM FULL](#) creates a new copy of the table and writes the data to it. The [VACUUM FULL](#) requires an [ACCESS EXCLUSIVE](#) lock on the table, and cannot be done in parallel with other operations on the table. Therefore, it should generally be done during a maintenance window. The [VACUUM FULL](#) command can also be executed on a single table or on specific columns of a table. Typically, [VACUUM FULL](#) should only be used when a significant amount of space needs to be reclaimed from within the table.

The [FREEZE](#) option causes the command to aggressively freeze tuples. Normally, [VACUUM](#) will freeze only tuples that are old enough to be considered stable. The knowledge of the appropriate cutoff point is known to the server, but sometimes it's useful to force freezing of all tuples.

The [VERBOSE](#) option causes the command to print a detailed vacuum activity report for each table.

The [table_and_columns](#) option specifies the table and columns to be vacuumed. If omitted, [VACUUM](#) processes every table in the current database. The [table_and_columns](#) option has the following subfields:

- [table_name](#) — the name (possibly schema-qualified) of a table to be vacuumed. If omitted, the command processes all tables in the current database that the current user has permission to vacuum.
- [column_name](#) — the name of a specific column to be analyzed. Defaults to all columns. If a column list is specified, [ANALYZE](#) must also be specified.

Recover disk space

As mentioned above, an [UPDATE](#) or [DELETE](#) command does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multi-version concurrency control (MVCC, see the [User Guide](#)): the old version of the row can be seen by transactions that started before the update or deletion command, but the old version of the row will eventually need to be removed from the table or its space will be consumed by dead rows that will grow larger and larger. A [SELECT](#) command can only see rows created by transactions that committed before the command started. Rows created by transactions that are still in progress or aborted are not visible to other commands. Information about which transactions have committed or aborted is stored in [pg_xact](#), and consulted frequently. An [INSERT](#) and [UPDATE](#) commands usually add new rows to the end of the table. In a busy environment, many processes may be inserting rows at almost the same time, sometimes causing contention for the end of the table.

These issues are not unique to PostgreSQL, but they do manifest differently than in some other systems, so this chapter will explain how ADP's PostgreSQL handles them.

The [VACUUM](#) command processes every table in the current database (or the specified table or tables), and discards dead row versions. To do this, it requires an [ACCESS EXCLUSIVE](#) lock on each table, and cannot be done in parallel with other operations on the table. Therefore, it should generally be done during a maintenance window.

Sometimes it's better to run [VACUUM](#) separately on individual tables, and it is possible to do routine vacuuming on tables without any parameters, so that PostgreSQL can determine which tables need attention. However, this strategy does not recover as much space as [VACUUM FULL](#) does.

Plain [VACUUM](#) may not be satisfactory when a table contains large numbers of dead row versions as a result of massive update or delete operations. If you have an [UPDATE](#) or [DELETE](#) command that will modify a large portion of a table, then you may want to run [VACUUM](#) afterwards. This command will mark the dead row versions in the table for deletion, and it will allow additional tables to be processed by any autovacuum daemon that is running. Add the [ANALYZE](#) option if you want to update statistics used by the planner to determine the most efficient way to execute a query.

Another option is the [VACUUM FULL](#) command. It does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed. It also requires extra disk space, since it writes a new copy of the table and doesn't release the old copy until the operation is complete. Usually, you should only need to use [VACUUM FULL](#) when you want to make a table physically smaller. Running [VACUUM](#) or [VACUUM FULL](#) rewrites the table into a new disk file with only the minimum amount of space needed, and returns the old space to the operating system.

Update planner statistics

The query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the [ANALYZE](#) command, which can be invoked by itself or as an option of [VACUUM](#). It is important to have reasonably accurate statistics, otherwise poor choices of plans might degrade database performance.

The autovacuum daemon, if enabled, will automatically issue [ANALYZE](#) commands whenever the content of a table has changed sufficiently. However, administrators might prefer to rely on manually scheduled [ANALYZE](#) operations, particularly if it is known that update activity on a table will not affect the statistics of "interesting" columns. The daemon schedules [ANALYZE](#) strategically rather than blindly, so it has no useful knowledge of when such an update will be significant.

Typically changed in performance and characteristics on its trigger [ANALYZE](#) on the system state. If the system state is simply or rarely changed, it may never be processed by autovacuum, and the statistics will be inaccurate. Before running queries on a freshly loaded table, or after making substantial changes in its contents, you should run [ANALYZE](#).

Update the visibility map

Vacuuming maintains an [visibility map](#) for each table to keep track of which pages contain only tuples that are known to be visible to all active transactions (and all future transactions, until the page is again modified). This has two purposes. First, vacuum itself can skip such pages on the next run, since there is nothing to clean up. Second, it allows PostgreSQL to answer some queries using only the index, without reference to the underlying table. Since indexes do not contain tuple visibility information, normally each index entry requires a fetch of the corresponding table tuple, a random access that can be quite expensive. With the visibility map, such queries can avoid fetching tuples that were recently modified. Vacuum keeps the visibility map small by not setting bits for pages that have had a lot of recent changes. Instead, it waits for a period in which the whole page is known to be unchanged. Visibility map bits are only set by vacuum, but are cleared by any data-modifying operation on the table.

Prevent transaction ID wraparound failures

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an XID greater than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss. (Actually, the data is still there, but that's cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason [VACUUM](#) exists is because of this wraparound problem. Before PostgreSQL 8.1, [VACUUM](#) was only rarely used, and so wraparound failures would likely start to occur after two billion transactions had elapsed since the last manual vacuum. Starting in 8.1, autovacuum is enabled by default, and it will launch [VACUUM](#) operations as needed to prevent wraparound failures. These automatic vacuums are not so disruptive as they once were, because cost-based vacuum delays are enabled by default.

To help the user avoid the ordinary hazards of wraparound failures, [VACUUM](#) issues hints to administrators when the system is running low on usable XIDs. Specifically, it will emit a warning when the oldest XID in a database has reached [autovacuum_freeze_max_age](#) transactions, and will refuse to assign additional XIDs when the oldest XID has reached 1 million transactions from the end of the wraparound limit. Both of these actions can be performed only when [VACUUM](#) is able to examine every table in the affected database. A convenient way to examine this information is to execute queries in the [pg_database](#) system catalog, for example:

```
SELECT datname, age(datfrozenxid) FROM pg_database;
```

Result:

```
  datname  |  age
-----------+-------
 template1 | 16911
 template0 | 16911
 postgres  | 16911
 mydb      | 16911
 myuser    | 16911
(5 rows)
```

The [datfrozenxid](#) column is the XID at which an [VACUUM](#) operation last completed scanning all tables in the database. The [age](#) column is the number of transactions before that XID. Provided that [VACUUM](#) is running properly, this column should never exceed [autovacuum_freeze_max_age](#) by more than the number of transactions that can occur in a single vacuum cycle.

If [VACUUM](#) is failing to keep up with transaction ID consumption, you will see warnings like this when it is within 40 million transactions of failure:

```
WARNING:  database "myfdb" must be vacuumed within 39985967 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in "myfdb".
```

Issuing a [VACUUM](#) command for the database will clear this warning.

Normally [VACUUM](#) should be the only operation that is necessary to avoid XID wraparound failures. However, if you have a [PREPARED](#) transaction that is very old, or a replication slow, [VACUUM](#) will not be able to advance the database's [datfrozenxid](#) past the XID of that transaction. You will see warnings about old prepared transactions in the log in this situation. The solution is to commit or abort the prepared transaction.

Multixacts and wraparound

Multixact IDs are used to support row locking by multiple transactions. Since there is only limited space in a tuple header to store lock information, that information is encoded as a "multiple transaction ID", or multixact ID for short, whenever there is more than one transaction concurrently locking a row. Information about which multixacts are used in which pages is stored in the [pg_multixact](#) subdirectory, and a multixact ID is stored in the [xmax](#) field or sometimes in the [xmax](#) field of a tuple header.

Like transaction IDs, multixact IDs are implemented as a 32-bit counter and corresponding storage, all of which requires careful aging management, storage cleanup, and wraparound handling.

There is a separate struct called [MultiXactMember](#) which stores the transaction ID and the member's lock mode for each member of the multixact. The lock modes used for different purposes are mentioned in the [explicit-locking](#) section of the [User Guide](#). Keeping this data efficiently is thus significantly more complex than storing transaction IDs, since each multixact may have anywhere from zero to many members. A multixact may also change its member lock mode as individual transactions end. For example, a multixact with two members, one locking and one updating, will be left with a multixact with one member if the locking transaction commits or rolls back.

Multixact IDs are assigned by the [MultiXactIdCreate](#) routine. Callers must provide a list of transaction IDs and lock modes for the initial members of the multixact. [MultiXactIdCreate](#) searches the existing multixacts to see if there is one that already has exactly the desired members. If not, it creates a new multixact, assigning it the next available multixact ID and adding the appropriate data to the [pg_multixact](#) files. (Adding data to these files in turn requires allocating appropriate amounts of shared memory.)

Autovacuum

ADP's PostgreSQL has an optional but highly recommended feature called [autovacuum](#) that automates the execution of the [VACUUM](#) and [ANALYZE](#) commands. When enabled, autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples. These checks use the statistics collection facility; therefore, autovacuum cannot be used unless [track_counts](#) is set to [true](#). In the default configuration, autovacuuming is enabled and the related configuration parameters are appropriately set.

The "autovacuum daemon" actually consists of multiple processes. There is a persistent daemon process, called the autovacuum launcher, which is in charge of starting [autovacuum worker](#) processes for all databases. The launcher will distribute the work across time, attempting to start one worker within each database every [autovacuum_naptime](#) seconds. (Therefore, if the installation has [N](#) databases, a new worker will be launched every [autovacuum_naptime/N](#) seconds.) A maximum of [autovacuum_max_workers](#) processes may be running at once. If several databases need to be processed simultaneously, the launcher will launch as many workers as it can, with as much as possible of a two-second delay between each. Note that the number of running autovacuum worker processes, if any, will not be counted against the [max_connections](#) or [superuser_reserved_connections](#) limits.

Tables whose [relfrozenxid](#) value is more than [autovacuum_freeze_max_age](#) transactions old are always vacuumed (this also applies to those tables whose freezing was disabled by setting [autovacuum_enabled](#) for their [pg_class](#) entry). Otherwise, if the number of tuples obsoleted since the last [VACUUM](#) exceeds the "vacuum threshold", the table is vacuumed. The vacuum threshold is defined as:

```
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

where the vacuum base threshold is [autovacuum_vacuum_threshold](#),
the vacuum scale factor is [autovacuum_vacuum_scale_factor](#),
and the number of tuples is [reltuples](#).

The table is also vacuumed if the number of tuples inserted since the last vacuum has exceeded the [insert threshold](#), which is defined as:

```
insert threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

where the vacuum base threshold is [autovacuum_vacuum_insert_threshold](#),
the vacuum scale factor is [autovacuum_vacuum_scale_factor](#),
and the number of tuples is [reltuples](#).

Temporary tables cannot be accessed by autovacuum. Therefore, temporary table operations should be performed by manual [VACUUM](#) or [ANALYZE](#) commands.

For analyze operations, similar [ANALYZE](#) operations are performed on tables where the [autovacuum_analyze_threshold](#) (parameter) and [autovacuum_analyze_scale_factor](#) (parameter) are used. Like vacuum, analyze is not performed on such tables.

```
analyze threshold = analyze base threshold + analyze scale factor * number of tuples
```

where the analyze base threshold is [autovacuum_analyze_threshold](#) (parameter),
the analyze scale factor is [autovacuum_analyze_scale_factor](#) (parameter),
and the number of tuples is [reltuples](#).

Partitioned tables do not directly store tuples and consequently are not processed by autovacuum. (Autovacuum does process table partitions in the usual fashion.) Manual [VACUUM](#) and [ANALYZE](#) operations on partitioned tables are also unsupported. To perform maintenance operations on a partitioned table, maintenance operations should be performed directly on the table partitions.

The autovacuum daemon cancels any autovacuum worker process whose operation is blocking other [ACCESS EXCLUSIVE](#) lock requests (such as an attempt to drop a table), after [deadlock_timeout](#) milliseconds have passed. This avoids locking problems that could result in a severe degradation of the system's performance.

Tuples changed in temporary tables are eligible for autovacuuming using the two settings mentioned above. For [ANALYZE](#) operations, similar [ANALYZE](#) operations are performed on such tables.

Obtain statistics via ANALYZE

Eugeny Kuzma

Contents

To Table of Contents

The ANALYZE command
- Overview
- Examples
- Configuration parameters affecting ANALYZE
- ANALYZE on partitioned and child tables

Statistics used by the planner
- Single-column statistics
- Extended statistics

Database statistics play a significant role in improving performance. The `ANALYZE` command collects statistics about the contents of tables in a database. The query planner uses these statistics to create efficient query plans.

The ANALYZE command

Overview

The `ANALYZE` command collects statistics about the contents of tables in the database and stores the results in the pg_statistic system catalog. Subsequently, the query planner uses these statistics to determine the most efficient execution plans for queries. `ANALYZE` has the following syntax:

```
ANALYZE [ ( <options> [, ...] ) ] [ <table_and_columns> [, ...] ]
ANALYZE [ VERBOSE ] [ <table_and_columns> [, ...] ]
```

where `<options>` can be one of:

```
    VERBOSE [ <boolean> ]
    SKIP_LOCKED [ <boolean> ]
```

and `<table_and_columns>` is:

```
    <table_name> [ ( <column_name> [, ...] ) ]
```

Parameters of the ANALYZE command

VERBOSE	Enables display of progress messages. When `VERBOSE` is specified, `ANALYZE` emits progress messages to indicate which table is currently being processed and prints various statistics
SKIP_LOCKED	Specifies that `ANALYZE` should not wait for any conflicting locks to be released when processing a relation. If a relation cannot be locked without waiting, the relation is skipped. Note that even with this option, `ANALYZE` can wait when opening relation indexes or when acquiring sample rows from partitions, child tables in the inheritance hierarchy, and some types of foreign tables. This option also forces `ANALYZE` to skip all partitions if there is a conflicting lock on a partitioned table
<boolean>	Specifies whether the selected option should be turned on or off. You can write `TRUE`, `ON`, or `1` to enable the option, and `FALSE`, `OFF`, or `0` to disable it. If the boolean value is omitted, `TRUE` is assumed
<table_name>	A name (possibly schema-qualified) of a table to analyze. If omitted, all regular tables, partitioned tables, and materialized views in the current database are analyzed but not foreign tables. If the specified table is the partitioned table, both the statistics of the entire partitioned table and the statistics of the individual partitions are updated
<column_name>	A name of a specific column to analyze. By default, all columns are analyzed

To analyze a table, a user must be the table's owner or a superuser. Database owners are allowed to analyze all tables in their databases, except shared catalogs. The `ANALYZE` skips over any tables that the user does not have permission to analyze.

Examples

ANALYZE VERBOSE on a database

To run `ANALYZE` on a database, first connect to the database. For example, you can do it using psql:

```
\c example
```

Run the `ANALYZE VERBOSE` command:

```
ANALYZE VERBOSE;
```

Result:

```
INFO:  analyzing "public.pgbench_accounts"
INFO:  "pgbench_accounts": scanned 30000 of 81968 pages, containing 1038000 live rows and 0 dead rows; 30000 rows in sample, 
1038000 estimated total rows
INFO:  analyzing "public.pgbench_branches"
INFO:  "pgbench_branches": scanned 1 of 1 pages, containing 50 live rows and 0 dead rows; 50 rows in sample, 50 estimated 
total rows
INFO:  analyzing "public.pgbench_history"
INFO:  "pgbench_tellers": scanned 1 of 1 pages, containing 100 live rows and 0 dead rows; 100 rows in sample, 100 estimated 
total rows
INFO:  analyzing "pg_catalog.pg_type"
INFO:  "pg_type": scanned 24 of 35 pages, containing 409 live rows and 0 dead rows; 409 rows in sample, 409 estimated total 
rows
INFO:  analyzing "pg_catalog.pg_foreign_table"
INFO:  "pg_foreign_table": scanned 0 of 0 pages, containing 0 live rows and 0 dead rows; 0 rows in sample, 0 estimated total 
rows
INFO:  analyzing "pg_catalog.pg_authid"
INFO:  "pg_authid": scanned 1 of 1 pages, containing 16 live rows and 0 dead rows; 16 rows in sample, 16 estimated total rows
INFO:  analyzing "pg_catalog.pg_statistic_ext_data"
INFO:  "pg_statistic_ext_data": scanned 0 of 0 pages, containing 0 live rows and 0 dead rows; 0 rows in sample, 0 estimated 
total rows
...
```

Since the `VERBOSE` option is used, the output contains detailed information on analyzed relations.

ANALYZE on a table

To analyze a table, specify its name in the command:

```
ANALYZE pgbench_accounts;
```

ANALYZE for specific columns

To analyze a column, specify a table name and a column name in parentheses:

```
ANALYZE pgbench_accounts (bid, abalance);
```

Foreign tables are analyzed only when explicitly selected. Not all foreign data wrappers support `ANALYZE`. If a table's wrapper does not support `ANALYZE`, the command prints a warning and does nothing.

The autovacuum daemon runs `ANALYZE` for tables when they are first loaded with data, and if they are changed during regular operations. If autovacuum is disabled (not recommended), you should run `ANALYZE` periodically and after major changes in the contents of tables. Accurate statistics help the planner to choose the most appropriate query plan, and improve the speed of query processing. A common strategy for databases that are mainly intended for reading is to run `VACUUM` and `ANALYZE` once a day during a low usage time of day. This is not sufficient if a database is being actively updated.

`ANALYZE` requires only a read lock on the target table, so it can run in parallel with other operations on the table.

If a table is empty, `ANALYZE` does not write new statistics for that table. Any existing statistics are retained.

Each backend running `ANALYZE` reports its progress to the `pg_stat_progress_analyze` view. For more information, see ANALYZE progress reporting.

Configuration parameters affecting ANALYZE

The statistics collected by `ANALYZE` usually include a list of the most common values and a histogram showing the approximate data distribution in each column. One or both of them can be omitted if they are not useful in the column data type does not support the appropriate operators. For example, there are no common values in a column with unique keys.

For large tables, `ANALYZE` takes a random sample of the table's contents, rather than examining each row. This allows even very large tables to be analyzed in a short period of time. Note, that the statistics are only approximate, and will change slightly each time `ANALYZE` is run, even if the actual contents of the table have not changed. It results in small changes in the planner's estimated costs shown by `EXPLAIN`. In rare situations, this non-determinism will cause the planner's choice of a query plan to change after `ANALYZE` is run. To avoid this, increase the amount of statistics collected by the `ANALYZE` command as described below.

The amount of statistics can be controlled by setting the default_statistics_target configuration parameter. Specify this parameter in the **ADPG configurations** section on the **Clusters** → **ADPG Cluster** → **Parameters** → **ADPG configuration** tab in ADCM UI (see Configuration parameters). You can also set the per-column statistics target with the following command: `ALTER TABLE … ALTER COLUMN … SET STATISTICS`. See ALTER TABLE for details.

```
ALTER TABLE table1 ALTER COLUMN column1 SET STATISTICS 0;
```

The target value sets the maximum number of entries in the most common value list and the maximum number of bins in the histogram. The default target value is `100`, but it can be lowered or increased to find a balance between the accuracy of the planner's estimates and the time required to run `ANALYZE`, as well as the amount of statistics in `pg_statistic`. Setting the statistics target to `0` disables collection of statistics for that column. It can be useful for columns that are never used as part of the `WHERE`, `GROUP BY`, or `ORDER BY` clauses of queries, since the planner does not use the statistics on such columns.

The largest statistics target among the columns determines the number of table rows selected for preparing statistics. Increasing the target causes a proportional increase in the time and space required by `ANALYZE`.

The second value estimated by `ANALYZE` is the number of distinct values that appear in each column. Because only a subset of the rows is examined, this estimate can be quite inaccurate, even with the largest possible statistics target. If this leads to bad query plans, a more accurate value can be determined manually and installed with the command: `ALTER TABLE … ALTER COLUMN … SET (n_distinct = <value>)`:

```
ALTER TABLE table2 ALTER COLUMN column2 SET (n_distinct = 500);
```

ANALYZE on partitioned and child tables

If a table has inheritance children, `ANALYZE` collects two sets of statistics: one on the rows of the parent table only, and the second including rows of the parent table and all of its children. The planner uses the latter set of statistics when processing queries that operate with the entire inheritance tree. In this case, the child tables are not individually analyzed. The autovacuum daemon only considers inserts or updates on the parent table itself when deciding whether to trigger an automatic `ANALYZE` for that table. If the table is rarely updated, the inherited statistics will be outdated unless you run `ANALYZE` manually.

For partitioned tables, `ANALYZE` gathers statistics by sampling rows from all partitions. In addition, it will recurse into each partition and update its statistics. Each partition is analyzed only once, even with multi-level partitioning. `ANALYZE` does not collect statistics for a partitioned table (without data from its partitions), because it is empty.

The autovacuum daemon does not process partitioned tables, and parent tables if only children or partitions were modified. It is usually necessary to periodically run `ANALYZE` manually to keep the statistics of the table hierarchy up to date.

If any child tables or partitions are foreign tables whose foreign data wrappers do not support `ANALYZE`, these tables are ignored while gathering statistics.

Statistics used by the planner

As mentioned above, the query planner needs to estimate the number of rows retrieved by a query to make efficient query plans. This section provides a quick look at the statistics that the system uses for these estimates.

Single-column statistics

One component of the statistics is the total number of entries and the number of disk blocks occupied by each table and index. The pg_class table contains this information in the `reltuples` and `relpages` columns.

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'pgbench_accounts%';
```

Result:

```
      relname      | relkind |  reltuples   | relpages
-------------------+---------+--------------+----------
 pgbench_accounts  | r       | 5.000046e+06 |    81968
 pgbench_accounts_pkey | i     | 5.000046e+06 |    13711
```

The `pgbench_accounts` table and its index contain `5.000046e+06` rows, but the index is smaller than the table.

For efficiency reasons, `reltuples` and `relpages` are updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`. The `VACUUM` + `ANALYZE` operation may not scan the entire table but incrementally updates `reltuples` on the basis of the part of the table it did scan, resulting in an approximate value. The planner scales the values from `pg_class` to match the current physical table size, and thus obtains a closer approximation.

Most queries retrieve only a fraction of the rows in a table limited by the `WHERE` clauses. The planner needs to estimate the selectivity of `WHERE` clauses, that is, the fraction of rows that match each condition in the `WHERE` clause. To do this, the planner uses information stored in the pg_statistic system catalog. The `ANALYZE` and `VACUUM ANALYZE` commands update entries in `pg_statistic`, but they are always approximate even immediately after an update.

Rather than look at `pg_statistic` directly, it is better to look at the pg_stats view when examining the statistics. The `pg_stats` view is readable by all, whereas `pg_statistic` is only available for a superuser. This prevents unprivileged users from obtaining information about the contents of other user's tables from the statistics. The `pg_stats` view is restricted to only rows about tables that the current user can read.

You can execute the following query to `pg_stats` where `tennisatla` is a table name:

```
SELECT attname, n_distinct,
array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'tennisatla';
```

Result:

```
    attname    | n_distinct |  most_common_vals
---------------+------------+---------------------
 terminal_code |         -1 |
 timezone      | -0.1654154 | Europe/Moscow     +
               |            | Asia/Yekaterinburg+
               |            | Asia/Krasnoyarsk  +
               |            | Asia/Irkutsk      +
               |            | Asia/Yakutsk      +
               |            | Europe/Samara     +
               |            | Asia/Vladivostok  +
               |            | Asia/Novokuznetsk +
               |            | Europe/Volgograd
 terminal_name |         -1 |
 city          |  -0.971153 | Moscow            +
               |            | Ulyanovsk
```

where:

- `attname` — name of column described by this row.
- `n_distinct` — if greater than zero, the value is the estimated number of distinct values in the column. If this number is less than zero, its modulus represents the number of distinct values divided by the number of rows. The negated form is used when `ANALYZE` expects that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values. `-1` indicates a unique column in which the number of distinct values is the same as the number of rows.
- `most_common_vals` — a list of the most common values in the column. `NULL` (empty in the result above) if no values seem to be more common than any others.

Extended statistics

It is common to see slow queries running bad execution plans because multiple columns used in the query clauses are correlated. The planner usually assumes that multiple conditions are independent of each other. Regular statistics cannot take into account cross-column correlation. However, ADPG/PostgreSQL can compute multivariate statistics that collect such information.

Since the number of possible column combinations is very large, it is impractical to compute multivariate statistics automatically. Instead, ADPG/PostgreSQL allows you to create **extended statistics objects** to obtain statistics on selected sets of columns.

Statistics objects are created using the `CREATE STATISTICS` command. The creation of such an object adds an entry expressing the need for these statistics. `ANALYZE` collects the requested statistical data, which can be viewed in the pg_statistic_ext_data catalog.

The simplest type of extended statistics is tracking functional dependencies. This concept is used in the definitions of database normal forms. The `b` column is functionally dependent on the `a` column if knowing the value of `a` is sufficient to determine the value of `b`, that is, there are no two rows with the same value of `a` but different values of `b`.

The existence of functional dependencies directly affects the accuracy of estimates in certain queries. To inform the planner about functional dependencies, `ANALYZE` can gather measurements of cross-column dependency. Data is collected only for those groups of columns that are referenced in the extended statistics object defined with the `dependencies` option. It is recommended to create dependencies statistics only for column groups that are strongly correlated to avoid unnecessary overhead in `ANALYZE` and later query planning.

The following code creates the `stat1` extended statistics object to collect functional-dependency statistics on the `terminal_name` and `city` columns of the `terminals` table:

```
CREATE STATISTICS stat1 (dependencies) ON terminal_name, city FROM terminals;

ANALYZE terminals;
```

Check the result:

```
SELECT stxname, stxkeys, stxddependencies
FROM pg_statistic_ext join pg_statistic_ext_data on (stxoid = stxoid)
WHERE stxname = 'stat1';
```

Result:

```
 stxname | stxkeys |          stxddependencies
---------+---------+------------------------------------
 stat1   | 2 3     | {"2 => 3": 1.000000, "3 => 2": 0.667513}
```

The column `2` (`terminal_name`) fully determines the column `3` (`city`) — the coefficient is `1.0`, while `city` only determines `terminal_name` about 67% of all cases, meaning that there are cities that have more than one terminal.

For more examples of multivariate statistics, refer to how the planner uses statistics.

Logging

Eugenia Kuzina

Contents

To Table of Contents

Configure log settings

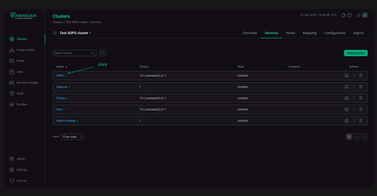
View log files

This topic describes how to configure ADPG log settings and view log files.

Configure log settings

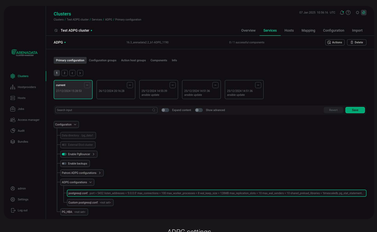
Use ADCM to define log settings.

Open the ***Clusters*** page and click the ADPG cluster. Open the ***Services*** tab and click the ***ADPG*** service.



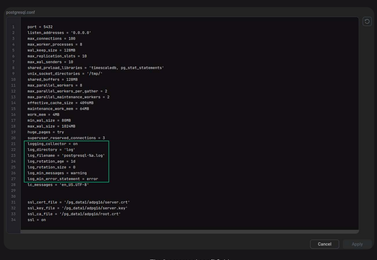
ADPG cluster services

The ***Primary configuration*** tab of the ADPG service is opened. Expand the ***ADPG configurations*** node and click ***postgresql.conf***.



ADPG settings

The text field with configuration parameters is displayed. In this field, you can specify the required logging parameters.



The "postgresql.conf" field

The most frequently used log settings are listed in the table.

ADPG log settings

Parameter name	Description	Default value
logging_collector	Enables the logging collector. The logging collector is a background process that captures log messages sent to stderr and redirects them into log files	The logging collector is enabled (`true`)
log_directory	Determines the directory that contains log files. It can be specified as an absolute path, or relative to the cluster data directory	log (the absolute path is /pg_data1/adpg16/log)
log_filename	Specifies the log file name pattern. The value can include strftime %-escapes to define time-varying file names. If you specify a file name pattern without escapes, use a log rotation utility to save disk space	postgresql-%a.log
log_rotation_age	Determines the maximum period of time to use a log file, after which a new log file is created. If this value is specified without units, it is taken as minutes. Set `log_rotation_age` to `0` to disable time-based log file creation	1d
log_rotation_size	Determines the maximum size of a log file. After a log file reaches the specified size, a new log file is created. If the value is set without units, it is taken as kilobytes. Set `log_rotation_size` to `0` to disable size-based log file creation	0
log_min_messages	Specifies the minimum severity level of messages that are written to a log file. Valid values are `debug5`, `debug4`, `debug3`, `debug2`, `debug1`, `info`, `notice`, `warning`, `error`, `log`, `fatal`, and `panic` (see the Severity levels table). Messages with the specified severity or higher are included in the log file. For example, if you set `log_min_messages` to `warning`, the log file will include the `warning`, `error`, `log`, `fatal`, and `panic` messages	warning
log_min_error_statement	Specifies which SQL statements that cause errors are logged. Valid values are `debug5`, `debug4`, `debug3`, `debug2`, `debug1`, `info`, `notice`, `warning`, `error`, `log`, `fatal`, and `panic` (see the Severity levels table). The log file includes SQL statements for messages of the specified severity or higher. To disable error statement logging, set `log_min_error_statement` to `panic`	error

Log message severity levels are listed below.

Severity levels

Value	Description
Debug1..Debug5	Detailed information for developers. `Debug1` — the least detailed information, `Debug5` — the most detailed information
Info	Information implicitly requested by a user, for example, output from `VACUUM VERBOSE`
Notice	Information that can be helpful to users, for example, a notification about truncation of long identifiers
Warning	A warning about a possible problem, for example, `COMMIT` outside a transaction block
Error	An error that aborts the current command
Log	Information for administrators, for example, checkpoint activity
Fatal	The current session is aborted due to an error
Panic	All database sessions are aborted

If you modify log settings, run the Reconfigure & Restart action of the ADPG service to apply changes.

The settings listed above are standard PostgreSQL server configuration parameters. For additional information about these settings, see the following topic: Error Reporting and Logging.

CAUTION

When the ADPG service executes the Reconfigure & Restart action, *postgresql.conf* is rewritten with settings specified in ***Services → ADPG → Configuration***. If you make changes directly to *postgresql.conf*, these changes will be lost.

View log files

You can find logs on the host where ADPG is installed.

1. Connect to the host (for example, via SSH), run the ls command, and pass a path to the log file directory as a parameter:

```
$ sudo ls -la /pg_data1/adpg16/log
```

   Log files in the specified directory are listed:

```
total 10708
drwx------  2 postgres postgres     136 May 26 00:00 .
drwx------ 20 postgres postgres    4096 May 26 08:54 ..
-rw-------  1 postgres postgres    1009 May 20 13:31 postgresql-Fri.log
-rw-------  1 postgres postgres     596 May 23 06:30 postgresql-Mon.log
-rw-------  1 postgres postgres 3803084 May 26 10:04 postgresql-Thu.log
-rw-------  1 postgres postgres     908 May 24 07:37 postgresql-Tue.log
-rw-------  1 postgres postgres 3343041 May 25 23:59 postgresql-Wed.log
```

2. You can use the tail command to view a log file content:

```
$ sudo tail /pg_data1/adpg16/log/postgresql-Mon.log
```

   Result:

```
2022-05-23 06:30:40.804 GMT [951] LOG:  starting PostgreSQL 14.2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44), 64-bit
2022-05-23 06:30:40.804 GMT [951] LOG:  listening on IPv6 address "::1", port 5432
2022-05-23 06:30:40.804 GMT [951] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2022-05-23 06:30:40.813 GMT [951] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
2022-05-23 06:30:40.845 GMT [1079] LOG:  database system was shut down at 2022-05-20 13:31:40 GMT
2022-05-23 06:30:40.972 GMT [951] LOG:  database system is ready to accept connections
```

Performance tuning

A maintenance...

Contents

To Table of Contents

- Optimize resource consumption
  - Memory settings
  - Settings related to I/O operations
- Determine disk usage
  - Use database object size functions
    - Inspect system catalogs
    - Use the old bloom module
- Populate a database
  - Disable autocommit
  - Use COPY
  - Remove indexes
  - Remove foreign key constraints
  - Increase maintenance_work_mem
  - Increase max_wal_size
  - Disable WAL archiving and backups
  - Run ANALYZE afterwards
  - Improve pg_dump performance

Query performance can be affected by many factors. A part of them can be controlled by a user or administrator, while others are mostly influenced by the underlying system architecture. This article provides some tips for understanding and tuning PostgreSQL performance.

Optimize resource consumption

This section contains general guidelines on configuration parameters related with resource consumption. Detailed information about parameters is available in the documentation. Each of them has its own resource consumption that depends on PostgreSQL settings based on the specific hardware configuration. Some of them are:

Memory settings

The table below contains basic recommendations on memory settings.

Parameter name	Description	Recommended value
effective_cache_size	Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is taken into account when estimating the cost of using an index. A higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both PostgreSQL's shared buffers and the portion of the kernel's disk cache that will be used for PostgreSQL data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by PostgreSQL, and it does not reserve kernel disk cache; it is used only for estimations. The system also does not assume data remains in the disk cache between queries. If the value is set without units, it is taken as blocks. The default is BLCKSZ (8192 bytes).	2/3 RAM
shared_buffers	Sets the amount of memory the database server uses for shared memory buffers. If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for shared_buffers is 1/4 of the memory in your system. There are some workloads where even large settings for shared_buffers are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to shared_buffers will work better than a smaller amount. Larger settings for shared_buffers usually require a corresponding increase in max_wal_size, in order to spread out the process of writing large amounts of new or changed data over appropriate periods of time. If the value is set without units, it is taken as blocks. The default is BLCKSZ (8192 bytes), typically 8 kB.	8Gb/4
temp_buffers	Sets the maximum amount of memory used for temporary buffers within each session. These are session-local buffers used only for access to temporary tables. If the value is set without units, it is taken as blocks. The default is BLCKSZ bytes, typically 8 kB.	256 MB
work_mem	Sets the base maximum amount of memory to be used by a query operation (such as a sort or hash table) before writing to temporary disk files. If this value is specified without units, it is taken as kilobytes. The default is 4 MB. Note that a complex query might perform several sort and hash operations at the same time, with each operation generally being allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of work_mem; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for ORDER BY, DISTINCT, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of IN subqueries.	8/64/32
maintenance_work_mem	Specifies the maximum amount of memory to be used by maintenance operations, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. It defaults to 64 MB. Since only one of these operations can be executed at a time by a session, and an installation usually does not execute such operations on a large number of concurrent sessions, it is safe to set this value significantly larger than work_mem. Larger settings may improve performance for vacuuming and for restoring database dumps. Note that when autovacuum runs, up to autovacuum_max_workers times this memory may be allocated, so be careful not to set the value too high. It may be useful to control the autovacuum_work_mem configuration parameter to control autovacuum.	8/64/16

Once you set up the specified parameters, you can check their values by using the `SHOW` command. For example, `SHOW work_mem;` will display the value for work_mem.

Settings related to I/O operations

The table below contains basic recommendations on I/O operations, changing the values of which can significantly affect performance.

Parameter name	Description
fsync	If this parameter is on, the PostgreSQL server tries to make sure that updates are physically written to disk, by issuing fsync() system calls or various equivalent methods (see wal_sync_method). This ensures that the database cluster can be restored to a consistent state after an operating system or hardware crash. However, using fsync results in a performance penalty: when a transaction is committed, PostgreSQL must wait for the operating system to flush the write-ahead log to disk. When fsync is off, PostgreSQL does not attempt to force the operating system to write out these buffers. Turning off fsync can result in unrecoverable data corruption in case of a power failure or a system crash. Therefore, it is recommended to turn off fsync only when you are willing to start your database server from scratch and re-load all of its data. Turning off fsync can also prevent data corruption in the event of certain types of software crashes or misconfiguration of the operating system or hardware that you may experience while fsync remains on. Finally, many users have seen good performance gains by turning off fsync, because it reduces the number of times that disk writes have to be forced to the disk. Because the default is on, you will need to actively change this setting in postgresql.conf if you want fsync to be off.
synchronous_commit	Specifies how much WAL processing must complete before the database server returns a success indication to the client. Valid values are on, remote_apply, local, and off. The default value is on. When it is off, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. Unlike fsync, setting this parameter to off does not create any risk of database inconsistency: an operating system or database server crash might result in some recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly. So, turning synchronous_commit off can be a useful alternative when performance is more important than exactly knowing whether a recent transaction has been saved to disk. For more information, refer to the Asynchronous Commit.
checkpoint_completion_target	The fraction of the checkpoint interval, which is used for writing checkpoint data (a fraction of effect when it overwrites). It is possible to control some transaction synchronization and others approximately. For example, to make a single multistatement transaction commit asynchronously when the default is the opposite, issue `SET LOCAL synchronous_commit TO OFF` within the transaction.
effective_io_concurrency	This value can be overridden for tables in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).
random_page_cost	Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is `4.0`. Reducing this value relative to `SEQ_PAGE_COST` will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs.

Determine disk usage

In ADPG/PostgreSQL, each table has a primary heap disk file where most of the data is stored. If a table has columns with potentially large values, there can also be a TOAST file associated with the table. Also, there is an associated index for each table. You can find the size of a table using object size functions, or by querying system catalogs. The following sections describe both methods.

The examples utilize the sample database.

You can monitor disk space in two ways:

- Use database object size functions.
- Inspect system catalogs.
- Use the old bloom module.

Use database object size functions

These SQL functions are the easiest to use and are generally recommended. They are listed in the following table.

For example, the `db_relation_size()` function returns the size of the specified table (`sqlmonk_accounts`) in the example's bytes. You can use the `\dt+` tool to find the following SQL functions are the easiest to use and are generally recommended. They are listed in the following table.

```
SELECT pg_relation_size('sqlmonk_accounts');
```

Result:

```
 pg_relation_size
------------------
         47144448
```

```
SELECT pg_size_pretty(pg_relation_size('sqlmonk_accounts'));
```

Result:

```
 pg_size_pretty
----------------
 45 MB
```

The `pg_size_pretty()` function converts a size in bytes into a more human-readable format with size units: bytes, kB, MB, GB or TB.

```
SELECT pg_size_pretty(pg_relation_size('sqlmonk_accounts'));
```

Result:

```
 pg_size_pretty
----------------
 45 MB
```

```
SELECT pg_size_pretty(pg_database_size('vendors'));
```

Result:

```
 pg_size_pretty
----------------
 7924 MB
```

You can use the `pg_database_size()` function to compute the total disk space used by the specified database (`example`) in human-readable format.

```
SELECT pg_size_pretty (pg_database_size('example'));
```

Result:

```
 pg_size_pretty
----------------
 7924 MB
```

Inspect system catalogs

Every table has a primary heap disk file in the database cluster to which it belongs. This file stores most of the data for the table. The following query determines the number of pages in the file (`sqlmonk_accounts`) for the table.

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'sqlmonk_accounts';
```

Result:

```
 pg_relation_filepath | relpages
----------------------+----------
 base/19321/25241     |     5748
```

Each page is typically 8 kilobytes. This is the same result that the `pg_relation_size()` function returns: `47144448 = 8192 * 5748`. Note that `relpages` is only updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`. The file path (`pg_relation_filepath`) assumes that the current user is a superuser.

To determine the values saved by the TOAST tables associated with the `my_table`, use the following query:

```
SELECT relname, relpages
FROM pg_class,
(SELECT reltoastrelid
FROM pg_class
WHERE relname = 'my_table') b
WHERE oid = b.reltoastrelid OR
oid = (SELECT indexrelid
FROM pg_index
WHERE indrelid = b.reltoastrelid)
ORDER BY relname;
```

Result:

```
        relname        | relpages
-----------------------+----------
 pg_toast_295925       |        0
 pg_toast_295925_index |        1
```

The query turns the bloated-row table for `sqlmonk_locations` table.

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'sqlmonk_locations' AND
  (i.indrelid = c.oid OR
  i.indrelid = 0) AND
  (i.indexrelid = c2.oid OR
  c2.oid = c.oid);
```

Result:

```
        relname        | relpages
-----------------------+----------
 sqlmonk_locations     |        2
 sqlmonk_locations_pkey |        1
```

To find the largest tables and indexes, use the following query:

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC LIMIT 5;
```

Result:

```
        relname        | relpages
-----------------------+----------
 sqlmonk_accounts      |    47144
 sqlmonk_accounts_pkey |    13713
 pg_proc               |     1349
 pg_depend             |      931
 pg_attribute          |      856
```

Use the oid2name module

The `oid2name` utility program helps administrators examine the file structure used by ADPG/PostgreSQL.

To make `oid2name` runs in a larger database and tablespaces (`SET`), specify the database (`dbname`) switch to name or else files belonging to relation1, and/or table name information.

```
$ oid2name
```

Result:

```
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  19321       example  pg_default
  19320       postgres  pg_default
  19319      template1  pg_default
  19318      template0  pg_default
```

For example, consider the `example` database from list above. To do this, change the current catalog to the database catalog.

```
$ cd /my_data/adpg/base/19321
```

where `19321` is the oid of the `example` database from the command-output above.

Display the first 5 database objects in the default tablespace, ordered by size (the biggest):

```
$ ls -lS | head -5
```

Result:

```
total 577724
-rw------- 1 postgres postgres 47144448 Jul 31 20:54 25241
-rw------- 1 postgres postgres 47144448 Jul 31 20:54 25241.1
-rw------- 1 postgres postgres 47144448 Jul 31 20:54 25241.2
-rw------- 1 postgres postgres 532480 Aug 1 08:15 2608
-rw------- 1 postgres postgres 516096 Jul 31 18:15 2658
```

Determine what the `25241*` is:

```
$ oid2name -d example -f 25241
```

Result:

```
From database "example":
  Filenode  Table Name
----------------------
     25241  sqlmonk_accounts
```

The `oid2name` utility requires a running database server with non-corrupt system catalogs. In situations with catastrophic database corruption, be sure to avoid it.

Populate a database

This section provides some suggestions on how to efficiently insert a large amount of data within populating a database for the first time or during the process of database use.

Disable autocommit

When using multi-row `INSERT` statements, turn off autocommit and do one commit at the end. In plain SQL terms this avoids invoking `COMMIT` at the start and `COMMIT` at the end. Some client libraries may do this behind the scenes, if you advise each manner to be committed separately. PostgreSQL does a lot of work for each transaction committed, so having all statements in one transaction is a fast improvement. At the same rate, the maximum of all rows will be inserted. So you don't run the risk with partially loaded data.

Use COPY

Use `COPY` to load all the rows in one command, instead of using a series of `INSERT` commands. The `COPY` command is optimized for loading large numbers of rows. It is less flexible than `INSERT`, but has significantly reduced overhead for large data loads. Since `COPY` is a single command, there is no need to disable autocommit if you use this method to populate a table.

If you cannot use `COPY`, it may be useful to utilize `PREPARE` to create a prepared `INSERT` statement, and then use `EXECUTE` as many times as required. This allows you to avoid the overhead of repeatedly parsing and planning the `INSERT`.

Note that loading a large number of rows using `COPY` is almost always faster than using `INSERT`, even if `PREPARE` is used and multiple insertions are batched into a single transaction.

`COPY` can be used with `CREATE TABLE` in the same transaction. In this case, no WAL needs to be written, because in case of an error, the file containing the newly loaded data will be removed anyway. However, this consideration only applies when `wal_level` is `minimal` as all commands must write WAL in order to archive or replication.

Remove indexes

If you are loading a fresh table or a large amount of data into an existing table, it is faster to remove the indexes, load the table, and then recreate the indexes. It is always quicker to update an index incrementally than to create it from scratch. Also, the resulting index will be more compact than an incrementally updated one, because empty pages are not left in the index.

Of course, the database performance for other users might suffer during the time the index is missing. Additionally, you should think twice before dropping a unique index, as without them the command may take significant time to recheck the constraint.

Remove foreign key constraints

Just as with indexes, a foreign key constraint can be checked "in bulk" more efficiently than row-by-row. It can be useful to drop foreign key constraints, load data, and re-create the constraints. Again, there is a trade-off between data load speed and loss of error checking while the constraint is missing.

Moreover, when you load data into a table with existing foreign key constraints, each new row requires an entry in the server's list of pending trigger events (since it is the firing of a trigger that checks the row's foreign key constraint). A huge number of pending events can take up lots of memory. It may be necessary and beneficial to break up your load operation into smaller transactions.

Increase maintenance_work_mem

Temporarily increasing the `maintenance_work_mem` configuration parameter when loading large amounts of data can lead to improved performance. This will help to speed up `CREATE INDEX` commands and `ALTER TABLE ADD FOREIGN KEY` commands. This does not affect the speed of the `COPY` command, so if this tip is only useful when you are using a set of techniques. Larger settings for `maintenance_work_mem` may be useful when restoring a database dump that contains lots of tables with indexes or foreign keys. However, be aware that memory is also used by each concurrent maintenance operation. To avoid exhausting memory, it is recommended to set `maintenance_work_mem` to no more than 1/8 of `max_connections`.

Increase max_wal_size

Temporarily increasing the `max_wal_size` configuration parameter can also make large data loads faster. Loading a large amount of data requires checkpoints to occur more often than the normal checkpoint frequency (specified by the `checkpoint_timeout` configuration setting). Whenever a checkpoint occurs, all dirty pages must be flushed to disk. By increasing `max_wal_size` temporarily during bulk data loads, the number of checkpoints that are required can be decreased. Increasing `max_wal_size` can also make the `pg_dump` command slow. Therefore, if you plan to use WAL archiving or streaming, you may want to disable it while doing a dump. To do that, set `archive_mode` to `off` and restart the server. After the dump is complete, change it back and restart again. Alternatively, if you want to keep archiving on, you can use `pg_dumpall -g` to dump only the global data, which is also needed to make the dump file usable.

Disable WAL archiving and backups

When loading large amounts of data into a cluster that uses WAL archiving and streaming replication, it might be faster to take a new base backup after the load is complete than to process a large amount of incremental WAL data.

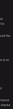
To disable WAL archiving, set the following configuration parameters:

- `wal_level` to `minimal`
- `archive_mode` to `off`
- `max_wal_senders` to `0`

Edit `ADPG.conf` configuration on the Clusters → `ADPG cluster` → Services → `ADPG` → Primary configuration tab in ADPG-UI to specify these parameters. See Configuration parameters for more information.

To disable `archive_command`, switch off the Archive mode toggle button displayed on the Primary configuration tab of the `ADPG` service (Clusters → `ADPG cluster` → Services → `ADPG` → Primary configuration).

To apply changes of these settings, access the Host Agent & Action section.



IMPORTANT

When you disable WAL archiving, it makes any base backups taken before unusable for point-in-time recovery. This is because crash recovery cannot replay the WAL data that was written during the base backup.

These settings can be used to discard WAL data for crash recovery only. If `wal_level` is set to `minimal`.

Run ANALYZE afterwards

Whenever you have significantly altered the distribution of data within a table, running the `ANALYZE` command is strongly recommended. Running `ANALYZE` or `VACUUM ANALYZE` ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner might make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics. Note that the `autovacuum` daemon, if enabled, will issue `ANALYZE` commands as

Improve pg_dump performance

`pg_dump` does not by itself alter the contents of the database. It takes a dump as quickly as possible, but the speed of other utilities depending on it is not necessarily true. It is useful to have data from a database backup as quickly as possible. Here are several guidelines and observations. All you have to consider is the following:

- Set appropriate (larger than normal) values for `maintenance_work_mem` and `max_wal_size`.
- If using WAL archiving or streaming, consider disabling them for the duration of the dump. See Disable WAL archiving and backups section.
- Experiment with the parallel dump and restore modes of both `pg_dump` and `pg_restore` and find the optimal number of concurrent jobs. Parallel data dumping and restoring, controlled by the `-j` option, should give you significantly higher performance over the serial mode.
- Consider using the `pg_dumpall -g` command to restore the dump in a single transaction. This will make your load faster by disabling `fsync` and `synchronous_commit`. Alternatively, you can choose to enable `fsync` and/or `synchronous_commit` and `wal_sync_method` before applying the dump. Any of the methods that you choose will make the recovery process faster, but also more vulnerable to problems with the underlying files and hardware. For example, there is an increased risk of data loss if the server crashes.
- If multiple CPUs are available in the database server, consider using the `pg_dump` option of `-j/--jobs`. This allows concurrent data loading and index creation.
- Run `ANALYZE` afterwards, as mentioned above.

A data dump does not contain the `VACUUM` or `ANALYZE` commands. Because of this, it may be useful to run `VACUUM` after restoring a data from a dump. If a `VACUUM` or `VACUUM ANALYZE` command is necessary, usually it is not necessary to specify any options for best results. A good combination of `VACUUM` options is `VACUUM (ANALYZE, FREEZE)`, which is short and fast. Running an `ANALYZE` command is necessary because `pg_dump` does not dump statistics data, even if `--clean` is specified. Running `VACUUM (FREEZE)` is necessary because `pg_dump` does not dump the contents of `relfrozenxid`. Running `VACUUM (FREEZE)` is necessary because `pg_dump` does not dump the contents of `relfrozenxid`. Some workloads can function well even if `relfrozenxid` is old and others cannot. `VACUUM` will also freeze `relfrozenxid` sometimes even if it is not very old. `VACUUM FREEZE` is necessary when `relfrozenxid` is close to the age where `VACUUM` would have begun scanning all unfrozen rows. `autovacuum_freeze_max_age` `vacuum_freeze_table_age` `vacuum_multixact_freeze_table_age` `vacuum_multixact_freeze_min_age` `VACUUM` at the end of the recovery.

Enable huge pages in Linux

Eugenia Kuzina

Contents

To Table of Contents

Enable huge pages

Calculate the required number of huge pages on a running cluster

Enable huge pages

Using huge pages reduces overhead when ADPG/PostgreSQL utilizes large contiguous chunks of memory. In particular, this occurs when the shared_buffers parameter is set to a large value. In ADPG, the default value of the `huge_pages` configuration parameter is `try`, but typically huge pages are not enabled in the operating system. To effectively use huge pages, enable them on operating system side. To do this, perform the steps below:

1. Check if a Linux kernel has the settings required for huge pages: `CONFIG_HUGETLBFS=y` and `CONFIG_HUGETLB_PAGE=y`. Utilize the following command:

```
$ grep HUGETLB /boot/config-$(uname -r)
```

   The output should be similar to:

```
CONFIG_ARCH_WANT_GENERAL_HUGETLB=y
CONFIG_CGROUP_HUGETLB=y
CONFIG_HUGETLBFS=y
CONFIG_HUGETLB_PAGE=y
```

2. Check that currently huge pages are not in use. Obtain the information from the virtual memory subsystem:

```
$ cat /proc/sys/vm/nr_hugepages
```

   Output:

```
0
```

   Also, you can get information related to huge pages from /proc/meminfo – the file that reports statistics about memory usage on the system:

```
$ grep ^HugePages /proc/meminfo
```

   Output:

```
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
```

3. Make sure that the `huge_pages` ADPG parameter value is set to `try`, for example, using the `psql` tool:

```
SHOW huge_pages;
```

   Result:

```
 huge_pages
------------
 try
```

4. Find the number of huge pages to reserve.
   Stop the cluster:

```
$ sudo systemctl stop adpg16-patroni.service
```

   Get a huge page size used in the system:

```
$ grep ^Hugepagesize /proc/meminfo
```

```
Hugepagesize:       2048 kB
```

   The runtime-computed PostgreSQL parameter `shared_memory_size_in_huge_pages` reports the number of huge pages required. This parameter value can be displayed with the `postgres` command. `/pg_data1/adpg16` is the path to *postgresql.conf*, it can be replaced by the `$PGDATA` environment variable if it is set:

```
$ postgres -D /pg_data1/adpg16 -C shared_memory_size_in_huge_pages
```

   Output:

```
73
```

   In this example, at least 73 huge pages are required, a larger setting would be appropriate if other programs also need huge pages.

   The command below displays a number of subdirectories of the form `hugepages-<size>kB`, where `<size>` is the size of the huge pages supported by the kernel/CPU combination.

```
$ ls /sys/kernel/mm/hugepages
```

   Output:

```
hugepages-1048576kB  hugepages-2048kB
```

   The default huge page size is 2 MB, but you can also explicitly request either 2 MB or 1 GB with the huge_page_size configuration parameter. To set this parameter, you can use the ***ADPG configurations*** section on the ***Primary configuration*** page of the ADPG service (see Configuration parameters). The number of pages calculated by `shared_memory_size_in_huge_pages`, will be adapted to the page size. To determine the optimal size of huge pages, test different page sizes with your database.

5. Set the `vm.nr_hugepages` kernel parameter that specifies the number of huge pages to reserve. You need to execute the following command on behalf of the `root` user:

```
$ sudo su -

$ echo 'vm.nr_hugepages = 73' >> /etc/sysctl.d/30-postgresql.conf
```

   Load kernel parameters from all files:

```
$ sysctl -p --system
```

   Check the result using the commands below:

```
$ grep ^HugePages /proc/meminfo
```

   Output:

```
HugePages_Total:      73
HugePages_Free:       73
HugePages_Rsvd:        0
HugePages_Surp:        0
```

```
$ sudo sysctl -a | grep 'hugepage'
```

   The output should contain the specified `vm.nr_hugepages` value:

```
...
vm.nr_hugepages = 73
...
```

6. Start the cluster:

```
$ sudo systemctl start adpg16-patroni.service
```

   See how the statistics changed:

```
$ grep ^HugePages /proc/meminfo
```

   Output:

```
HugePages_Total:      73
HugePages_Free:       63
HugePages_Rsvd:       63
HugePages_Surp:        0
```

Calculate the required number of huge pages on a running cluster

The optimal size and number of huge pages are determined as a result of testing, and if the first calculation method described above does not suit your needs, you can try to calculate the number of huge pages on a running cluster. The results obtained also need to be tested.

1. Determine pid of the postmaster process:

```
$ sudo head -1 /pg_data1/adpg16/postmaster.pid
```

   Output:

```
895
```

2. Get the peak memory usage of the postmaster process using pid:

```
$ grep ^VmPeak /proc/895/status
```

   Output:

```
VmPeak:   403440 kB
```

3. Calculate the number of huge pages as `VmPeak` divided by the huge page size plus `1` – `403440/2048+1`. In this example, the required number of huge pages is 197.

Since this method does not require stopping the cluster, you need to restart it instead of starting it after specifying the `vm.nr_hugepages` parameter (step 6 described above). Press `y` and `Enter` when needed to confirm the operation:

```
$ /usr/lib/adpg16/usr/bin/patronictl -c /etc/adpg16-patroni/adpg16-patroni.yml \
   restart a88bfefc-755f-4ccb-8614-989f0ddda0e7 test_adpg_cluster_eek1-adpg-adpg3
```

Where:

- `a88bfefc-755f-4ccb-8614-989f0ddda0e7` is a cluster UUID. You can find it in ADCM UI on the cluster ***Configuration*** tab.
- `test_adpg_cluster_eek1-adpg-adpg3` is a Patroni member name of the current node. In the ADCM UI, you can find this name on the ***Info*** tab of the ADPG service. Alternatively, it can be obtained from the output of the `list` Patroni command: `/usr/lib/adpg16/usr/bin/patronictl -c /etc/adpg16-patroni/adpg16-patroni.yml list`.

NOTE

Huge pages are configured individually for each host. Enable huge pages on all hosts where it is necessary.

Data directory structure

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

Contents of the cluster data directory

Relation files

Tablespace files

Obtain path to relation file

This article describes a data directory structure of an ADPG cluster.

ADPG stores data files in the directory specified by the ***Data directory*** configuration parameter. The default value is `/pg_data1`. You can change this value during cluster installation before the ADPG service is installed.

Contents of the cluster data directory

The directory specified in ***Data directory*** contains a subdirectory for the current cluster. For ADPG v16.x.x, it is *adpg16*. The contents of the cluster subdirectory are listed in the table below.

Item	Description
PG_VERSION	File containing the ADPG/PostgreSQL major version number
base	Directory containing per-database subdirectories
current_logfiles	File that records what log files the logging collector currently writes to
global	Directory containing cluster-wide tables, such as `pg_database`
pg_commit_ts	Directory containing transaction commit timestamp data
pg_dynshmem	Directory containing files used by the dynamic shared memory subsystem
pg_logical	Directory containing status data of subscriptions or replication slots for logical decoding
pg_multixact	Directory containing multi-transaction status data used for shared row locks
pg_notify	Directory containing `LISTEN/NOTIFY` status data
pg_replslot	Directory containing replication slot data
pg_serial	Directory containing information about committed serializable transactions
pg_snapshots	Directory containing exported snapshots
pg_stat	Directory containing permanent files for the statistics subsystem
pg_stat_tmp	Directory containing temporary files for the statistics subsystem
pg_subtrans	Directory containing subtransaction status data
pg_tblspc	Directory containing symbolic links to tablespaces
pg_twophase	Directory containing state files for prepared transactions
pg_wal	Directory containing WAL files
pg_xact	Directory containing transaction commit status data
postgresql.auto.conf	File used for storing configuration parameters that are set by the `ALTER SYSTEM` SQL command
postmaster.opts	File recording the command-line options the server was last started with
postmaster.pid	Lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (could be empty), the first valid value of the `listen_address` parameter (IP address, or `*`, or empty if not listening on TCP), and shared memory segment ID. This file is not present after server shutdown

For each database, there is a subdirectory within the *base* directory named after the database's OID. You can obtain OID from the pg_database system catalog:

```
SELECT oid, datname FROM pg_database;
```

```
 oid |  datname
-----+-----------
   5 | postgres
   1 | template1
   4 | template0
```

To list the contents of the *base* directory, run the following command:

```
$ cd /pg_data1/adpg16/base && ls
```

Result:

```
1  4  5
```

The database subdirectory is the default location for the database's files including system catalogs.

Relation files

Standard relations such as tables or indexes are stored in separate files. These files are named by the table or index `filenode` value, which is stored in the `relfilenode` column of the pg_class system catalog. As an example, display a relation name and `filenode` number for the first five records from `pg_class`:

```
SELECT relname, relfilenode FROM pg_class LIMIT 5;
```

```
       relname       | relfilenode
---------------------+-------------
 books_book_id_seq   |       16390
 books               |       16391
 books_pkey          |       16396
 authors_id_seq      |       16398
 authors             |       16399
```

Since the listed relations belong to the `postgres` database, the corresponding files with names `16398`, `16391`, `16396`, `16398`, and `16399` are located in the */pg_data1/adpg16/base/5* directory.

For temporary relations, the file name has the `t<backend>_<filenode>` format, where `<backend>` is the number of the backend process, which creates the file, and `<filenode>` is the `filenode` value. For example, `t8_16433`.

In addition to the main file, each table and index has a free space map, which stores information about free space available in the relation. The free space map has the same file name as a corresponding relation plus the `_fsm` suffix. For example, `16432_fsm`.

Tables also have visibility maps to track pages that do not contain dead tuples. A visibility map is stored using the same file name as a corresponding table with the `_vm` suffix. For example, `16432_vm`.

Unlogged tables and indexes have an additional file known as the initialization fork, which is stored with the suffix `_init`.

Note that although a table's `filenode` often matches its OID, this is not always the case. Some operations as `TRUNCATE`, `REINDEX`, `CLUSTER`, and some `ALTER TABLE` forms can change `filenode` while preserving the OID.

When a table or index exceeds 1 GB, it is divided into segments with a maximum size of 1 GB. The first segment's file name is the same as `filenode`. Subsequent segments are named `filenode.1`, `filenode.2`, etc. This arrangement avoids problems on platforms that have file size limitations.

If a table contains columns with potentially large entries, a TOAST table is created to store these large values separately. The main table is linked to its TOAST table via the `reltoastrelid` field of the pg_class system catalog. For more information, see TOAST.

Temporary files required for resource-consuming operations are created within *base/pgsql_tmp*. The name of a temporary file has the following format: `pgsql_tmp<PID>.<Number>`, where `<PID>` is the PID of the owning backend process, `<Number>` distinguishes different temporary files of the current backend process. For example, `pgsql_tmp29522.1`.

Tablespace files

Each user-defined tablespace has a symbolic link in the *pg_tblspc* subdirectory located in the cluster data directory (by default, */pg_data1/adpg16*), which points to the physical tablespace directory – the location specified in the `CREATE TABLESPACE` command. This symbolic link is named according to the tablespace OID. Inside the physical tablespace directory, ADPG creates a subdirectory with a name that depends on the ADPG version, for example, *PG_16_202307071*. Within this subdirectory, there is a subdirectory for each database that has elements in the tablespace. It is named according to the database's OID. Tables and indexes are stored within this directory using the `filenode` naming scheme described above.

The `pg_default` tablespace is not accessed through *pg_tblspc*, but corresponds to the *base* subdirectory located in the cluster data directory. Similarly, the `pg_global` tablespace is not accessed through *pg_tblspc*, but corresponds to *global*.

Temporary files are created within the *pgsql_tmp* subdirectory of a tablespace directory.

Obtain path to relation file

The `pg_relation_filepath(relation regclass)` function returns the path relative to the cluster data directory for any relation. Keep in mind that this function gives the name of the first segment related to the current relation. You may need to append a segment number and/or the `_fsm`, `_vm`, or `_init` suffix to obtain all the files associated with the relation.

Example:

```
SELECT pg_relation_filepath('books');
```

```
 pg_relation_filepath
----------------------
 base/5/16391
```

It is also possible to display the relation name and its path to a file for each relation from the `pg_class` system catalog:

```
SELECT relname, pg_relation_filepath(oid) FROM pg_class;
```

```
 books_book_id_seq                | base/5/16390
 books                            | base/5/16391 1
 books_pkey                       | base/5/16396
 authors_id_seq                   | base/5/16398
 authors                          | base/5/16399
 authors_pkey                     | base/5/16403
 pg_statistic                     | base/5/2619
 pg_type                          | base/5/1247
 pgbench_accounts                 | base/5/16422
 pgbench_branches                 | base/5/16423
 pgbench_tellers                  | base/5/16425
 pgbench_branches_pkey            | base/5/16426
 pgbench_tellers_pkey             | base/5/16428
 pgbench_accounts_pkey            | base/5/16438
 pgbench_history                  | base/5/16457
 pg_toast_1255                    | base/5/2836
 pg_toast_1255_index              | base/5/2837
...
 table1                           | pg_tblspc/16451/PG_16_202307071/5/16454 2
...
```

1. Table from the default tablespace.
2. Table from a user-defined tablespace.

Databases

Eugenia Kuzina

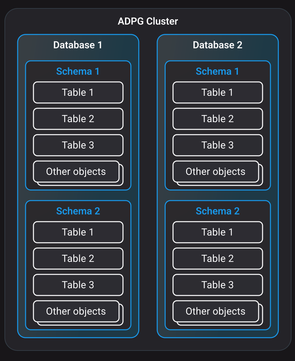
Contents

To Table of Contents

Overview
View existing databases
Create a database
Delete a database

Overview

A **database** is the topmost hierarchical level for organizing SQL objects (database objects). The full hierarchy is cluster → database → schema → table and other objects (functions, views, operators).



Logical layer of ADPG cluster

An ADPG cluster can contain multiple databases, but it is possible to establish a connection to only one database. However, you can open multiple connections to different databases or a single database.

Databases are physically separated. By default, ADPG places the database files at the following path: */pg_data1/adpg16*. ADPG creates the *adpg16* directory at the path specified in the `Data directory` parameter (see Configuration parameters). The default value of this parameter is `/pg_data1`.

After installation, an ADPG cluster contains three databases described in the table below.

Template and default databases

Name	Description
postgres	This is a default database for initial connections. It is a copy of `template1` and can be dropped and recreated if necessary
template0	This is a template database. It does not accept connections and should not be changed. You can specify this template in the CREATE DATABASE command to create an initial database without changes made in `template1`
template1	This is a template database. It contains the same data as `template0`, but you can change it. By default, `template1` is used for each new database that you create with the `CREATE DATABASE` command. You can add objects that should be in all new databases to this template. For example, add tables or install procedural languages

View existing databases

You can use the `\l` psql meta-command or `-l` command-line option to view existing databases:

```
$ psql

\l
```

```
$ psql -l
```

The result:

```
   Name    |  Owner   | Encoding |   Collate   |    Ctype    |   Access privileges
-----------+----------+----------+-------------+-------------+-----------------------
 postgres  | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | =c/postgres          +
           |          |          |             |             | postgres=CTc/postgres
```

Alternatively, select data from the pg_database catalog:

```
SELECT oid, datname, datistemplate, datallowconn FROM pg_database;
```

The result:

```
  oid  |  datname  | datistemplate | datallowconn
-------+-----------+---------------+--------------
 14486 | postgres  | f             | t
     1 | template1 | t             | t
 14485 | template0 | t             | f
```

In the `datallowconn` column, you can see that connections to `template0` are not allowed.

The following command shows the database you connected to:

```
SELECT current_database();
```

To determine the database size, use the `pg_size_pretty` and `pg_database_size` functions as follows:

```
SELECT pg_size_pretty(pg_database_size('postgres'));
```

Create a database

To create your first database, connect to the `postgres` database. The `psql` utility connects to the `postgres` database by default. You should also have the correct privileges for this operation or be a superuser.

Execute the CREATE DATABASE command to create a database. The following command creates the `book_store` database:

```
CREATE DATABASE book_store;
```

The current role automatically becomes the owner of the new database. It is also possible to specify another database owner. The command below makes `role1` the owner of the newly created `book_store` database:

```
CREATE DATABASE book_store OWNER role1;
```

You can also specify the template for a new database. The following code creates the `book_store` database by copying `template0`:

```
CREATE DATABASE book_store TEMPLATE template0;
```

Delete a database

Only the database owner or superuser can drop a database. This operation deletes all objects contained in the database. Deleting a database is an irreversible operation.

Use the DROP DATABASE command to delete a database:

```
DROP DATABASE book_store;
```

It is not possible to execute the `DROP DATABASE` command while there is at least one connection to the specified database. However, you can connect to any other, including `template1`. The `template1` database may be the only option if you need to drop the last user database of the cluster.

Schemas

Evgeniy Kozina

AD.PG

Contents

To Table of Contents

- Overview
- Manage schemas
- The default schema
- The schema search path
- Privileges on schemas
- Usage patterns
  - Secure schema usage pattern
  - Delete the public schema from the search path
  - The default pattern

Overview

A **schema** is a namespace that contains named database objects such as tables, views, indexes, data types, functions, stored procedures, and operators. A database can include one or more schemas. Schemas belong to the logical layer of the data model.

Logical layer of ADPG cluster

The distribution of objects in different schemas can be helpful in the following cases:

- to allow many users to use one database without interfering with each other;
- to divide objects into logical groups to manage them effectively;
- put third-party applications into separate schemas, so they do not collide with the names of other objects.

Schemas cannot be nested within each other.

Manage schemas

Use the `CREATE SCHEMA` command to create a schema:

```
CREATE SCHEMA schema1;
```

Schema names beginning with `pg_` are reserved for system purposes and cannot be created by users.

To create or access objects in a schema, write a fully qualified name consisting of the schema name and the table name separated by a dot:

```
CREATE TABLE schema1.table1 (
    column1 serial PRIMARY KEY,
    column2 VARCHAR (50) UNIQUE NOT NULL
);
```

It is possible to use the same object names in different schemas without conflicts. Create a new schema and add `table1` to it:

```
CREATE SCHEMA schema2;

CREATE TABLE schema2.table1 (
    column1 serial PRIMARY KEY,
    column2 VARCHAR (50) UNIQUE NOT NULL
);
```

Utilize qualified names to access these tables:

```
SELECT * FROM schema1.table1;

SELECT * FROM schema2.table1;
```

You can use the `ALTER SCHEMA` command to rename a schema or change its owner. For example:

```
ALTER SCHEMA schema2 RENAME TO myschema;
```

To drop a schema, execute the `DROP SCHEMA` command. The schema should be empty for this operation. If you need to delete a schema with all contained objects, use the `CASCADE` keyword:

```
DROP SCHEMA myschema CASCADE;
```

For more information about the deletion of dependent objects, see Dependency tracking.

The default schema

If you create a table or any other object without specifying a schema name, PostgreSQL puts this object into the `public` schema. Each new database contains this schema and uses it by default. The following code lines are equivalent:

```
CREATE TABLE books ( ... );

CREATE TABLE public books ( ... );
```

The command below returns the current schema:

```
SELECT current_schema();
```

The result:

```
 current_schema
----------------
 public
(1 row)
```

To list all available schemas, use the `\dn psql` meta-command.

The result:

```
  List of schemas
  Name  |  Owner
--------+----------
public  | postgres
schema1 | postgres
```

Alternatively, select the `nspname` column from the pg_namespace catalog:

```
SELECT nspname FROM pg_namespace;
```

The result:

```
      nspname
--------------------
 pg_toast
 pg_catalog
 public
 information_schema
```

In addition to `public`, the result includes PostgreSQL system schemas. The table below describes them:

System schemas

Name	Description
pg_toast	Contains TOAST tables
information_schema	Consists of a set of views with information about the current database objects
pg_catalog	Contains the system tables and all the built-in data types, functions, and operators

The schema search path

In most cases, we refer to tables by unqualified names (without schema names) in queries. The system uses a **search path** to determine which table should be processed. The search path is a list of schemas to look in. PostgreSQL operates with the first matching table. If there is no match in the search path schemas, an error occurs, even if matching table names exist in other schemas that the search path does not include.

To display the current search path, use the following command:

```
SHOW search_path;
```

The result:

```
 search_path
--------------
 "$user", public
```

The first element `"$user"` refers to a schema with the same name as the current user. If this schema does not exist, PostgreSQL ignores this entry. The second element is the `public` schema. The first existing schema in the search path is the default location for new objects. That is why objects are created in the `public` schema by default.

To add a new schema in the search path, use the following command:

```
SET search_path TO schema1, public;
```

Since `schema1` is the first element in the path, new objects will be created in it.

The search path also works in the same way for data type, function, and operator names.

Note the `pg_catalog` schema mentioned above is a part of the search path. You can use the `current_schemas` function to check it. This function returns an array of the schema names from the effective search path, in their priority order. If the boolean argument is `true`, implicitly-searched system schemas are included in the result.

```
SELECT current_schemas(true);
```

The result:

```
    current_schemas
---------------------------
 {pg_catalog,public}
```

If `pg_catalog` is not explicitly included in the search path, PostgreSQL searches in it before inspecting path schemas. This behavior ensures that built-in objects are always be found. However, you can explicitly place `pg_catalog` at the end of your search path if you want user-defined object names to override built-in names.

Privileges on schemas

Users cannot access any objects in schemas they do not own. To allow that, the schema owner must grant the `USAGE` privilege on the schema. You may need to grant additional privileges on objects to allow users to manipulate objects in the schema. For example, if you want users to be able to create objects, grant them the `CREATE` privilege on the schema.

Everyone has `CREATE` and `USAGE` privileges on the `public` schema.

Usage patterns

The *Secure schema usage pattern* prevents untrusted users from changing queries of other users. When a database does not use this pattern, users who want to securely execute queries need to perform protective operations at the start of each session. They should set the search path to the empty string or remove schemas that are writable by non-superusers from the search path.

Secure schema usage pattern

Constrain ordinary users to user-private schemas and revoke their `CREATE` privilege from the `public` schema. To implement this, execute the following command:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Create a schema for each user with their name. Since the default search path starts with `$user`, which resolves to the user name, each user has a separate schema, and they access their own schemas. When applying this pattern to a database that untrusted users may already be connecting to, check if there are any objects in the `public` schema with the same names as the objects in the `pg_catalog` schema.

This pattern is secure unless an untrusted user is the database owner or has the `CREATEROLE` privilege.

Delete the public schema from the search path

Execute the following command to remove the `public` schema from the default search path:

```
ALTER ROLE ALL SET search_path = "$user";
```

Everyone retains the ability to create objects in the `public` schema, but only qualified names can be used for these objects. While qualified table references are safe, function calls in the `public` schema are unsafe and unreliable. If you create functions or extensions in the `public` schema, use the first pattern instead.

This pattern is secure unless an untrusted user is the database owner or has the `CREATEROLE` privilege.

The default pattern

If you keep the default settings, all users access the `public` schema implicitly. This simulates the situation where schemas are not available. However, this is not a secure pattern. It is acceptable only if the database has a single user or a few trusted users.

Tables overview

Supercharged

Contents

To Table of Contents

- Create a table
  - Examples
- Alter a table
- Specify default column values
- Define generated columns
- Use constraints
  - Check constraints
  - NOT NULL constraints
  - Unique constraints
  - Primary keys
  - Foreign keys
  - Exclusion constraints
- System columns

Tables are the fundamental building blocks of any relational database system. They store data in rows and columns, similar to a spreadsheet. Each column has a specific data type, which determines what kind of values can be stored in it. For example, a column might be defined to hold integers, text, dates, or more complex types like arrays or custom types. When creating a table, you must define the name and type of each column.

PostgreSQL includes a powerful system of built-in data types that meet the needs of most applications. You can also define your own data types. Most built-in data types have obvious names and semantics, so we defer detailed explanations to the next chapter. For example, we use several of the data types in the table `products`, which has a column named `name` of type `text`, a column named `price` of type `numeric`, and a column named `test_col` of type `timestamp`. Values that are fractional components of seconds are rounded to microseconds.

Create a table

To create a table, use the `CREATE TABLE` command(s). In this command, you need to specify a name for a new table, the names of columns, and the data type of each column. An example of the command to create the table `products`:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

You can enter this command into `psql` with the line editing commands, or you can use another interface. `psql` has the ability to read commands from a file. To do this, type: `psql -f filename`.

To remove a table, use the `DROP TABLE` command. For example, `DROP TABLE products;`. Attempts to remove tables that don't exist will cause errors. Use `DROP TABLE IF EXISTS products;` to avoid errors.

If you need to modify a table that already exists, see the section on altering tables later in this chapter.

With the tools discussed so far, you can create fully functional tables. The remainder of this chapter is concerned with adding features to the table definition to ensure data integrity, security, or convenience. If you are eager to fill your tables with data now, you can skip ahead to the next chapter and read the rest of this chapter later.

Examples

Create a table with NOT NULL and PRIMARY KEY constraints

```
CREATE TABLE books (
    book_id     INTEGER CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(400) NOT NULL,
    date_pub    date,
    para        varchar(10)
);
```

Create a temporary table

```
CREATE TEMP TABLE short (
    num     INTEGER CONSTRAINT firstkey PRIMARY KEY,
    title   varchar(400) NOT NULL,
    date_pub date,
    para    varchar(10)
);
```

Use INHERITS

```
CREATE TABLE book_location (
  ) INHERITS (books);
```

Use LIKE

```
CREATE TABLE book_location2 (
    ADDITIONAL STRING
) LIKE books INCLUDING ALL;
```

Drop a table

If you no longer need a table you can use the `DROP TABLE` command to remove it.

```
DROP TABLE books;
```

Attempting to drop a non-existent table causes an error. To avoid it you can use the `IF EXISTS` clause.

```
DROP TABLE IF EXISTS books;
```

Specify default column values

You can assign a default value to a column using the `DEFAULT` keyword in the CREATE TABLE command. The default value is inserted into any row that does not explicitly specify a value for that column.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 47.99
);
```

In the above table definition, values are not often specified for the column price type:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
```

The default value can be an expression, which is evaluated when the default value is inserted. For example, a `DEFAULT` clause using the function `current_timestamp` sets the default value to the time the row is inserted:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    ordered_time timestamp DEFAULT current_timestamp
);
```

If no value is specified for the column `ordered_time` in an `INSERT` statement, the value returned by `current_timestamp` is used.

PostgreSQL also allows you to define commands to be executed when inserting new rows or updating existing rows. These commands are called triggers and are discussed in detail in the chapter on triggers. Triggers can be used to set default values for columns, but they are more flexible than the `DEFAULT` clause. For example, a trigger can set a default value based on another column in the row or on data from another table.

Define generated columns

A **generated column** is a special kind of column that is computed from other columns. PostgreSQL supports only stored generated columns, meaning the computed value is stored as part of the table's data.

To create a generated column, use the `GENERATED ALWAYS AS` clause in `CREATE TABLE`. It is also necessary to use the `STORED` keyword.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    sales_tax numeric GENERATED ALWAYS AS (price * 0.1) STORED
);
```

A generated column value cannot be specified in the `INSERT` or `UPDATE` commands, because it is impossible to write to the generated column. However, you can use the keyword `DEFAULT` to specify that you want the stored value of a generated column.

There are the following differences between generated columns and columns with a default value:

- The default value expression is evaluated once when the value is first assigned to the row. The value of the generated column is updated whenever the row changes, and cannot be overridden.
- The default value expression cannot refer to other columns in the table, but the generating expression refers to them.
- A column default can use volatile functions, for example, `CURRENT_TIMESTAMP`. Functions referring to the current time are not allowed for generated columns.

Use constraints

Boolean data types can be constrained to allow only one value, or perhaps only non-null values. For example, a constraint enforcing the price to be positive:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

Now, you might want to add a constraint to the `products` table requiring that the product name is not blank.

```
CREATE TABLE products (
    product_no integer,
    name text CHECK (name <> ''),
    price numeric CHECK (price > 0)
);
```

A `CHECK` constraint consists of the `CHECK` keyword, followed by an expression in parentheses. The `CHECK` constraint specifies that the expression must produce a true or unknown result for each row to be inserted. If any row of an `INSERT` command or an updated row causes the expression to produce a false result, an error is raised and the command is aborted. A check constraint is satisfied if the expression evaluates to true or the unknown result. It ensures that a column never contains a false result.

In this example, the product name and price are constrained to be non-null (the name was constrained to non-empty strings). Non-null constraints are described in the next section.

```
CREATE TABLE products (
    product_no integer,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

Note that the order of constraints does not matter. The order in which constraints are checked is not guaranteed.

A check constraint can also refer to several columns. Suppose you store the regular price and the discounted price, and you want to ensure that the discounted price is lower than the regular price:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

The first two constraints should look familiar. The third one introduces a new syntax. It is not part of the definition of any one column. It is an additional item in a comma-separated list of things. Column definitions and this kind of definition can be listed in any order. The idea is that you can say whatever you have to say about a table after you've named all its columns.

If a constraint is an expression, then it must appear after all the columns it needs to refer to. Since a column definition can only refer to the column it is for, it follows that any constraint referring to more than one column must be written separately.

You can also give explicit names to your constraints:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CONSTRAINT positive_price CHECK (price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

It is optional to name constraints, but it is highly recommended. Named constraints are more readable and also give you more control over how error messages are reported.

NOT NULL constraints

A `NOT NULL` constraint specifies that a column must not assume the `NULL` value.

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

A `NOT NULL` constraint is always written as a column constraint. A column constraint is exactly equivalent to any constraint, since it does not matter to which column you assign a constraint that applies to only one column.

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

It is also possible to specify a constraint on a table command:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

To define a unique constraint for a group of columns, write it as a table constraint with the column names separated by commas. You can also assign your own name for a unique constraint:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CONSTRAINT must_be_different UNIQUE (name, price)
);
```

Primary keys

A primary key constraint indicates that a column, or group of columns, can be used as a unique identifier for rows in the table. This requires that the values be both unique and not null. The following two table definitions accept the same data:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

A primary key can contain more than one column.

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    PRIMARY KEY (product_no, vendor_no)
);
```

Adding a primary key creates a unique B-tree index on the column or group of columns listed in the primary key, and forces the column(s) to be marked `NOT NULL`.

A table can have at most one primary key. There can be more than one unique and `NOT NULL` constraint, which is functionally almost equivalent to primary keys, but only one primary key can be named such as defined.

Foreign keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values from some row of another table. This maintains the **referential integrity** between two related tables.

Consider the following example (`products` table):

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Also, we have the `orders` table which has a column that refers to values in the `products` table. A row in `orders` table consists of products that will actually exist. So we define a foreign key constraint in the `orders` table that references the product table:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

Now it is impossible to create orders with non-null `product_no` entries that do not occur in the `products` table. The `orders` is said to be the **referencing** table and its `product_no` column is called the **referencing** column. Similarly, there are referencing and referenced columns.

You can omit a column list if the referenced table has a primary key, as the following example shows. This primary key will be automatically assigned.

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

In this case, the primary key of the referenced table is used as the referenced column.

A foreign key can also constrain and reference a group of columns. As usual, it then must be created using a table constraint. The number and type of the constrained columns need to match the number and type of the referenced columns.

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

A foreign key constraint can refer to the same table (**self-referential** foreign key). It allows you to implement a node tree structure.

```
CREATE TABLE tree (
    node_id integer PRIMARY KEY,
    parent_id integer REFERENCES tree,
    name text
);
```

The top-level nodes have `parent_id` as `NULL`, other `parent_id` entries refer to valid nodes.

A table can have more than one foreign key constraint. This is common for many-to-many relationship tables. Suppose you have tables for products and orders, but now you want to allow one order to contain possibly many products (which the earlier structure did not allow). You could use this table structure:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

The last table has a multi-column primary key. Note that the foreign key constraints also refer to this two-column key.

To ensure data integrity in tables for many-to-many relationships, you can specify the restrictions, even when updating and deleting rows. For example:

```
CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

The restrictions mentioned above let the product_no and order_id be deleted only if no rows in `order_items` table reference them. Deleting an order_id will cause the deletion of rows in the `order_items` table that reference the deleted order.

There are two options for changing the referenced data: `ON DELETE` and `ON UPDATE`. For each of these options, you can specify the action that should be taken when the referenced data is deleted or updated. The default action is `NO ACTION`, which means that any attempt to delete or update a row in the referenced table will cause an error. Other actions are `RESTRICT`, `CASCADE`, `SET NULL`, and `SET DEFAULT`. `RESTRICT` prevents deletion or updating of referenced rows. `CASCADE` specifies that when a referenced row is deleted or updated, the row(s) referencing it should be automatically deleted or updated as well. `SET NULL` and `SET DEFAULT` mean that the referencing column(s) will be set to nulls or their default values, respectively.

The main difference between `NO ACTION` and `RESTRICT` is that `NO ACTION` allows you to defer validation during a transaction, while `RESTRICT` does not.

A foreign key must reference columns that either are a primary key or form a unique constraint.

Exclusion constraints

Exclusion constraints ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return `FALSE` or `NULL`. For example:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

System columns

Every table has several system columns that are implicitly defined by the system. These names cannot be used as names of user-defined columns.

Name	Description
tableoid	The OID of the table containing this row. This column is particularly useful when `SELECT` from a `UNION` of `INHERIT`ed tables, to find out which table the row came from. The `REGCLASS` type can be used to cast this value to the table's name.
xmin	The identity (transaction ID) of the inserting transaction for this row version. A row version is an individual state of a row. Each update of a row creates a new row version for the same logical row.
cmin	The command identifier (starting at zero) within the inserting transaction.
xmax	The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or an attempted deletion was rolled back.
cmax	The command identifier within the deleting transaction, or zero.
ctid	The physical row location within its table. Note that although the `ctid` can be used to locate the row version very quickly, a row's `ctid` will change if it is updated or moved by `VACUUM FULL`. Therefore `ctid` is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

`xmin` and `xmax` values are 32-bit quantities. There is a possibility for wraparound failure in long-lived databases. Wraparound means that the clock is up-counted to zero and then continues from the minimum value. The remedy for this is to perform `VACUUM` operations on each table at least once per billion transactions. `VACUUM` operations are also necessary to remove dead row versions.

`VACUUM` operations are also necessary to update statistics used by the planner. Without statistics, it is difficult to choose between the many possible plans for executing a query.

`VACUUM` operations are performed in two contexts: periodic cleaning and during transaction wraparound prevention. Periodic cleaning is performed by processes that are part of the autovacuum facility. Transaction wraparound operations are performed only when `VACUUM` is manually invoked.

Inheritance

Eugenia Kuzina

To Table of Contents

The inheritance allows you to create a child table that get column descriptions from a parent table.

For example, we need to create the following tables:

```
CREATE TABLE books (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(50) NOT NULL,
    author VARCHAR(50) NOT NULL);

CREATE TABLE shelves (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(50) NOT NULL,
    author VARCHAR(50) NOT NULL,
    location VARCHAR(10) NOT NULL);
```

The second table has the same structure as the first one but includes an additional column. You can use the `INHERITS` clause to create the second table:

```
CREATE TABLE shelves (
    location VARCHAR(10) NOT NULL
) INHERITS (books);
```

In this case, `shelves` **inherits** all columns (`book_id`, `title`, and `author`) from its parent `books` table. The `shelves` table also has an additional column – `location`.

The use of `INHERITS` creates a persistent relationship between a new child table and a parent table. Parent schema modifications propagate to the child table, and the data of the child table is included in scans of the parent.

Add data to the tables:

```
INSERT INTO books (title, author) VALUES
('Hyperion', 'Dan Simmons'),
('1984', 'George Orwell');

INSERT INTO shelves (title, author, location) VALUES
('The Time Machine', 'Herbert George Wells', 'B32'),
('The Great Gatsby', 'F. Scott Fitzgerald', 'C14');
```

Select data from `books`:

```
SELECT * FROM books;
```

The result also includes rows of the `shelves` table:

```
 book_id |      title       |        author
---------+------------------+----------------------
       1 | Hyperion         | Dan Simmons
       2 | 1984             | George Orwell
       3 | The Great Gatsby | F. Scott Fitzgerald
       4 | The Time Machine | Herbert George Wells
```

The query returns all rows of the parent table and its descendant.

To return only the parent table rows, use the `ONLY` keyword:

```
SELECT * FROM ONLY books;
```

The result:

```
 book_id |  title   |    author
---------+----------+---------------
       1 | Hyperion | Dan Simmons
       2 | 1984     | George Orwell
```

You can use the `tableoid` system column and data from the pg_class catalog to determine from which table each row:

```
SELECT p.relname, b.title
FROM books b, pg_class p
WHERE b.tableoid = p.oid;
```

The result:

```
 relname |      title
---------+------------------
 books   | Hyperion
 books   | 1984
 shelves | The Great Gatsby
 shelves | The Time Machine
```

Parent tables can be ordinary tables or foreign tables. A table can inherit columns from more than one table. If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match each of the parent tables. If there is no conflict, the duplicate columns are merged into a single column in the new table.

If the column name list of the child table contains a column name that is also inherited from a parent, the data type of these columns must match, and the column definitions are merged into one. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations.

Child tables inherit all `CHECK` and `NOT NULL` constraints on the parent table unless a constraint description contains the `NO INHERIT` clause. Other types of constraints (`UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY`) are not inherited.

Columns or `CHECK` constraints of child tables cannot be dropped or altered if they are inherited from parent tables.

You can also add a new parent relationship to a table, which is already defined in a compatible way. A child table must already include columns with the same names and types as the columns of the parent. Use the `INHERIT` option of the `ALTER TABLE` command for this purpose:

```
CREATE TABLE books2 (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(50) NOT NULL,
    author VARCHAR(50) NOT NULL);

CREATE TABLE shelves2 (
    book_id SERIAL PRIMARY KEY,
    title VARCHAR(50) NOT NULL,
    author VARCHAR(50) NOT NULL,
    location VARCHAR(10) NOT NULL);

ALTER TABLE shelves2 INHERIT books2;
```

Operations with tables created by inheritance have the following specifics:

- The `SELECT`, `UPDATE`, and `DELETE` commands that are executed on the parent table also perform operations on descendant tables. You can use the `ONLY` keyword to avoid this behavior.
  The `INSERT` and `COPY` commands executed on the parent do not operate with descendants.
- PostgreSQL does not check access permissions for the child tables when processing them with their parent table. For example, granting the `UPDATE` permission on the `books` table implies the permission to update rows in the `shelves` table, when they are accessed through `books`.
- PostgreSQL cannot drop a parent table while any of its children remain. If you wish to remove a table and all of its descendants, drop the parent table with the `CASCADE` option:

```
DROP TABLE books CASCADE;
```

Partitioning

Ryan Booz

Contents

To Table of Contents

- Overview
- Declarative partitioning
  - Range partitioning
  - List partitioning
  - Hash partitioning
  - Partition maintenance
- Use inheritance to implement partitioning
  - Example
  - Maintenance for inheritance partitioning

Overview

PostgreSQL supports basic table **partitioning**. Partitioning allows you to split one large logical table into multiple smaller physical pieces. Partitioning is only useful for very large tables. Which tables will benefit from partitioning depends on the application, although it should generally be picked for tables that are larger than the server RAM.

Partitioning can provide the following benefits:

- Improve dramatically query performance in certain situations. For example, when the most part of the frequently requested rows is in one or only a few partitions. Partitioning effectively substitutes for the upper levels of large indexes, making it likely that the most requested parts of the indexes can be in memory.
- To improve the performance for queries that access a large percentage of rows from a single partition. PostgreSQL can use a sequential scan of this partition instead of an index, which requires random-reading of data scattered across the whole table.
- You can accomplish bulk loads and deletes by adding or removing partitions. The `DROP TABLE` command, which can be used to drop an individual partition, or `ALTER TABLE ... DETACH PARTITION` are faster than bulk operations with rows.
- Rarely used data can be migrated to cheaper and slower storage media.

Declarative partitioning

PostgreSQL allows you to specify conditions according to which a table is divided into partitions. The divided table is a **partitioned** table. The partitioning declaration includes a **partitioning method** and a list of columns or expressions to be used as a **partition key**. The partitioning methods are described in the table below.

Partitioning methods

Method	Description
Range partitioning	The table is partitioned into ranges defined by a key column or set of columns. For example, partitioning by date ranges, or by ranges of identifiers for particular business objects. The limits of each range are considered to include the lower value and exclude the upper value. For example, if one partition's range is from 1 to 10, and the second partition range is from 10 to 20, the value 10 belongs to the second partition, not the first.
List partitioning	The table is partitioned by a list that specifies which key values should be in each partition.
Hash partitioning	The table is partitioned by specifying a modulus and a remainder for partitions. Each partition holds the rows for which the hash value of the partition key divided by the specified modulus produces the specified remainder.

If your application needs to use other partitioning forms not listed above, you can use alternative methods, for example, [via inheritance to implement partitioning](). This method offers flexibility but it does not have performance benefits that are built-in to declarative partitioning.

Use the `CREATE TABLE` command to define a partitioned table and its partitions. The base syntax of the partitioned table creation is:

```
CREATE <table_name> PARTITION BY { RANGE | LIST | HASH } ( { <column_name> | ( <expression> ) } )
```

The base syntax of partitions:

```
CREATE TABLE <partition_name> PARTITION OF <table_name>
  { FOR VALUES FROM (<value_min>) TO (<value_max>)
  | FOR VALUES IN (<value>)
  | FOR VALUES WITH (MODULUS <modulus_value>, REMAINDER <remainder_value>) }
```

The partitioned table is a virtual table that does not hold rows. Partitions are ordinary tables associated with the partitioned one. Each partition stores a subset of rows defined by its partition key values. Operations that are applicable to ordinary tables can usually be applied to the partitioned one. If you update a row partition key value and it no longer satisfies the partition values, the row is moved into a different partition.

You cannot convert a regular table to a partitioned table or vice versa. However, you can add an existing table as a partition to a partitioned table, or detach a partition and turn it into a separate table.

Range partitioning

Range partitioning is often used with `TIMESTAMP` and `DATE` columns.

The table below creates columns and distributes books according to their `delivery_date`. Books delivered in *February* will be placed in the `books_2022_02` partition, books delivered in March — in `books_2022_03`, and in April — in `books_2022_04`. It also creates a default partition for books that do not suit other partitions:

```
CREATE TABLE books (
    code       char(5) NOT NULL,
    title      varchar(40) NOT NULL,
    delivery_date  date NOT NULL,
    genre      varchar(10)
) PARTITION BY RANGE (delivery_date);

CREATE TABLE books_2022_02 PARTITION OF books FOR VALUES FROM ('2022-02-01') TO ('2022-03-01');
CREATE TABLE books_2022_03 PARTITION OF books FOR VALUES FROM ('2022-03-01') TO ('2022-04-01');
CREATE TABLE books_2022_04 PARTITION OF books FOR VALUES FROM ('2022-04-01') TO ('2022-05-01');
CREATE TABLE books_default PARTITION OF books DEFAULT;
```

The following row is added to the `books_2022_04` partition:

```
INSERT INTO books (code, title, delivery_date, genre) VALUES
  ('EC-34', 'Hyperion', date'2022-04-20', 'sci-fi');
```

Also, the special `MINVALUE` and `MAXVALUE` values can be used when creating a partition to indicate that there is no lower or upper bound on the column value. For example:

```
CREATE TABLE table1_part1 PARTITION OF table1 FOR VALUES FROM (MINVALUE) TO (100);
CREATE TABLE table1_part2 PARTITION OF table1 FOR VALUES FROM (100) TO (1000);
CREATE TABLE table1_part3 PARTITION OF table1 FOR VALUES FROM (1000) TO (MAXVALUE);
```

List partitioning

The table below uses the `genre` column as the partition key for list partitioning. You can specify multiple values for one partition. The code below creates a default partition, a partition for novels, a partition for sci-fi books, and one partition for detective and mystery books:

```
CREATE TABLE books (
    code       char(5) NOT NULL,
    title      varchar(40) NOT NULL,
    delivery_date  date NOT NULL,
    genre      varchar(10) NOT NULL
) PARTITION BY LIST (genre);

CREATE TABLE books_novel PARTITION OF books FOR VALUES IN ('novel');
CREATE TABLE books_scifi PARTITION OF books FOR VALUES IN ('sci-fi');
CREATE TABLE books_detective PARTITION OF books FOR VALUES IN ('detective', 'mystery');
CREATE TABLE books_default PARTITION OF books DEFAULT;
```

Hash partitioning

You can use hash partitioning if you need to equally distribute rows across partitions and a table does not have an appropriate partition key.

Example:

```
CREATE TABLE books (
    code       char(5) NOT NULL,
    title      varchar(40) NOT NULL,
    delivery_date  date NOT NULL,
    genre      varchar(10) NOT NULL
) PARTITION BY HASH (code);

CREATE TABLE books_part1 PARTITION OF books FOR VALUES WITH (MODULUS 5, REMAINDER 0);
CREATE TABLE books_part2 PARTITION OF books FOR VALUES WITH (MODULUS 5, REMAINDER 1);
CREATE TABLE books_part3 PARTITION OF books FOR VALUES WITH (MODULUS 5, REMAINDER 2);
CREATE TABLE books_part4 PARTITION OF books FOR VALUES WITH (MODULUS 5, REMAINDER 3);
CREATE TABLE books_part5 PARTITION OF books FOR VALUES WITH (MODULUS 5, REMAINDER 4);
```

Partition maintenance

You can create an index on the key columns on the partitioned table or any other indexes you may need. It is not necessary to create a key index, but in most cases it is useful. This operation automatically creates a corresponding index on each partition, and any partitions that you create or append later will also contain this index. Indexes or unique constraints created on a partitioned table are virtual. In fact, the data resides in the child indexes of the partition tables.

The following example implies that the `delivery_date` column is used as the partitioning key and creates an index on it:

```
CREATE INDEX ON books (delivery_date);
```

Although all partitions must have the same columns as the partitioned parent table, each partition can have its own indexes, constraints, and default values.

> **NOTE**
>
> For query optimization, the `enable_partition_pruning` option must be set to `on`, which is the default value in *ADPG*. See Partition pruning for details.

Sub-partitioning

Partition tables can be partitioned to implement **sub-partitions**. For example, we can add sub-partitions to the list partitioning example:

```
CREATE TABLE books_novel PARTITION OF books FOR VALUES IN ('novel') PARTITION BY RANGE (delivery_date);
CREATE TABLE books_novel_before2000 PARTITION OF books FOR VALUES IN ('novel') AND (delivery_date) < ('2000-01-01');
CREATE TABLE books_scifi PARTITION OF books FOR VALUES IN ('sci-fi') PARTITION BY RANGE (delivery_date);
CREATE TABLE books_scifi_2022_02 PARTITION OF books_scifi FOR VALUES FROM ('2022-02-01') TO ('2022-03-01');
CREATE TABLE books_scifi_2022_03 PARTITION OF books_scifi FOR VALUES FROM ('2022-03-01') TO ('2022-04-01');
```

PostgreSQL inserts the following row to the `books_scifi_2022_02` table:

```
INSERT INTO books (code, title, delivery_date, genre) VALUES
  ('KC-29', 'The time machine', date'2022-02-28', 'sci-fi');
```

Remove a partition

Each of our partitions can be changed. It is a standard operation to remove partitions with obsolete data and add partitions for new data. One of the most important benefits of partitioning is that it allows you to perform resource-consuming operations instantly by changing the structure of the partitions, rather than physically moving large amounts of data.

You can use the `DROP TABLE` command to drop a partition that is no longer needed:

```
DROP TABLE books_novel;
```

This operation can delete millions of records quickly but requires the `ACCESS EXCLUSIVE` lock on the partitioned table.

Another option is to remove a partition from the partitioned table but retain access to it as an ordinary table. It allows further operations to be performed on the data before it is dropped. It has two forms:

```
ALTER TABLE books DETACH PARTITION books_novel;
ALTER TABLE books DETACH PARTITION books_novel CONCURRENTLY;
```

The first form of the command requires the `ACCESS EXCLUSIVE` lock on the parent table. The command with the `CONCURRENTLY` qualifier utilizes only the `SHARE UPDATE EXCLUSIVE` lock. See `DETACH PARTITION` for details.

Add a partition

You can use the following code to add a new partition:

```
CREATE TABLE books_2022_05 PARTITION OF books FOR VALUES FROM ('2022-05-01') TO ('2022-06-01');
```

Alternatively, it is possible to create a new table outside the partition structure and use the `ATTACH PARTITION` clause to make it a partition later. This allows data to be loaded, checked, and transformed before it appears in the partitioned table:

```
CREATE TABLE books_2022_06 (LIKE books INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE books_2022_06 ADD CONSTRAINT month_2022_06
  CHECK ( delivery_date >= DATE '2022-06-01' AND delivery_date < DATE '2022-07-01' );
\d+ books_2022_06
ALTER TABLE books ATTACH PARTITION books_2022_06
  FOR VALUES FROM ('2022-06-01') TO ('2022-07-01');
```

The `ATTACH PARTITION` clause requires the `SHARE UPDATE EXCLUSIVE` lock on the partitioned table. It is recommended to create a `CHECK` constraint on the attached table that matches the new partition constraint, as illustrated above. This way the scan when is needed to validate the implicit partition constraint.

Limitations

The following limitations are applied to partitioned tables:

- To create a unique or primary key constraint on a partitioned table, the partition keys must not include any expressions or function calls and the constraint's columns must include all the partition key columns.
- It is impossible to create an exclusion constraint for the whole partitioned table. You can only put such a constraint on each partition individually.
- The `BEFORE ROW` triggers on `INSERT` cannot change the partition into which a new row will be inserted.
- It is impossible to mix temporary and permanent relations in the same partitioned table.
- A partition table cannot have any parents other than the partitioned table.
- Partitions cannot include columns that do not exist in the partitioned table.
- The `CHECK` and `NOT NULL` constraints of a partitioned table are always inherited by all its partitions. You cannot drop these constraints on a partition column if the same constraint exists in the partitioned table.

Use inheritance to implement partitioning

While the built-in declarative partitioning is suitable for most cases, there are some circumstances where a more flexible approach may be useful. Partitioning can be implemented by table inheritance, which allows for several features not supported by declarative partitioning, such as:

- For declarative partitioning, partitions must have the same set of columns as the partitioned table, whereas with table inheritance, child tables can include extra columns not existing in the parent.
- Table inheritance allows for multiple inheritance.
- Declarative partitioning only supports range, list, and hash partitioning, while table inheritance allows the partitioning of data according to the user's choice. Note, if constraint exclusion is unable to prune child partition tables effectively, query performance might be poor.

Example

This example creates a partitioning structure equivalent to the declarative range partitioning example above.

Create the parent table, from which all the child tables inherit structure:

```
CREATE TABLE books (
    code       char(5) NOT NULL,
    title      varchar(40) NOT NULL,
    delivery_date  date NOT NULL,
    genre      varchar(10)
);
```

Create several child tables that inherit from the parent and add non-overlapping constraints to define the allowed partitioning key values in each table:

```
CREATE TABLE books_2022_02 (
CHECK ( delivery_date >= DATE '2022-02-01' AND delivery_date < DATE '2022-03-01' )
) INHERITS (books);

CREATE TABLE books_2022_03 (
CHECK ( delivery_date >= DATE '2022-03-01' AND delivery_date < DATE '2022-04-01' )
) INHERITS (books);

CREATE TABLE books_2022_04 (
CHECK ( delivery_date >= DATE '2022-04-01' AND delivery_date < DATE '2022-05-01' )
) INHERITS (books);
```

For each child table, create an index on the key column and any other indexes if you need:

```
CREATE INDEX books_2022_02_delivery_date ON books_2022_02 (delivery_date);
CREATE INDEX books_2022_03_delivery_date ON books_2022_03 (delivery_date);
CREATE INDEX books_2022_04_delivery_date ON books_2022_04 (delivery_date);
```

PostgreSQL should insert data into an appropriate child table. To do this, attach a suitable trigger function to the parent table:

```
CREATE OR REPLACE FUNCTION books_select_part()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.delivery_date BETWEEN date'2022-02-01' AND date'2022-03-01' ) THEN
        INSERT INTO books_2022_02 VALUES (NEW.*);
    ELSIF ( NEW.delivery_date BETWEEN date'2022-03-01' AND date'2022-04-01' ) THEN
        INSERT INTO books_2022_03 VALUES (NEW.*);
    ELSIF ( NEW.delivery_date BETWEEN date'2022-04-01' AND date'2022-05-01' ) THEN
        INSERT INTO books_2022_04 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'This date not in your partitions. Add a partition.';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Create a trigger that calls the function:

```
CREATE TRIGGER insert_books_trigger
    BEFORE INSERT ON books
    FOR EACH ROW EXECUTE FUNCTION books_select_part();
```

In practice, it is best to check the condition on the last child table first if rows are inserted into it most often. For simplicity, we arranged the trigger tests in the same order as in the other code snippets for this example.

> **NOTE**
>
> If you use this type of partitioning, the `constraint_exclusion` parameter should not be disabled. Otherwise, child tables may be scanned when not required. In *ADPG*, the default value is `partition`.

Maintenance for inheritance partitioning

Remove a partition

You can use the `DROP TABLE` command to drop a child table that is no longer needed:

```
DROP TABLE books_2022_01;
```

To remove the child table from the inheritance hierarchy but retain access to it as an ordinary table, use the `ALTER TABLE` command:

```
ALTER TABLE books_2022_02 NO INHERIT books;
```

Add a partition

To add a new child table to handle new data, create an empty child table:

```
CREATE TABLE books_2022_05 (
CHECK ( delivery_date >= DATE '2022-05-01' AND delivery_date < DATE '2022-06-01' )
) INHERITS (books);
```

Alternatively, it is possible to create and populate a new child table before adding it to the table hierarchy. This allows new data to be loaded, checked, and transformed before it appears in the parent table:

```
CREATE TABLE books_2022_06 (LIKE books INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE books_2022_06 ADD CONSTRAINT month_2022_06
  CHECK ( delivery_date >= DATE '2022-06-01' AND delivery_date < DATE '2022-07-01' );
\d+ books_2022_06
ALTER TABLE books_2022_06 INHERIT books;
```

You also need to update a trigger function:

```
CREATE OR REPLACE FUNCTION books_select_part()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.delivery_date BETWEEN date'2022-02-01' AND date'2022-03-01' ) THEN
        INSERT INTO books_2022_02 VALUES (NEW.*);
    ELSIF ( NEW.delivery_date BETWEEN date'2022-03-01' AND date'2022-04-01' ) THEN
        INSERT INTO books_2022_03 VALUES (NEW.*);
    ELSIF ( NEW.delivery_date BETWEEN date'2022-04-01' AND date'2022-05-01' ) THEN
        INSERT INTO books_2022_04 VALUES (NEW.*);
    ELSIF ( NEW.delivery_date BETWEEN date'2022-05-01' AND date'2022-06-01' ) THEN
        INSERT INTO books_2022_05 VALUES (NEW.*);
    ELSIF ( NEW.delivery_date BETWEEN date'2022-06-01' AND date'2022-07-01' ) THEN
        INSERT INTO books_2022_06 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'This date not in your partitions. Add a partition.';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Limitations

The partitioning implemented with inheritance has the following limitations:

- PostgreSQL cannot automatically check if all `CHECK` constraints are mutually exclusive. It is safer to write code that generates child tables and creates or modifies associated objects.
- Indexes and foreign key constraints apply to single tables and not to their inheritance children.
- The example above assumes that the key column values cannot be changed to move a row to another partition. An attempt to do that fails because of the `CHECK` constraints. If you need to handle such cases, you can put appropriate update triggers on the child tables.
- If you utilize the `VACUUM` or `ANALYZE` commands, you need to run them on the parent and each child table individually.
- The `INSERT` statements with the `ON CONFLICT` clauses do not work as expected because the `ON CONFLICT` action executes only in case of unique violations on the specified target relation, not its child relations.
- You need to create triggers to route rows to child tables. Triggers are complicated to write, and they are slower than the routing performed internally by declarative partitioning.

Foreign tables

© Eugene Kozina

Contents

To Table of Contents

Overview
Examples
- Access data stored in external PostgreSQL servers
- Access data files in the server file system

Overview

PostgreSQL partially implements the SQL/MED specification. It allows you to use regular SQL queries to access data that resides outside PostgreSQL. This data is called third-party data. To access it, follow the steps below:

1. Use a **foreign data wrapper**. A foreign data wrapper connects to external data sources and obtains data from them. The contrib extension included in the ADPG installation contains multiple foreign data wrappers. See Additional Supplied Modules for details. You can also use third-party data wrappers or write your own. See Writing a foreign data wrapper.
2. Define a **foreign server** object that encapsulates connection information for the corresponding foreign data wrapper. Use the CREATE SERVER command for this:

```
CREATE SERVER [ IF NOT EXISTS ] <server_name> [ TYPE '<server_type>' ] [ VERSION '<server_version>' ]
FOREIGN DATA WRAPPER <fdw_name>
[ OPTIONS ( <option 'value' [, ... ]> ) ]
```

Parameters of the CREATE SERVER command

IF NOT EXISTS	When this option is specified, PostgreSQL does not throw an error if a server with the specified name exists. There is no guarantee that the existing server matches the one that would be created
<server_name>	The name of a new foreign server. The server name must be unique within the database
<server_type>	An optional parameter that specifies a server type. It can be useful to foreign data wrappers
<server_version>	An optional parameter that specifies a server version. It can be useful to foreign data wrappers
<fdw_name>	A name of a foreign data wrapper that manages the foreign server
OPTIONS (<option 'value' [, ...]>)	Specifies the server options. These options define the connection details, but the names and values are dependent on the specified foreign data wrapper

The user, who defines the server, becomes its owner. To create the server, the user should have the **USAGE** privilege on the specified foreign data wrapper.

3. Access to remote data may require authentication to an external data source. Create a **user mapping** object to specify a user name and password that can be used for the authentication of a current PostgreSQL role. To do this, execute the CREATE USER MAPPING command that has the following syntax:

```
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { <user_name> | USER | CURRENT_ROLE | CURRENT_USER | PUBLIC }
  SERVER <server_name>
  [ OPTIONS ( <option 'value' [, ... ]> ) ]
```

Parameters of the CREATE USER MAPPING command

IF NOT EXISTS	Do not throw an error if a mapping of the given user to the given foreign server already exists. There is no guarantee that the existing user mapping matches the one that would be created
<user_name>	Specifies the name of an existing user that is mapped to a foreign server. **CURRENT_ROLE**, **CURRENT_USER**, and **USER** match the name of the current user. When **PUBLIC** is specified, the public mapping is created that is used when no user-specific mapping is applicable
<server_name>	Defines a name of an existing server for which the user mapping is created
OPTIONS (<option 'value' [, ...]>)	Specifies the options for user mapping. The options typically define the actual remote user name and password. The allowed option names and values are specific to the used foreign data wrapper

The owner of a foreign server can create user mappings for that server for any user. A user also can create a user mapping for their own user name if the **USAGE** privilege on the server is granted to the user

4. Create one or more **foreign tables** that define the remote data structure. A foreign table does not store data on the PostgreSQL server, but you can execute queries on it as on a regular table. PostgreSQL utilizes a foreign data wrapper to fetch data from an external source or transmit data to this source in case of update commands. Execute the CREATE FOREIGN TABLE command to define a foreign table:

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] <table_name> ( [
  { <column_name> <data_type> [ COLLATE <collation> ] [ <column_constraint> [ ... ] ] | <table_constraint> }
  [, ... ]
] )
[ INHERITS ( <parent_table> [, ... ] ) ]
  SERVER <server_name>
[ OPTIONS ( <option 'value'> [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] <table_name>
  PARTITION OF <main_table> [ (
  { <column_name> [ WITH OPTIONS ] [ <column_constraint> [ ... ] ]
    | <table_constraint> }
    [, ... ]
) ] }
{ FOR VALUES <partition_bound_spec> | DEFAULT }
  SERVER <server_name>
[ OPTIONS ( <option 'value' [, ... ]> ) ]
```

Parameters of the CREATE FOREIGN TABLE command

IF NOT EXISTS	When this option is specified, PostgreSQL does not throw an error if a relation with the specified name is in the database. There is no guarantee that the existing relation matches the one that would be created
<table_name>	The name of a new table. Can be schema-qualified (**<schema_name>.<table_name>**)
<column_name>	The name of a column to be created in the new table
<data_type>	The column data type. This can include array specifiers. For more information on supported data types, refer to Data types
COLLATE <collation>	Defines the column collation. If this setting is not specified, PostgreSQL uses the column data type's default collation
INHERITS (<parent_table> [, ...])	Specifies a list of tables from which the new table automatically inherits all columns. Parent tables can be ordinary tables or foreign tables. See Inheritance
PARTITION OF <parent_table>	Creates a foreign table as a partition of the given parent table with specified partition bound values. See Partitioning for more information on partitioning options. Note that it is not allowed to create the foreign table as a partition of the parent table if there are **UNIQUE** indexes on the parent table
<column_constraint>	Defines column constraints that are described in the Use constraints section
<table_constraint>	Specifies table constraints that are described in the Use constraints section
<server_name>	Defines the name of an existing foreign server to use for a foreign table
OPTIONS (<option 'value' [, ...]>)	Specifies options associated with a new foreign table or one of its columns. The allowed option names and values are specific to each foreign data wrapper

Alternatively, you can import the entire foreign schema. To do this, execute the IMPORT FOREIGN SCHEMA command. It creates foreign tables that represent tables existing on a foreign server. PostgreSQL creates foreign tables with column definitions and options that match the remote tables. The user, who executes the **IMPORT FOREIGN SCHEMA** command, is the owner of new foreign tables. The command has the following syntax:

```
IMPORT FOREIGN SCHEMA <remote_schema>
  [ { LIMIT TO | EXCEPT } ( <table_name> [, ...] ) ]
  FROM SERVER <server_name>
  INTO <local_schema>
  [ OPTIONS ( <option 'value' [, ... ]> ) ]
```

Parameters of the IMPORT FOREIGN SCHEMA command

<remote_schema>	Defines a remote schema to import from. What exactly a remote schema is depends on the used foreign data wrapper
LIMIT TO (<table_name> [, ...])	Imports only foreign tables matching one of the given table names. Other tables existing in the foreign schema are ignored
EXCEPT (<table_name> [, ...])	Excludes specified foreign tables from the import. All tables existing in the foreign schema are imported except the ones listed in this clause
<server_name>	Defines the foreign server to import from
<local_schema>	Sets the schema in which the imported foreign tables are created
OPTIONS (<option 'value' [, ...]>)	Specifies options to be used during the import. The allowed option names and values are specific to each foreign data wrapper

The list of imported tables can be limited to a specified subset, or specific tables can be excluded from import. The target local schema must already exist.

You can **IMPORT FOREIGN SCHEMA** the user must have the **USAGE** privilege on the foreign server and the **CREATE** privilege on the target local schema.

To display foreign tables from the current database, use the following query:

```
SELECT * FROM information_schema.foreign_tables;
```

Examples

The contrib extension contains the following data wrappers:

- **postgres_fdw** – can be used to access data stored in external PostgreSQL servers. See postgres_fdw.
- **file_fdw** – can be used to access data files in the server file system, or to execute programs on the server and read their output. See file_fdw.

You can find examples of using these data wrappers below.

Access data stored in external PostgreSQL servers

Assume that an external PostgreSQL server contains the **books** and **authors** tables in the **books_store** database.

The **authors** table is created as follows:

```
CREATE TABLE authors (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);
```

To access data from this server, follow the steps below:

1. Use an existing role on a remote server or create a new user and grant appropriate permissions. For example:

```
CREATE ROLE user1 WITH PASSWORD 'password' LOGIN;

GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO user1;

GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA public TO user1;
```

2. Since ADPG includes the contrib extension, you just need to execute the **CREATE EXTENSION** command to use the **postgres_fdw** data wrapper on the local server.

```
CREATE EXTENSION postgres_fdw;
```

3. Create a foreign server in your local database. The following code creates a server that connects to the **books_store** database located on the **10.92.6.225** host:

```
CREATE SERVER server1 FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host '10.92.6.225', port '5432', dbname 'books_store');
```

If the remote server uses the standard port, the **port** option can be omitted.

4. Define user mapping. Pass the credentials of the user you created on the remote server as options. The following code creates user mapping for the **postgres** user from the local server:

```
CREATE USER MAPPING FOR postgres SERVER server1 OPTIONS (user 'user1', password 'password');
```

5. Create a foreign table on the local server:

```
CREATE FOREIGN TABLE authors (
    id int,
    name VARCHAR(100) NOT NULL
)
SERVER server1;
```

Check the result:

```
SELECT * FROM authors;
```

The output:

```
 id |       name
----+-------------------
  1 | Virginia Woolf
  2 | Harper Lee
  3 | F. Scott Fitzgerald
  4 | J.R.R. Tolkien
  5 | George Orwell
```

Alternatively, import the entire schema:

```
CREATE SCHEMA schema1;
IMPORT FOREIGN SCHEMA public from SERVER server1 into schema1;
```

Display the existing foreign tables to check the result:

```
SELECT * FROM information_schema.foreign_tables;
```

The result:

```
 foreign_table_catalog | foreign_table_schema | foreign_table_name | foreign_server_catalog | foreign_server_name
-----------------------+----------------------+--------------------+------------------------+---------------------
 postgres              | public               | authors            | postgres               | server1
 postgres              | schema1              | authors            | postgres               | server1
 postgres              | schema1              | books              | postgres               | server1
```

The **books** and **authors** tables have been imported into **schema1**.

Access data files in the server file system

For example, we have the following books.csv file:

```
1,Mrs. Dalloway,1,1925,novel
2,To the Lighthouse,1,1927,novel
3,To Kill a Mockingbird,2,1960,novel
4,The Lord of the Rings,4,1955,fantasy
5,1984,5,1949,sci-fi
```

We need to use SQL to access the data from this file. To accomplish this task, perform the following steps:

1. Execute the **CREATE EXTENSION** command to use the **file_fdw** data wrapper:

```
CREATE EXTENSION file_fdw;
```

2. Define a foreign server:

```
CREATE SERVER file_server FOREIGN DATA WRAPPER file_fdw;
```

3. Create a foreign table. Its structure should match the file structure. Pass the file path and file format in the **OPTIONS** clause:

```
CREATE FOREIGN TABLE foreign_books (
    id int,
    title VARCHAR(255),
    author_id INT,
    public_year INT,
    genre VARCHAR(50)
) SERVER file_server
OPTIONS ( filename '/var/lib/pgsql/books.csv', format 'csv' );
```

Now you can use SQL commands to access data, for example:

```
SELECT * FROM foreign_books;
```

The result:

```
 id |         title          | author_id | public_year |  genre
----+------------------------+-----------+-------------+---------
  1 | Mrs. Dalloway          |         1 |        1925 | novel
  2 | To the Lighthouse      |         1 |        1927 | novel
  3 | To Kill a Mockingbird  |         2 |        1960 | novel
  4 | The Lord of the Rings  |         4 |        1955 | fantasy
  5 | 1984                   |         5 |        1949 | sci-fi
```

TOAST

Eugenia Kuzina

Contents

To Table of Contents

Overview

Out-of-line, on-disk TOAST storage

Out-of-Line, in-memory TOAST storage

Overview

PostgreSQL uses a fixed page size (the default value is 8 KB) and does not allow tuples to span multiple pages. Therefore, it is not possible to store large field values directly. PostgreSQL compresses large field values and/or splits them into multiple physical rows. The rows are saved to special storage. This method is known as **TOAST** (The Oversized-Attribute Storage Technique). PostgreSQL also uses the TOAST infrastructure to improve handling of large data values in memory.

Only certain data types support TOAST (for example, text, json, jsonb, and bytea). To support TOAST, a data type must have a variable-length (varlena) representation, in which, the first four-byte word contains the total value length in bytes (including itself). TOAST does not constrain the rest of the data type's representation. TOAST values work by manipulating and interpreting the first word. Therefore, the C-level functions, which support TOASTed data types, must carefully handle potentially TOASTed input values. An input might not consist of a four-byte length word and contents until it is converted from the TOAST representation. To do this, invoke the `PG_DETOAST_DATUM` macros before processing an input value. In some cases more efficient techniques are possible, see TOAST Considerations for detail.

TOAST uses two bits of the varlena length word for additional settings. It limits the logical size of any TOAST value to 1 GB. When both bits are zero, the value is an ordinary non-TOASTed value, and the remaining bits of the length word give the total datum size in bytes. When the highest-order or lowest-order bit (depending on the architecture) is set, the value has only a single-byte header instead of the four-byte header, and the remaining bits of that byte give the total datum size in bytes. This approach supports space-efficient storage of values shorter than 127 bytes, while still allowing the value of this data type to be extended to 1 GB if needed. As a special case, if the remaining bits of a single-byte header are all zero, the value is a TOAST pointer to out-of-line data, with several possible alternatives described below. The code stored in the second byte determines the type and size of the TOAST pointer. If the highest-order or lowest-order bit is clear but the adjacent bit is set, the content of the datum is compressed and must be decompressed before use. In this case, the remaining bits of the four-byte length word give the total size of the compressed datum, not the original data. The compression is also possible for out-of-line data but the varlena header does not include information about that. The content of the TOAST pointer contains this information instead.

You can select the compression technique used for either in-line or out-of-line compressed data for each column. To do this, set the `COMPRESSION` column option in the `CREATE TABLE` or `ALTER TABLE` command. The default_toast_compression parameter determines the default `COMPRESSION` value if the `COMPRESSION` option is not set.

PostgreSQL provides multiple types of TOAST pointers:

- The oldest and most common type is a pointer to **out-of-line on-disk data** stored in a separate TOAST table, associated with the table containing the TOAST pointer. PostgreSQL creates this on-disk pointer when a tuple is too large to be stored as is.
- A TOAST pointer can refer to **out-of-line data in memory**. This data is necessarily short-lived, and will never appear on the disk. This technique is very useful to avoid copying and redundant processing of large data values.

Out-of-line, on-disk TOAST storage

If any columns of a table are stored in the TOAST format, the table has an associated TOAST table. The TOAST table `OID` is saved in the `pg_class.reltoastrelid` value of the main table. All TOAST values are kept in this TOAST table.

PostgreSQL divides out-of-line values (after compression if used) into chunks of at most `TOAST_MAX_CHUNK_SIZE` bytes. The default value is about 2000 bytes, so that four chunk rows fit on a page. PostgreSQL stores each chunk as a separate row in the TOAST table. Every TOAST table has the following columns:

- `chunk_id` — an OID identifying the particular TOASTed value;
- `chunk_seq` — a sequence number for the chunk within its value;
- `chunk_data` — the actual data of the chunk.

A unique index on `chunk_id` and `chunk_seq` provides fast retrieval of the values. A TOAST pointer contains the OID of the related TOAST table and the OID of the specific value (its `chunk_id`). The pointer also stores the logical size of the data (original uncompressed data length), physical stored size (different if compression is applied), and the compression method, if specified. The total size of an on-disk TOAST pointer is 18 bytes regardless of the actual size of the represented value.

The TOAST management code is triggered only when a row value is wider than `TOAST_TUPLE_THRESHOLD` bytes (normally 2 KB). The TOAST code compress and/or move field values out-of-line until the row value is shorter than `TOAST_TUPLE_TARGET` bytes (also normally 2 KB, adjustable), or it will be impossible to reduce the size. If out-of-line values are not changed during an `UPDATE` operation, there are no costs associated with TOAST.

The TOAST management code recognizes four different strategies for storing TOAST-able columns on the disk:

- `PLAIN` prevents compression or out-of-line storage. It also disables the use of single-byte headers for varlena types. This is the only possible strategy for columns of data types that do not support TOAST.
- `EXTENDED` allows both compression and out-of-line storage. This is the default value for most data types supported TOAST. PostgreSQL uses compression first, then out-of-line storage if the row is still too big.
- `EXTERNAL` allows out-of-line storage but not compression. `EXTERNAL` makes substring operations on text and bytea columns faster because these operations are optimized to fetch only the required parts of the out-of-line value when it is not compressed.
- `MAIN` allows compression but not out-of-line storage. Actually, out-of-line storage is possible for such columns, but only when there is no other way to make the row small enough to fit on a page.

Each TOAST-able data type specifies a default strategy for columns of that data type, but you can change the strategy with the ALTER TABLE ... SET STORAGE command.

You can also use the `ALTER TABLE … SET (toast_tuple_target = N)` to adjust `TOAST_TUPLE_TARGET` for each table.

This scheme has a number of advantages compared to a more straightforward approach such as allowing row values to span pages. Assuming that queries are usually qualified by comparisons against relatively small key values, most of the work will be done using the main row entry. The big values of TOASTed attributes is only pulled out (if selected) at the time the result is sent to the client. The main table is much smaller and more of its rows fit in the shared buffer cache than would be the case without any out-of-line storage. Sort sets shrink also, and PostgreSQL processes sorting in memory more often.

Out-of-Line, in-memory TOAST storage

TOAST pointers can also point to data located in the memory of the current server process. There are currently two types of pointers:

- **Indirect TOAST pointers** refers to a non-indirect varlena value stored in memory. It is currently used during logical decoding to avoid creating physical tuples larger than 1 GB. Note, that the referenced data is available as long as the pointer exists, and there is no infrastructure to save this data.
- **Expanded TOAST pointers** are useful for complex data types whose on-disk representation is not suited for computational purposes. As an example, the standard varlena representation of a PostgreSQL array includes dimensionality information, a NULLs bitmap if there are any NULL elements, then the values of all the elements in order. When the element type has a variable length, the only way to find an element is to scan through all the preceding elements. This representation is appropriate for on-disk storage. For computations, it is better to have an "expanded" or "deconstructed" representation in which all the element starting locations is identified. The TOAST pointer mechanism supports this by allowing a pass-by-reference Datum to point to a standard varlena value (the on-disk representation) or a TOAST pointer that points to an expanded representation in memory. The details of this expanded representation depend on the data type, though it must have a standard header and meet the other API requirements given in Declarations for access to "expanded" value representations.

TOAST pointers to expanded values can be read-write or read-only pointers. A function that receives a read-write pointer is allowed to modify the referenced value. A function, which gets a read-only pointer, must first create a copy to modify the value. This distinction and some associated conventions help to avoid unnecessary copying of extended values during query execution.

For all types of in-memory TOAST pointer, the TOAST management code ensures that no such pointer datum can accidentally be stored on disk. In-memory TOAST pointers are expanded to varlena values before storage — and can then be converted to TOAST pointers on disk if the containing tuple would otherwise be too big.

Tablespaces

Eugenia Kuzina

To Table of Contents

Tablespaces define the physical location of data in a file system.

Tablespaces can be created on different disk partitions or on different disks. For example, a frequently used index can be placed on a fast and highly available SSD drive, while a table with archived data that is rarely used or not performance critical can be stored on a slower disk system. Also, if there is not enough space on the partition on which the cluster was initialized, a tablespace can be created on another partition.

IMPORTANT

Although they may be located outside the main PostgreSQL data directory, tablespaces are part of a cluster and cannot be treated as an autonomous collection of data files. They depend on metadata contained in the main directory and cannot be attached to another database cluster or backed up individually. Also, if you lose a tablespace as a result of a file deletion or disk failure, the cluster may be unavailable or unable to start. Placing a tablespace on a temporary file system, such as a RAM disk, compromises the reliability of the entire cluster.

To define a tablespace, use the CREATE TABLESPACE command:

```
CREATE TABLESPACE tablespace1 LOCATION '/home/user1/tablespace1_data';
```

Where `/home/user1/tablespace1_data` – the directory where the object files belonging to this tablespace are placed. It must be created, owned by the operating system user under which the ADPG is running, and be empty.

For a particular tablespace, you can also override the planner's estimated cost of reading pages from tables, and the executor prefetching behavior set by configuration options for a particular tablespace. It is useful if a tablespace is located on a disk which is faster or slower than others. The available parameters are seq_page_cost, random_page_cost, effective_io_concurrency, and maintenance_io_concurrency. Use the `WITH` clause to override these parameters, for example:

```
CREATE TABLESPACE tablespace2 LOCATION '/home/user1/tablespace2_data' WITH (seq_page_cost=0.8, random_page_cost=6.0);
```

ADPG works with data on disks only through file systems, the most popular are EXT3, EXT4, and XFS. Raw devices are not supported. It is good practice to create one tablespace per logical file system since you cannot control the location of individual files within a logical file system.

A database superuser must create a tablespace. After creation, you can allow ordinary database users to utilize the new tablespace. To do that, call the GRANT command to assign these users the `CREATE` privilege on the new tablespace.

Tables, indexes, and entire databases can be placed in a particular tablespace. To do this, a user with `CREATE` privilege for a given tablespace should pass the name of the tablespace as a parameter to an appropriate command (`CREATE` or `ALTER`).

For example, the following code creates the `t1` table in `tablespace1`:

```
CREATE TABLE t1 (c1 integer) TABLESPACE tablespace1;
```

The commands below move table `t1` and database `database1` to tablespaces `tablespace2` and `tablespace3` respectively:

```
ALTER TABLE t1 SET TABLESPACE tablespace2;

ALTER DATABASE database1 SET TABLESPACE tablespace3;
```

You can use the default_tablespace parameter to set the tablespace to be used by default in `CREATE TABLE` and `CREATE INDEX`:

```
SET default_tablespace = tablespace2;
```

Alternatively, you can add this parameter to a field of the ***ADPG configurations*** section on the ***Clusters → ADPG cluster → Services → ADPG → Primary configuration*** page in ADCM UI (see Configuration parameters).

PostgreSQL also provides the temp_tablespaces parameter, which determines tablespaces for temporary tables, indexes, and temporary files that are used for sorting large data sets. It can be a list of tablespace names, rather than only one so that PostgreSQL spreads the load associated with temporary objects over multiple tablespaces. Each time a temporary object is created, PostgreSQL will randomly select a name from the specified list.

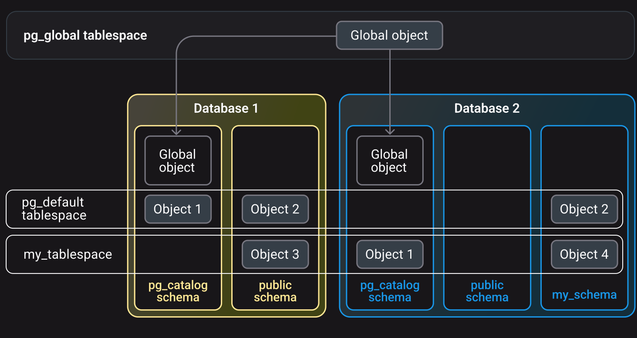
During a cluster installation, two tablespaces are created. They are described in the table below.

Default tablespaces

Name	Description
pg_default	Used by default for databases and templates (`template0`, `template1`). The `pg_default` tablespace is located in the */pg_data1/adpg16/base* directory. Within this directory, objects are placed in database subdirectories
pg_global	This tablespace contains the system objects of the entire cluster. The `pg_global` tablespace is located in the */pg_data1/adpg16/global*. At the logical level of data organization, the `pg_catalog` schema contains objects from the `pg_global` tablespace

The `Data directory` configuration parameter determines the first part of the directory path (*/pg_data1*).

Objects in the same database can be placed in multiple tablespaces. One tablespace can contain objects from multiple databases.



Physical layer of ADPG cluster

To get a list of cluster tablespaces, use the psql `\db` and `\db+` meta-commands. You can also query the `pg_tablespace` system directory:

```
SELECT * FROM pg_tablespace;
```

Since a tablespace can be used within any database, it cannot be dropped until all objects in all databases using the tablespace are removed. You need to delete all objects first. To drop an empty tablespace, call DROP TABLESPACE command.

```
DROP TABLESPACE tablespace2;
```

PostgreSQL uses symbolic links to simplify the implementation of tablespaces. This means that tablespaces can be used only on systems that support symbolic links. The */pg_data1/adpg16/pg_tblspc* directory contains symbolic links that point to non-built-in tablespaces defined in the cluster.

Views and materialized views

Contents

To hide all contents

- Views
  - Create a view
  - Alternative view
  - Rename view recursive
  - Drop view if exists
- Materialized views
  - Create a materialized view
  - Change materialized view properties
  - Refresh a materialized view
- Views vs materialized views
- Get information about views and materialized views
- System views

Views

A **view** is a database object that is defined by a query. In PostgreSQL, a view can be defined on one or more tables or other views. It does not hold data itself. Instead, it dynamically retrieves data when it is queried. A view can be used to encapsulate complex queries, simplify data access, and present data in a specific format without storing it separately.

Views in PostgreSQL can be updated under certain conditions, such as when they are based on a single table and do not contain aggregate functions or `GROUP BY` clauses.

```
-- Create a view
CREATE VIEW view_name AS query;

-- Create or replace a view
CREATE OR REPLACE VIEW view_name AS query;

-- Drop a view
DROP VIEW view_name;
```

Create a view

Data for this example:

```
CREATE TABLE users (
    id serial PRIMARY KEY,
    username varchar(50) UNIQUE NOT NULL,
    password varchar(50) NOT NULL,
    email varchar(255) UNIQUE NOT NULL,
    created_on timestamp NOT NULL,
    last_login timestamp
);

INSERT INTO users (username, password, email, created_on, last_login)
VALUES ('john', 'password123', 'john@example.com', '2022-01-01 12:00:00', '2022-01-02 12:00:00'),
       ('jane', 'password456', 'jane@example.com', '2022-01-03 12:00:00', '2022-01-04 12:00:00'),
       ('bob', 'password789', 'bob@example.com', '2022-01-05 12:00:00', '2022-01-06 12:00:00');
```

Syntax to create a view:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- `CREATE VIEW view_name AS`: This part of the statement specifies the name of the view you want to create. Replace `view_name` with a meaningful name for your view.
- `SELECT column1, column2, ...`: This is the query that defines the data that the view will contain. You can select specific columns from one or more tables, or you can use `*` to select all columns.
- `FROM table_name`: This specifies the table or tables that the data will be selected from.
- `WHERE condition`: This is an optional clause that allows you to filter the data that is included in the view. You can use any valid SQL condition here.

```
CREATE VIEW user_info AS
SELECT id, username, email, created_on, last_login
FROM users;
```

- `CREATE VIEW user_info AS`: This creates a new view named `user_info`.
- `SELECT id, username, email, created_on, last_login`: This selects the `id`, `username`, `email`, `created_on`, and `last_login` columns from the `users` table.
- `FROM users`: This specifies that the data will be selected from the `users` table.

Create the view and query it:

```
CREATE VIEW user_info AS
SELECT id, username, email, created_on, last_login
FROM users;

SELECT * FROM user_info;
```

Drop the view:

```
DROP VIEW user_info;
```

The result:

```
SELECT * FROM user_info;
 id | username |       email       |     created_on      |     last_login
----+----------+-------------------+---------------------+---------------------
  1 | john     | john@example.com  | 2022-01-01 12:00:00 | 2022-01-02 12:00:00
  2 | jane     | jane@example.com  | 2022-01-03 12:00:00 | 2022-01-04 12:00:00
  3 | bob      | bob@example.com   | 2022-01-05 12:00:00 | 2022-01-06 12:00:00
(3 rows)
```

Update the table data:

```
UPDATE users
SET email = 'john.doe@example.com'
WHERE id = 1;
```

The result:

```
SELECT * FROM user_info;
 id | username |       email       |     created_on      |     last_login
----+----------+-------------------+---------------------+---------------------
  1 | john     | john.doe@example.com | 2022-01-01 12:00:00 | 2022-01-02 12:00:00
  2 | jane     | jane@example.com  | 2022-01-03 12:00:00 | 2022-01-04 12:00:00
  3 | bob      | bob@example.com   | 2022-01-05 12:00:00 | 2022-01-06 12:00:00
(3 rows)
```

Updatable view

Make the view updatable: `WITH [CASCADED | LOCAL] CHECK OPTION` and `WITH CHECK OPTION` are used to create updatable views. They ensure that any data modified through the view meets the conditions specified in the view's `WHERE` clause.

- `CASCADED`: This is the default option. It checks the conditions of the current view and all underlying views.
- `LOCAL`: This option checks the conditions of the current view only.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition
WITH [CASCADED | LOCAL] CHECK OPTION;
```

Create a view with `WITH CHECK OPTION`:

```
CREATE VIEW user_info AS
SELECT id, username, email, created_on, last_login
FROM users
WHERE id > 0
WITH CHECK OPTION;
```

- `CREATE VIEW user_info AS`: This creates a new view named `user_info`.
- `SELECT id, username, email, created_on, last_login`: This selects the `id`, `username`, `email`, `created_on`, and `last_login` columns from the `users` table.
- `FROM users`: This specifies that the data will be selected from the `users` table.
- `WHERE id > 0`: This filters the data to only include rows where the `id` column is greater than 0.
- `WITH CHECK OPTION`: This ensures that any data modified through the view meets the conditions specified in the `WHERE` clause.

Create the view and query it:

```
CREATE VIEW user_info AS
SELECT id, username, email, created_on, last_login
FROM users
WHERE id > 0
WITH CHECK OPTION;

SELECT * FROM user_info;
```

Drop the view:

```
DROP VIEW user_info;
```

The result:

```
SELECT * FROM user_info;
 id | username |       email       |     created_on      |     last_login
----+----------+-------------------+---------------------+---------------------
  1 | john     | john@example.com  | 2022-01-01 12:00:00 | 2022-01-02 12:00:00
  2 | jane     | jane@example.com  | 2022-01-03 12:00:00 | 2022-01-04 12:00:00
  3 | bob      | bob@example.com   | 2022-01-05 12:00:00 | 2022-01-06 12:00:00
(3 rows)
```

Now we can't insert a row with `id = 0`:

```
INSERT INTO user_info (id, username, email, created_on, last_login)
VALUES (0, 'test', 'test@example.com', '2022-01-07 12:00:00', '2022-01-08 12:00:00');
ERROR:  new row violates check option for view "user_info"
DETAIL:  Failing row contains (0, test, test@example.com, 2022-01-07 12:00:00, 2022-01-08 12:00:00).
```

Alternative view

`CREATE OR REPLACE VIEW` is used to create a new view or replace an existing view with the same name. This is useful when you want to modify the definition of a view without having to drop it first.

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Create a view:

```
CREATE VIEW user_info AS
SELECT id, username, email, created_on, last_login
FROM users
WHERE id > 0;
```

Create or replace the view:

```
CREATE OR REPLACE VIEW user_info AS
SELECT id, username, email, created_on, last_login
FROM users
WHERE id > 0
WITH CHECK OPTION;
```

Rename view recursive

Use the `WITH` CTE to rename a view recursively:

```
WITH RECURSIVE view_name(column1, column2, ...) AS (
    SELECT column1, column2, ...
    FROM table_name
    WHERE condition
    UNION ALL
    SELECT column1, column2, ...
    FROM view_name
    WHERE condition
)
SELECT * FROM view_name;
```

Drop view if exists

Use the `DROP VIEW IF EXISTS` statement to drop a view if it exists:

```
DROP VIEW IF EXISTS view_name;
```

- `DROP VIEW`: This is the command that tells PostgreSQL to drop a view.
- `IF EXISTS`: This is an optional clause that prevents an error from occurring if the view does not exist.
- `view_name`: This is the name of the view you want to drop.

Materialized views

Materialized views are a powerful feature in PostgreSQL that allow you to store the results of a query as a physical table. This can be useful for improving the performance of complex queries that are frequently executed. Materialized views are similar to regular views, but they store the data in a physical table rather than just storing the query definition.

```
-- Create a materialized view
CREATE MATERIALIZED VIEW view_name AS query;

-- Refresh a materialized view
REFRESH MATERIALIZED VIEW view_name;

-- Drop a materialized view
DROP MATERIALIZED VIEW view_name;
```

Create a materialized view

Data for this example:

```
CREATE TABLE products (
    id serial PRIMARY KEY,
    name varchar(50) NOT NULL,
    price numeric(10, 2) NOT NULL,
    quantity integer NOT NULL
);

INSERT INTO products (name, price, quantity)
VALUES ('Product 1', 10.00, 100),
       ('Product 2', 20.00, 50),
       ('Product 3', 30.00, 25);
```

Syntax to create a materialized view:

```
CREATE MATERIALIZED VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition
WITH [NO] DATA;
```

- `CREATE MATERIALIZED VIEW view_name AS`: This part of the statement specifies the name of the materialized view you want to create. Replace `view_name` with a meaningful name for your view.
- `SELECT column1, column2, ...`: This is the query that defines the data that the materialized view will contain. You can select specific columns from one or more tables, or you can use `*` to select all columns.
- `FROM table_name`: This specifies the table or tables that the data will be selected from.
- `WHERE condition`: This is an optional clause that allows you to filter the data that is included in the materialized view. You can use any valid SQL condition here.
- `WITH [NO] DATA`: This is an optional clause that specifies whether the materialized view should be populated with data when it is created. If you use `WITH DATA`, the materialized view will be populated with data. If you use `WITH NO DATA`, the materialized view will be created but not populated with data.

Create a materialized view and query it:

```
CREATE MATERIALIZED VIEW product_info AS
SELECT id, name, price, quantity
FROM products
WHERE quantity > 0
WITH DATA;

SELECT * FROM product_info;
```

Drop the materialized view:

```
DROP MATERIALIZED VIEW product_info;
```

The result:

```
SELECT * FROM product_info;
 id |   name    | price | quantity
----+-----------+-------+----------
  1 | Product 1 | 10.00 |      100
  2 | Product 2 | 20.00 |       50
  3 | Product 3 | 30.00 |       25
(3 rows)
```

Materialized views store the data in a physical table, so they can be indexed to improve query performance:

```
CREATE INDEX ON product_info (name);
```

The result:

```
SELECT * FROM product_info;
 id |   name    | price | quantity
----+-----------+-------+----------
  1 | Product 1 | 10.00 |      100
  2 | Product 2 | 20.00 |       50
  3 | Product 3 | 30.00 |       25
(3 rows)
```

Materialized views do not automatically update when the underlying data changes. You can use the `REFRESH MATERIALIZED VIEW` command to update the data in a materialized view:

```
REFRESH MATERIALIZED VIEW product_info;
```

The result:

```
SELECT * FROM product_info;
 id |   name    | price | quantity
----+-----------+-------+----------
  1 | Product 1 | 10.00 |      100
  2 | Product 2 | 20.00 |       50
  3 | Product 3 | 30.00 |       25
(3 rows)
```

Change materialized view properties

Use the `ALTER MATERIALIZED VIEW` command to change the properties of a materialized view:

```
-- Rename a materialized view
ALTER MATERIALIZED VIEW view_name RENAME TO new_view_name;

-- Change the owner of a materialized view
ALTER MATERIALIZED VIEW view_name OWNER TO new_owner;

-- Change the schema of a materialized view
ALTER MATERIALIZED VIEW view_name SET SCHEMA new_schema;

-- Change the tablespace of a materialized view
ALTER MATERIALIZED VIEW view_name SET TABLESPACE new_tablespace;

-- Change the storage parameters of a materialized view
ALTER MATERIALIZED VIEW view_name SET (storage_parameter = value);

-- Change the refresh mode of a materialized view
ALTER MATERIALIZED VIEW view_name SET (refresh_mode = 'mode');
```

Create a materialized view and query it:

```
CREATE MATERIALIZED VIEW product_info AS
SELECT id, name, price, quantity
FROM products
WHERE quantity > 0
WITH DATA;

SELECT * FROM product_info;
```

Rename the materialized view:

```
ALTER MATERIALIZED VIEW product_info RENAME TO product_info_new;
```

The result:

```
SELECT * FROM product_info_new;
 id |   name    | price | quantity
----+-----------+-------+----------
  1 | Product 1 | 10.00 |      100
  2 | Product 2 | 20.00 |       50
  3 | Product 3 | 30.00 |       25
(3 rows)
```

Change the owner of the materialized view:

```
ALTER MATERIALIZED VIEW product_info_new OWNER TO new_owner;
```

Change the schema of the materialized view:

```
ALTER MATERIALIZED VIEW product_info_new SET SCHEMA new_schema;
```

Change the tablespace of the materialized view:

```
ALTER MATERIALIZED VIEW product_info_new SET TABLESPACE new_tablespace;
```

Change the storage parameters of the materialized view:

```
ALTER MATERIALIZED VIEW product_info_new SET (fillfactor = 70);
```

Change the refresh mode of the materialized view:

```
ALTER MATERIALIZED VIEW product_info_new SET (refresh_mode = 'CONCURRENTLY');
```

Refresh a materialized view

To update the data in a materialized view, you can use the `REFRESH MATERIALIZED VIEW` command. This command will recalculate the data in the materialized view and update the stored data.

```
REFRESH MATERIALIZED VIEW view_name;
```

- `REFRESH MATERIALIZED VIEW`: This is the command that tells PostgreSQL to refresh a materialized view.
- `view_name`: This is the name of the materialized view you want to refresh.

Create a materialized view and query it:

```
CREATE MATERIALIZED VIEW product_info AS
SELECT id, name, price, quantity
FROM products
WHERE quantity > 0
WITH DATA;

SELECT * FROM product_info;
```

Update the data in the underlying table:

```
UPDATE products
SET quantity = quantity - 10
WHERE id = 1;
```

Refresh the materialized view:

```
REFRESH MATERIALIZED VIEW product_info;
```

The result:

```
SELECT * FROM product_info;
 id |   name    | price | quantity
----+-----------+-------+----------
  1 | Product 1 | 10.00 |       90
  2 | Product 2 | 20.00 |       50
  3 | Product 3 | 30.00 |       25
(3 rows)
```

Views vs materialized views

The following table compares views and materialized views:

Feature	View	Materialized view
Definition	A virtual table that is defined by a query	A physical table that is created by a query
Data storage	Does not store data	Stores data
Performance	Can be slow for complex queries	Can be fast for complex queries
Updates	Automatically updated when the underlying data changes	Must be manually refreshed to update the data
Indexing	Cannot be indexed	Can be indexed
Use cases	Simplifying complex queries, providing a consistent interface to data	Improving the performance of complex queries, caching data

Get information about views and materialized views

You can query `pg_views` and `pg_matviews` to get information about views and materialized views.

pg_views

```
SELECT * FROM pg_views;
```

The result:

```
-[ RECORD 1 ]-----------------------
```

Sequences

AD.PG

Contents

To Table of Contents

Create a sequence
Change a sequence
Delete a sequence
Work with sequences
- nextval
- setval
- currval
- lastval

List sequences in a database

The sequence is the schema object that generates a sequence of unique integers in ascending or descending order. The sequence is not associated with any table, but it can be used to associate values in the primary key of a table. The PostgreSQL [serial](#) type uses sequences.

Create a sequence

To create a sequence, use the `CREATE SEQUENCE` command that defines a new sequence generator.

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

Parameters of the CREATE SEQUENCE command

TEMPORARY | TEMP	If specified, the sequence object is created only for the current session. A temporary sequence is dropped at the end of the session. Existing permanent sequences with the same name are not visible in the current session while the temporary sequence exists. It is only possible to reference a permanent sequence with a qualified schema name.
IF NOT EXISTS	PostgreSQL creates a sequence if it does not exist. If the same sequence name already exists, PostgreSQL issues a notice instead. Note that the existing sequence may have different parameters than the new one would have.
name	The name of the new sequence. Can be schema-qualified: *[schema_name].sequence_name*.
data_type	The optional clause `AS data_type` specifies the data type of the sequence. Valid types are: `smallint`, `integer`, and `bigint`. The default value is `bigint`. The data type determines the default minimum and maximum values of the sequence.
increment	The optional clause `INCREMENT [ BY ] increment` specifies the value that is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.
minvalue / NO MINVALUE	The optional clause `MINVALUE minvalue` determines the minimum value that a sequence can generate. If this option is not specified or `NO MINVALUE` is specified, defaults are used. The default value for an ascending sequence is 1. The default value for a descending sequence is the minimum value of the data type for ascending and descending sequences.
maxvalue / NO MAXVALUE	The optional clause `MAXVALUE maxvalue` determines the maximum value for the sequence. If this option is not specified or `NO MAXVALUE` is specified, defaults are used. The default value for a descending sequence is -1. The default value for an ascending sequence is the maximum value of the data type for ascending and descending sequences.
start	The optional clause `START WITH start [AS]` specifies the start value at which to start the sequence. The default for an ascending sequence is `MINVALUE`, and for a descending sequence is `MAXVALUE`. Setting `START WITH` to a value that is outside the sequence boundaries will cause an error. `NO CYCLE` is used by default.
cache	The optional clause `CACHE cache` specifies how many sequence values should be stored ahead of time. If this clause is not specified or `NO CACHE` is specified, the default value is 1. The minimum value is 1 (only one value can be generated at a time, i.e. no cache). The maximum value is 2^63-1.
CYCLE | NO CYCLE	The `CYCLE` option allows the sequence to wrap around when `maxvalue` or `minvalue` is reached by an ascending or descending sequence respectively. If the limit is reached, the next generated number will be the `minvalue` or `maxvalue`, respectively. If `NO CYCLE` is specified, any calls to `nextval` after the sequence has reached its limit will return an error. `NO CYCLE` is used by default.
OWNED BY *table_name.column_name* / NONE	The `OWNED BY` option causes the sequence to be associated with a specific table column. If this column or its whole table is dropped, the sequence will be automatically dropped as well. The table must have the same owner and be in the same schema as the sequence. `OWNED BY NONE` specifies that there is no such association and is used by default.

The following example creates a sequence starting with `2`:

```
CREATE SEQUENCE accepsequence START 2;
```

`CREATE SEQUENCE` creates and initializes a new special single-row table with the specified name. The user who executes `CREATE SEQUENCE` becomes the owner of the sequence.

The optional clauses `MINVALUE` and `MAXVALUE` determine the range of values that the sequence will generate. If these clauses are omitted or `NO MINVALUE` and `NO MAXVALUE` are specified, the default values are used. The default value for an ascending sequence is 1 and for a descending sequence is -1. The optional clause `START WITH` determines the first value that the sequence will generate. The default for an ascending sequence is `MINVALUE`, and for a descending sequence is `MAXVALUE`.

The optional clause `INCREMENT BY` specifies the value that is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

The optional clause `CACHE` specifies how many sequence values are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache), and this is also the default.

The optional clause `CYCLE` allows the sequence to wrap around when the `maxvalue` or `minvalue` has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`, respectively.

If `NO CYCLE` is specified, any calls to `nextval` after the sequence has reached its limit will return an error. `NO CYCLE` is used by default.

The `OWNED BY` option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The referenced table must have the same owner and be in the same schema as the sequence. `OWNED BY NONE`, the default, specifies that there is no such association.

```
SELECT * FROM accepsequence;
```

The result:

```
 last_value | log_cnt | is_called
------------+---------+-----------
          2 |       0 | f
```

The `last_value` is the value that was allocated by any process. This value might be obscured by the time it is visible if other sessions are actively doing `nextval` calls.

The `is_called` flag can be tricky to XXX, read! Nothing of a value from a sequence for performance reasons. If perhaps a file is broken, a sequence, if needs steps, and so check values are not changed. `log_cnt` shows how many times values have been allocated, and `is_called` shows if the sequence has been used.

The `is_called` flag is `f` (false) when the sequence is created or when `setval` is called with `is_called` = `f`. `false` `nextval` will return the `last_value` value.

If a sequence is created, use the sequence manipulation functions to operate with the sequence. See [Work with sequences](#).

> If the `CACHE` value greater than 1 for a large sequence, it is used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object and increase the sequence's last value accordingly. Then, the next `CACHE` (-1) `nextval` within this session return values from the sequence cache without accessing the sequence object. Any values that are cached but not used in the current session are lost when the session ends, resulting in "holes" in the sequence.
>
> Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values might be generated out of sequence when all the sessions are considered. For example, with a cache setting of `10`, the first session might return `1`–`10`, and a second session might return `11`–`20` next. If the first session then does `nextval` returns `21`, since `nextval` of the first session generates `2`. Therefore, with a cache setting of `1`, you can be sure that `nextval` values are generated sequentially; with a cache setting greater than one you can only assume that the `nextval` values are all distinct. However, they might be generated in a somewhat scrambled order. This behavior can be altered by setting the sequence's `CACHE` parameter to `1` or by using the `NO CACHE` option.

`CREATE SEQUENCE` conforms to the SQL standard, with the following exceptions:

- The clauses `AS data_type`, `OWNED BY`, and `NO MINVALUE` `NO MAXVALUE` are extensions.
- The `OWNED BY` clause is a PostgreSQL extension.

Change a sequence

You can use `ALTER SEQUENCE` to change the definition of a sequence generator. Any parameters that are not set in the `ALTER SEQUENCE` command retain their previous settings. You must own the sequence to use `ALTER SEQUENCE`. To change a sequence's schema you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the sequence's schema.

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ AS new_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

Parameters of the ALTER SEQUENCE command

name	The name of a sequence to be altered. Can be schema-qualified: *[schema_name].sequence_name*.
IF EXISTS	Attempt the update if the sequence exists. Don't throw an error if it's not. Without this option, an error is thrown if the sequence does not exist in the database. A notification is displayed in this case.
data_type	The optional clause `AS data_type` changes the data type of the sequence. Valid types are: `smallint`, `integer`, and `bigint`. Changing the data type automatically changes the minimum and maximum values of the sequence (if previously selected) to the maximum supported by the new data type. If the minimum or maximum values were created using `NO MINVALUE` or `NO MAXVALUE` explicitly or implicitly, Otherwise, the minimum and maximum values of the new data type are selected. You cannot alter the data type of a sequence owned by the column of a table. Sequences owned by columns are always `bigint`.
increment	The optional clause `INCREMENT [ BY ] increment` changes the value that is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. If not specified, the old `INCREMENT` value is retained.
minvalue / NO MINVALUE	The optional clause `MINVALUE minvalue` changes the minimum value that a sequence can generate. If `NO MINVALUE` is specified, the old minimum value of the data type for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value is retained.
maxvalue / NO MAXVALUE	The optional clause `MAXVALUE maxvalue` changes the maximum value for the sequence. If `NO MAXVALUE` is specified, the old maximum value of the data type for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value is retained.
start	The optional clause `START WITH start [AS]` changes the value at which to start the sequence. This has no effect on the current sequence value; it sets the value that the `ALTER SEQUENCE RESTART` command will use.
restart	The optional clause `RESTART [ WITH *restart*]` changes the current value of the sequence. This is similar to calling the `setval(sequence, restart, false)` function, except that PostgreSQL records the new value in the sequence's system catalog immediately. Specifying `RESTART WITH` retrieves only existing association
cache	The optional clause `CACHE *cache*` sets how many sequence values should be stored ahead of time. If this clause is not specified, the old cache value is retained. `ALTER SEQUENCE RESTART WITH` retrieves only existing association
CYCLE | NO CYCLE	The `CYCLE` option allows the sequence to wrap around when `maxvalue` or `minvalue` is reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the `minvalue` or `maxvalue`. For a descending sequence.
OWNED BY *table_name.column_name* / NONE	The `OWNED BY` option causes the sequence to be associated with a specific table column, such that if that column is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association. Specifying `OWNED BY NONE` removes any existing association
new_owner	The user name of the new sequence owner
new_name	The new name for the sequence
new_schema	The new schema for the sequence

The following statement makes the `accepsequence` sequence descending from 1% to 5 using the step 2 and the cycle option:

```
ALTER SEQUENCE accepsequence
    INCREMENT -2
    MINVALUE 5
    MAXVALUE 1%
    RESTART 10
    CYCLE;
```

`ALTER SEQUENCE` does not immediately affect `nextval` results in backends that have cached sequence values. They will only notice the changes after all cached values are used. Changes to the sequence's `RESTART` value are not affected.

`ALTER SEQUENCE` does not affect the `currval` status for the sequence.

`ALTER SEQUENCE` does not affect the `currval` status for the sequence.

For historical reasons, `ALTER TABLE` can be used with sequences too; but only with variants that are equivalent to the forms shown above.

Delete a sequence

The `DROP SEQUENCE` command removes a sequence. It can only be dropped by its owner or a superuser.

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Parameters of the DROP SEQUENCE command

IF EXISTS	When this option is specified, PostgreSQL does not throw an error if a sequence with the specified name does not exist in the database. A notification is displayed in this case.
name [, ...]	The name of a sequence to be removed. Can be schema-qualified: *[schema_name].sequence_name*. You can specify multiple names separated by commas to remove multiple sequences.
CASCADE	Drop objects that depend on the sequence, and in turn all objects that depend on those objects. See [Dependency tracking](#).
RESTRICT	Refuse to drop the sequence if any objects depend on it. This is the default behavior.

Example:

```
DROP SEQUENCE accepsequence;
```

Work with sequences

The functions described below provide simple and multiuser-safe methods for obtaining successive values from sequence objects.

nextval

The `nextval` function advances the sequence object to its next value and returns that value:

```
nextval( c'sequence' ) → bigint
```

If a future sessions execute `nextval` concurrently, each safely receives a different sequence value. If the sequence object was created with default parameters, successive `nextval` results will return successive values beginning with `1`. Other behaviors can be obtained by using appropriate parameters in the [CREATE SEQUENCE](#) command.

This function requires `SCHEMA` or `UPDATE` privilege on the sequence.

This example function shows how to create a sequence associated with a table column and fill the column using `nextval`. To do this, perform the following steps:

1. Create a table

```
CREATE TABLE book_orders (
    book_id INTEGER,
    item_id TEXT NOT NULL,
    book_name VARCHAR NOT NULL,
    price NUMERIC(5,2)
    PRIMARY KEY(book_id, item_id)
);
```

2. Create a new sequence associated with the `item_id` column of the `book_orders` table

```
CREATE SEQUENCE book_orders_item START 1 OWNED BY book_orders.item_id;
```

3. Add rows to the table using the `nextval()` function to fill the `item_id` column:

```
INSERT INTO
    book_orders(item_id, book_name, price)
VALUES
    ('1', 'nextval()', 'Harrier',13),
    ('2', 'nextval()', 'Book and Peace ', 20),
    ('3', 'nextval()', 'Other', 25),
    ('4', 'nextval()', 'The Tiny Machine',17);
```

4. Check the table data:

```
SELECT * FROM book_orders;
```

The result:

```
 order_id | item_id |   book_name    | price
----------+---------+----------------+-------
        1 |       1 | Harrier        |    13
        2 |       2 | Book and Peace |    20
        3 |       3 | Other          |    25
        4 |       4 | The Tiny Machine |    17
(4 rows)
```

If you drop the `book_orders` table, the `book_order_item` sequence will also be dropped.

setval

The `setval` function sets the current value of the sequence object, and optionally its `is_called` flag. `setval` can be called with two or three parameters:

```
setval( c'sequence', nbigint [, nboolean ] ) → bigint
```

Where:

- `c'sequence'` is a sequence name
- `nbigint` is the `last_value` value
- `nboolean` is the `is_called` value

The form with two parameters sets the `last_value` field to the specified value (`nbigint`), and sets its `is_called` field to `true`, meaning that the next `nextval` will advance the sequence before returning a value. The specified value can be set to one less than the minimum value of the sequence (`ALTER SEQUENCE` can be used for `setval` that). The next form is the same as the first, but the `is_called` field can be set to either `true` or `false`. `true` has the same effect as the first form. `false` sets the sequence object's `last_value` parameter to the specified value. The next `nextval` will return the specified value. The `currval` function will also return that value. The next `nextval` will advance the sequence by one before returning a value.

Command	Value returned by nextval
SELECT setval('myseqname', 10);	10
SELECT setval('myseqname', 10, true);	10
SELECT setval('myseqname', 10, false);	9

The result returned by `setval` is the second argument value.

The `setval` function requires `UPDATE` privilege on the sequence.

currval

The `currval` function returns the value most recently obtained using `nextval`, for the specified sequence in the current session:

```
currval( c'sequence' ) → bigint
```

If `nextval` was never been called for this sequence in the current session, and you try to execute `currval`, an error occurs. Once `currval` returns a value for the current session, it returns the same value until `nextval` is called for the sequence.

This is the equivalent `SELECT` and shows an example:

Example:

```
SELECT currval('nsequence');
```

The result:

```
 currval
---------
      10
(1 row)
```

lastval

The function returns the value most recently obtained by `nextval` in the current session for any sequence.

```
lastval () → bigint
```

This function is identical to `currval`, but instead of taking the sequence name as an argument, it refers to the most recent `nextval` call related to any sequence in the current session. If `nextval` has never been called in the current session, `lastval` will return an error.

`lastval` requires `USAGE` or `SELECT` privilege on the last used sequence.

Example:

```
SELECT * FROM lastval();
```

The result:

```
 lastval
---------
      14
(1 row)
```

List sequences in a database

You can get information about sequences from the [information_schema](#) view, created as:

```
SELECT sequence_name, sequence_name, data_type, minimum_value, maximum_value, increment, cycle_option
FROM information_schema.sequences;
```

The result:

```
 sequence_schema | sequence_name | data_type | minimum_value | maximum_value | increment | cycle_option
-----------------+---------------+-----------+---------------+---------------+-----------+--------------
public          | basesequence  | bigint    | 1             | 9223372036854775807 | 1         | NO
public          | book_orders_order_id_seq | integer | 1 | 2147483647 | 1 | NO
public          | book_orders_item_id_seq | bigint | 1 | 9223372036854775807 | 1 | NO
(3 rows)
```

In addition, you can obtain data from the sequence's `book_orders_order_id_seq` sequence for the `order_id` column of the table `book_orders`:

```
SELECT * FROM book_orders_order_id_seq;
```

The [pg_sequences](#) view also provides access to useful information about sequences in the database. For example, you can get information about sequences owners that `information_schema` does not show:

```
SELECT sequencename, sequenceowner FROM pg_sequences;
```

The result:

```
    sequencename     | sequenceowner
---------------------+---------------
basesequence         | postgres
book_orders_order_id_seq | postgres
book_order_items     | postgres
(3 rows)
```

In addition, you can also obtain data from the [pg_statio](#) and [pg_class](#) system catalogs using the following query:

```
SELECT pg_class.relname, pg_sequences.* FROM pg_sequences INNER JOIN pg_class ON pg_sequences.sequencename = pg_class.oid;
```

The result:

```
-[ RECORD 1 ]-+-----------------
relname       | basesequence
schemaname    | public
sequencename  | basesequence
sequenceowner | postgres
last_value    |
start_value   | 1
increment_by  | 1
max_value     | 9223372036854775807
min_value     | 1
cache_size    | 1
cycle         | f
-[ RECORD 2 ]-+-----------------
relname       | book_orders_order_id_seq
schemaname    | public
sequencename  | book_orders_order_id_seq
sequenceowner | postgres
last_value    |
start_value   | 1
increment_by  | 1
max_value     | 2147483647
min_value     | 1
cache_size    | 1
cycle         | f
-[ RECORD 3 ]-+-----------------
relname       | book_order_items
schemaname    | public
sequencename  | book_order_items
sequenceowner | postgres
last_value    |
start_value   | 1
increment_by  | 1
max_value     | 9223372036854775807
min_value     | 1
cache_size    | 1
cycle         | f
```

It is also possible to use the [psql](#) [\ds+] meta-command to list sequences:

```
\ds
```

The result:

```
                    List of relations
 Schema |        Name        |   Type   |  Owner
--------+--------------------+----------+----------
public  | book_order_items   | sequence | postgres
public  | book_orders_order_id_seq | sequence | postgres
public  | basesequence       | sequence | postgres
(3 rows)
```

Indexes

AD.PG

Contents

Try Table Of Contents

- Overview
- Index types
  - B-Tree
  - Hash
  - GiST
  - SP-GiST
  - GIN
  - BRIN
- Manage indexes
- Rebuild indexes
- Multicolumn indexes
- Unique indexes
- Indexes on expressions
- Partial indexes
- Examine the use of indexes

Overview

Indexes are a common way to increase database performance. Without indexes, the database must perform a full table scan (sequential scan) to find the data. This can be a slow and resource-intensive process, especially for large tables. Indexes can significantly speed up searches because they provide a data structure that points to the location of the information in the table. In this way, a database server can find the data it needs in a fraction of the time it would take without an index.

Indexes are not always a positive factor for performance. When data is inserted, updated, or deleted, the indexes must be updated, which can result in additional time and resources being used. Therefore, it is important to carefully consider which columns should be indexed and which should not to create unnecessary load of the system.

Index types

PostgreSQL provides multiple index types: B-tree, Hash, GIST, SP-GIST, GIN, BRIN, and the `bloom` extension. Each index type uses a different algorithm that is best suited for a certain type of clauses.

B-Tree

B-tree indexes can handle equality and range queries on data that can be sorted in some order. The query planner considers using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: `<`, `<=`, `=`, `>=`, `>`, and `BETWEEN`.

Combined equivalent to combinations of these operators, such as `BETWEEN` and `IN`, can be implemented with a B-tree index search. A B-tree index can also handle the `IS NULL` and `IS NOT NULL` operators.

In addition, the optimizer can also use a B-tree index for queries with the pattern matching operators `LIKE` or `~` if the pattern is a constant and is anchored to the beginning of the string. For example, `col LIKE 'foo%'` or `col ~ '^foo'`, but not `col LIKE '%bar'`. However, if your database does not use the C locale, you need to create an index with a special operator class to support indexing of pattern-matching queries. See Operator Classes and Operator Families.

B-tree indexes can also be used to retrieve data in sorted order. This is not always faster than a simple scanning and sorting, but can sometimes be useful.

Hash

Hash indexes store a 32-bit hash code derived from the value of the indexed column. Hence, such indexes can only handle simple equality comparisons. The query planner considers using a hash index when an indexed column is involved in a comparison using the equality operator `=`.

The following command creates a hash index named `test_hash_idx` on the `name` table:

```
CREATE INDEX name_hash_idx ON name USING HASH (id);
```

GiST

GiST (Generalized Search Tree) indexes are designed to work with complex data types such as geometric objects, text, and arrays. GiST indexes allow you to quickly search spatial, text, and hierarchical data. GiST indexes are not a single kind of index. They are an infrastructure within which many different indexing strategies can be implemented. As a consequence, the operators with which a GiST index can be used vary depending on the indexing strategy. The KNN distance-ordered searches (GiST operator classes and GiST indexes: part 2) are two-dimensional geometric data types, which support indexed queries with the following operators: `<<`, `&<`, `&>`, `>>`, `<<|`, `&<|`, `|&>`, `|>>`, `@>`, `<@`, `~=`, and `&&`. These operators are described in the Geometric Functions and Operators section of the PostgreSQL documentation. Operator Families and Operator Classes.

The GiST operator classes included in the standard distribution are described in the GiST and GIN Operator Classes article. Many other GiST operator classes can be found in the contrib collection and in the PostGIS project. The creation of a GiST index is permitted in the ALTER TABLE command. See Index Definitions.

GiST indexes are also capable of optimizing the "nearest-neighbor" search, for ZIP-GiST operator classes that support distance ordering, the corresponding operator is listed in the **Ordering Operators** column in the Built-in GiST Operator Classes table.

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

The code above finds ten places closest to a given target point. Whether an index can be used in this way depends on the class of operator. In the Built-in GiST Operator Classes table, operators that can be used in this way are listed in the column **Ordering Operators**.

You can also create a GiST index for searching text data:

```
CREATE INDEX ind_gist ON my_table USING gist (to_tsvector('english', ind_column));
```

SP-GiST

SP-GiST indexes (space-partitioned GiST), like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST allows you to implement a wide range of different unbalanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries). For example, PostgreSQL includes SP-GiST operator classes for two-dimensional points, which support indexed queries with the following operators: `<<`, `>>`, `~=`, `<@`, `<^`, and `>^`. These operators are described in the Geometric Functions and Operators article. The SP-GiST operator classes included in the standard distribution are listed in the Built-in SP-GiST Operator Classes table. For more information, see SP-GiST Indexes.

SP-GiST also supports the "nearest-neighbor" searches. For SP-GiST operator classes that support distance ordering, the corresponding operator is listed in the **Ordering Operators** column in the Built-in SP-GiST Operator Classes table.

The following example creates an SP-GiST index for a field storing IP addresses:

```
CREATE INDEX spg_inet ON my_table USING spgist (ip_address);
```

```
SELECT * FROM my_table WHERE ip_address << '192.168.1.0/24';
```

GIN

GIN (Generalized Inverted Index) indexes are inverted indexes that are appropriate for data values with multiple component values. They are used for full-text search and search in arrays. GIN indexes can be combined together in queries that involve searching large numbers of rows. An inverted index contains a separate entry for each component value, and can work effectively to quickly find the check for the presence of certain component values.

GIN indexes are capable of optimizing searches with the following operators: `<@`, `@>`, `=`, and `&&`. The GIN operator classes included in the standard distribution are listed in the Built-in GIN Operator Classes table. For more information, see GIN Indexes.

The GIN operator classes included in the standard distribution are listed in the Built-in GIN Operator Classes table. Many other GIN operator classes can be found in the contrib collection and in the PostGIS project.

The following example creates a GIN index for full-text search:

```
CREATE INDEX ind_gin ON my_table USING gin (to_tsvector('english', ind_column));
```

BRIN

BRIN (Block Range Indexes) indexes store summaries about the values saved in consecutive physical block ranges of a table. They are most effective for columns whose values are well correlated with the physical order of the table rows. BRIN indexes can handle queries that use the following operators: `<`, `<=`, `=`, `>=`, `>`, and `IS NULL`.

The BRIN operator classes included in the standard distribution are listed in the Built-in BRIN Operator Classes table. For more information, see BRIN Indexes.

The following example creates a BRIN index:

```
CREATE INDEX ind_brin ON my_table USING brin (ind_column);
```

Manage indexes

You can create, alter, and drop indexes using corresponding commands.

The CREATE INDEX command builds an index on the specified column or columns of a relation that can be a table or a materialized view. By default, `CREATE INDEX` creates B-tree indexes, which are efficient in most cases. The following example creates a B-tree index on the `TITLE` column in the `books` table:

```
CREATE INDEX table_idx ON books (title);
```

To change the index type, add the `USING` keyword with the index type name. For example, to create a hash index, use the following command:

```
CREATE INDEX name ON table USING HASH (column2);
```

Creating an index can interfere with regular operation of a database. Normally, PostgreSQL locks the table to be indexed against writes and performs the entire index build with a single scan of the table. While an index is being created, the table is usually still available for queries, but any `INSERT`, `UPDATE`, and `DELETE` commands that modify the table will be blocked until the index build is finished. This behavior can have a severe effect if the system is a production environment with large amounts of data. In some cases, users may tolerate read operations being delayed, but it is usually unacceptable to be unable to modify data for any extended time.

PostgreSQL supports building indexes without locking out writes. This method is invoked by specifying the `CONCURRENTLY` option of `CREATE INDEX`. When this option is used, PostgreSQL must perform two scans of the table, and in addition it must wait for all existing transactions that could potentially modify or use the index. This method requires more work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment.

You can use the `REINDEX` command to change the definition of an existing index. `ALTER INDEX` does not allow changing the index method. Note that the required lock level may differ for different `ALTER INDEX` forms and for `REINDEX`.

The following example renames an existing index:

```
ALTER INDEX distributors_idx RENAME TO suppliers_idx;
```

The example below moves an index to a different tablespace:

```
ALTER INDEX distributors_idx SET TABLESPACE my_tablespace;
```

To delete an index, utilize `DROP INDEX`.

This command removes the `suppliers_idx` index:

```
DROP INDEX suppliers_idx;
```

Rebuild indexes

PostgreSQL provides the `REINDEX` command that allows you to rebuild indexes. `REINDEX` rebuilds an index using the data stored in the index's table and replaces the old copy of the index. `REINDEX` can be useful in the following scenarios:

- An index is corrupted, and no longer contains valid data. Although in theory this should never happen, in practice indexes can become corrupted due to software bugs or hardware failures. `REINDEX` provides a recovery method.
- An index contains many empty or nearly-empty pages. This can occur with B-tree indexes in PostgreSQL under certain access patterns. `REINDEX` writes a new version of the index with the pages compacted, thus reducing the amount of disk space used by the index.
- You changed a storage parameter (such as `FILLFACTOR`) for an index, and wish to ensure that the change has full effect.
- If an index build was interrupted by `CONCURRENTLY` option, this index becomes invalid. Use `REINDEX` to rebuild it. Note that only the `INTEGRITY INDEX` command can perform a concurrent build on an invalid index.

The `REINDEX` command rebuilds the specified index, all indexes of the specified table, all indexes of the specified schema, all indexes within the current database, and all indexes on system catalogs within the current database. See REINDEX for details.

The following example rebuilds a single index:

```
REINDEX INDEX my_index;
```

The command below rebuilds all the indexes on the `my_table` table:

```
REINDEX TABLE my_table;
```

Multicolumn indexes

An index can be defined on more than one column of a table. For example, you have the following table:

```
CREATE TABLE authors (
  id SERIAL PRIMARY KEY,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL
);
```

And you frequently issue the queries:

```
SELECT * FROM authors WHERE first_name = 'value1' AND last_name = 'value2';
```

You can define an index on the `first_name` and `last_name` columns together:

```
CREATE INDEX idx_authors_names ON authors (last_name, first_name);
```

Currently, only the B-tree, GIST, GIN, and BRIN index types support multiple-key-column indexes. The ability to build an index on multiple key columns does not depend on the ability to add non-key columns to the index using `INCLUDE`. The number of columns in the index is limited to 32 with `INCLUDE` columns.

Multicolumn indexes can be used with queries that involve any subset of the index's columns, but are most efficient when there are constraints on the leading (first) columns. The exact rule is that equality constraints on leading columns, plus any inequality constraints on the first column that does not have an equality constraint, will make the index scan relatively efficient. However, these rules depend on PostgreSQL's traversal strategy for multicolumn B-tree indexes. For example, sometimes it is advantageous to add a redundant column to a multicolumn index, even if the query conditions don't specify that column. Partitioning in Multiple Indexes and Index-Only Scans and Covering Indexes.

Unique indexes

Indexes can also ensure that a value or a column is unique or a combination of values in multiple columns is unique. To create a unique index, utilize the `UNIQUE` keyword:

```
CREATE UNIQUE INDEX unique_name ON static (customer [, ...]);
```

For example:

```
CREATE UNIQUE INDEX ind_unique ON table1 (column1, column2);
```

Currently, only B-tree indexes can be declared as unique.

If an index is declared unique, multiple table rows with the same index key values cannot be added to the table. The `NULL` values are not considered equal. A multicolumn unique index only rejects rows in which all indexed columns contain the same values.

PostgreSQL defines a unique index when a unique constraint or primary key is specified for a table. The index is created for the primary key columns or a unique constraint. It can be a multicolumn index. This mechanism enforces the constraint. The way you do not need to manually create indexes for unique columns — they will duplicate the automatically created index.

Indexes on expressions

An index can be created on a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.

For example, a common way to do case-insensitive comparisons is to use the `lower()` function:

```
SELECT * FROM table1 WHERE lower(column1) = 'value1';
```

The query can use the index that is defined on the result of `lower(column1)` function:

```
CREATE INDEX idx_lower_column1 ON table1 (lower(column1));
```

If this index is declared as unique `UNIQUE`, it will prevent the creation of rows where the values of `column1` differ only in case and of the values of `column1` are the same. This way, indexes on expressions can be used to enforce constraints that cannot be written as simple uniqueness constraints.

As another example, the following query can be often executed:

```
SELECT * FROM authors WHERE (first_name || ' ' || last_name) = 'Ivan Ivanov';
```

Then it might be worth to create the following index:

```
CREATE INDEX idx_names ON authors ((first_name || ' ' || last_name));
```

The syntax of the `CREATE INDEX` command requires writing the additional parentheses around index expressions as shown in the second example. The parentheses can be omitted when the expression is a function call.

When an index is defined on an expression, the query planner may consider using the index when the index expressions are not recognized during an extended query, since they are already stored in the index. In such expressions above, PostgreSQL considers the query as `WHERE (indexed_column) = (constant)` and so the indexed column can appear to be a constant. This happens if the index expressions are used when retrieval speed is more important than insertion and update speed.

Partial indexes

A partial index is an index built over a subset of a table. The subset is defined by a conditional expression called the predicate of the partial index. The index contains entries only for table rows that satisfy the predicate.

The main purpose of partial indexes is to avoid indexing common values. If a value is contained in a significant percentage of all rows, a query searching for this value does not use an index. So, there is no point to keep these rows in the index. It reduces the size of the index and speed up the queries that use this index. It also makes faster table update operations since the index is not always updated.

Consider the following example: web server access logs are stored in a database.

```
CREATE TABLE access_log (
  url varchar,
  client_ip inet,
  ...
);
```

Most connections come from the organization's IP address range, but some come from other sources, such as remote employee connections. If you usually search by IP of outside accesses, you do not need to index the IP range that corresponds to the organization subnet. It is possible to create the following index:

```
CREATE INDEX log_ip_idx ON access_log (client_ip)
  WHERE NOT (client_ip > inet '192.168.100.0' AND
             client_ip < inet '192.168.100.255');
```

A typical query that can use this index:

```
SELECT * FROM access_log
  WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

The following query cannot use a partial index because it uses an IP address that is excluded from the index.

```
SELECT * FROM access_log
  WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

This type of partial index requires that the common values be predetermined, so such partial indexes are best used for data distributions that do not change. These indexes can be recreated occasionally to adjust for new data distributions, but this makes them much more difficult to maintain.

As a second example, consider a table that stores both paid and unpaid invoices. The unpaid invoices are only a small part of the entire table, but are the most interesting. You can improve query performance by creating an index on only unpaid invoices. The following command creates the index:

```
CREATE INDEX orders_unpaid_idx ON orders (order_nr)
  WHERE paid is not true;
```

Use the following query:

```
SELECT * FROM orders WHERE paid is not true AND order_nr < 10000;
```

However, the index can also be used in queries that do not involve `order_nr`:

```
SELECT * FROM orders WHERE paid is not true AND amount > 5000.00;
```

This is not as efficient as a partial index on the `amount` column, since the system should scan the entire index. If there are relatively few unpaid orders, using this partial index just to find the unpaid orders is efficient.

Note that the following query cannot use this index:

```
SELECT * FROM orders WHERE order_nr = 4201;
```

The order `SQL` can be among the paid or unpaid orders.

This example shows that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates if they include only the columns that the table contains. However, keep in mind that the predicate must match the conditions of the queries that the index is intended to optimize.

The third possible use of partial indexes does not involve using the index in queries. The idea is to create a unique index on a subset of the table's rows. This enforces uniqueness among the rows that are included in the index, without constraining those that are not.

Suppose that a table describes test results. It is necessary to ensure that there is only one "successful" entry for a given subject and target combination, but that there are many "unsuccessful" entries. To achieve this goal, you can use the following index:

```
CREATE TABLE tests (
  subject text,
  target text,
  success boolean,
  ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
  WHERE success;
```

This approach will be especially effective when there are more unsuccessful attempts than successful ones.

Examine the use of indexes

Although indexes in PostgreSQL do not need maintenance or tuning, it is important to check which indexes are actually used by the real-world queries. Examine the use of indexes for each query's execution plan.

It is also possible to collect overall statistics about index usage in a running server, as described in The Statistics Collector. For example, the `pg_stat_user_tables` and `pg_stat_all_indexes` views are useful for determining which indexes to consider. If a table has a lot of sequential scans, but its indexes are not used, then it is worth investigating why.

It is difficult to formulate a general approach for determining which indexes to create. Often a lot of experimentation is necessary. You can also use the following tips:

- Always run ANALYZE first. This command collects statistics about the distribution of the table. This information is required to estimate the number of rows returned by a query, which is necessary for the planner to make reasonable choices. Without these statistics, some default values are assumed that are almost certainly not to be true. It is almost impossible to understand which indexes are used without running `ANALYZE`. See Updating Planner Statistics and The Autovacuum Daemon for details.
- Use real data for experimentation. By analyzing the system's performance with test data, you will understand what indexes are needed for real data, but nothing more. Experimenting with data sets that are too small is misleading because the system will find that small tables are faster without indexes (it will usually be right). Data sets that are too large will take a long time to generate and index. Generate a moderate-sized data set that is representative of your real data, and then verify that the indexes you create are used often enough.
- Also, be careful when making up test data whose distribution is not uniform. Often values that are very similar, completely random, or inserted in sorted order, you will get very different results than your application is likely to produce.
- When indexes are not used, it can be useful for testing to force their use. PostgreSQL provides run-time parameters that can turn off various plan types (see Planner Method Configuration). For example, turning off `enable_seqscan` and `enable_sort` will cause the system to use an index scan for many queries that would otherwise choose a sequential scan or explicit sort. This is a very crude test, but useful when it is not clear why some index is not being used. For example, the query condition does not match the index.
- If the system uses only indexes when it is forced, then you need to determine either the system is right and the use of the index is actually ineffective, or the cost estimates of the system are not reflecting reality. In the case, you should measure the execution time of the query with and without indexes. The `EXPLAIN ANALYZE` command can be useful in analyzing this situation.
- If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed as the sum of each row's cost and of per-page cost estimates multiplied by the values of parameters. The costs estimated for the plan nodes can be adjusted via run-time parameters described in Planner Cost Constants. On the other side, selectivity estimates are the result of the planner's guesses. They can be improved by using the statistics-gathering parameters (see ALTER TABLE).

The total cost is calculated as the cost of each row of each plan node multiplied by the selectivity estimate of the plan node.

Triggers

A Superb Author

Contents

Overview

Trigger types

- Row-level and statement-level triggers
- BEFORE, AFTER, and INSTEAD OF triggers

Operations with triggers

- Create a trigger
- Modify a trigger
- Disable a trigger
- Delete a trigger
- List triggers

Trigger behavior

- The AFTER trigger
- Partitioned and parent tables
- Return values of trigger functions
- Generated columns
- Trigger firing order
- Trigger WHEN condition
- Cascading triggers

Trigger arguments

Visibility of data changes

Overview

A trigger is a specification that the PostgreSQL should execute a particular function whenever a certain type of operation is performed. Triggers can be attached to tables, views, and foreign tables.

On tables and foreign tables, you can define triggers to execute either before or after any `INSERT`, `UPDATE`, or `DELETE` operation, either once per modified row, or once per SQL statement. `UPDATE` triggers can also be configured to fire only if certain columns are mentioned in the `SET` clause of the `UPDATE` statement. Triggers can also fire for `TRUNCATE` statements.

If a trigger event occurs, the trigger's function is called to handle the event. Trigger functions can be written in most of the available procedural languages, including PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python.

On views, triggers can be defined to execute instead of `INSERT`, `UPDATE`, or `DELETE` operations. `INSTEAD OF` triggers are fired once for each view row that should be modified. The trigger function is responsible for making changes in the view's underlying tables and, if necessary, returning the modified view row as they will appear in the view. Triggers on views can also be defined to execute before or after `INSERT`, `UPDATE`, or `DELETE` operations. However, these triggers are fired only if there is also an `INSTEAD OF` trigger on the view. Otherwise, the statement targeting the view must be rewritten into a statement operating with its underlying base tables, and the triggers attached to the base tables will be fired.

The trigger function must be defined before the trigger creation. The trigger function must be declared as a function with no arguments and returning the `TRIGGER` type. The same function can be used for multiple triggers.

Trigger types

Row-level and statement-level triggers

PostgreSQL offers per-row triggers and per-statement triggers. With a per-row trigger, PostgreSQL invokes a trigger function once for each row affected by the statement fixing the trigger.

Conversely, invokes a per-statement trigger only once when an appropriate statement is executed, regardless of the number of rows processed by this statement. If a statement changes zero rows, it also raises per-statement triggers.

These two types of triggers are called **row-level triggers** and **statement-level triggers**, respectively.

Triggers on the `TRUNCATE` statement can only be defined at statement level, not at row level.

BEFORE, AFTER, and INSTEAD OF triggers

Triggers are also classified according to whether they fire before, after, or instead of the operation. Statement-level `BEFORE` triggers fire before the statement starts during. statement-level `AFTER` triggers raise at the end of the statement. These types of triggers may be defined on tables, views, or foreign tables.

Row-level `BEFORE` triggers fire immediately before a row is processed, row-level `AFTER` triggers fire at the end of the statement but before any statement-level `AFTER` triggers. These types of triggers are not recommended to use with views.

`INSTEAD OF` triggers can only be defined on views, and only at row level. They fire as soon as the view row is identified as being processed.

`BEFORE` triggers are typically used for checking or modifying the data that will be inserted or updated. For example, a `BEFORE` trigger can be used to insert the current time into a `timestamp` column or to check that two elements of the row are consistent. Row-level `AFTER` triggers are effective for cascading updates of other tables or checking the consistency of the updated row against other tables.

An `AFTER` trigger can access the final value of the row, while a `BEFORE` trigger does not, because there may be other `BEFORE` triggers that fire later. If there is no special reason to choose between `BEFORE` or `AFTER` triggers, the `BEFORE` trigger is preferable because it does not require the information about the operation be retained until the end of the statement.

Operations with triggers

Create a trigger

To create a new trigger you need to perform the following steps:

1. Create a trigger function.
2. Use the `CREATE TRIGGER` statement to bind the trigger function to a relation.

Consider an example that logs changes – stores changes to an existing table in a separate table. The example uses a `BEFORE UPDATE` trigger.

There is a `books` table that keeps information about books:

```
CREATE TABLE books (
  id SERIAL PRIMARY KEY,
  book_name VARCHAR NOT NULL,
  price NUMERIC NOT NULL
);

INSERT INTO
  books( book_name, price)
VALUES
  ('Hyperion',25),
  ('Man and Fauna',26),
  ('1984',28),
  ('The Time Machine',15);
```

Create a new table to save changes:

```
CREATE TABLE books_log (
  id SERIAL PRIMARY KEY,
  book_id INT,
  book_name VARCHAR NOT NULL,
  price NUMERIC NOT NULL,
  change_date TIMESTAMP NOT NULL
);
```

Create a trigger function. Use a procedural language for this. The code below illustrates the syntax of a PL/pgSQL trigger function. It stores old values from `books` and date of changes to the `books_log` table:

```
CREATE OR REPLACE FUNCTION save_changes()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
  AS
  $$
  BEGIN
      IF (NEW.book_name <> OLD.book_name) OR (NEW.price <> OLD.price) THEN
      INSERT INTO books_log(book_id,book_name,price,change_date)
      VALUES(OLD.id,OLD.book_name,OLD.price, now());
      END IF;

      RETURN NEW;
  END;
  $$
```

When a PL/pgSQL function is called as a data change trigger, several special variables are created automatically in the top-level block. They are listed in the table below. The code above uses the `NEW` and `OLD` variables to access the new and old table row, respectively.

PL/pgSQL data change trigger function variables

Name	Data type	Description
NEW	record	A new table row for `INSERT`/`UPDATE` operations in row-level triggers. This variable is `NULL` in statement-level triggers and for `DELETE` operations.
OLD	record	An old table row for `UPDATE`/`DELETE` operations in row-level triggers. This variable is `NULL` in statement-level triggers and for `INSERT` operations.
TG_NAME	name	A name of the fired trigger
TG_WHEN	text	`BEFORE`, `AFTER`, or `INSTEAD OF`, depending on the trigger's definition
TG_LEVEL	text	`ROW` or `STATEMENT`, depending on the trigger's definition
TG_OP	text	An operation for which the trigger was fired: `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE`.
TG_RELID	oid (references pg_class.oid)	An object ID of the table that caused the trigger invocation
TG_TABLE_NAME	name	A name of the table that caused the trigger invocation
TG_TABLE_SCHEMA	name	The name of the schema that contains the table causing the trigger invocation
TG_NARGS	integer	The number of arguments given to the trigger function in the `CREATE TRIGGER` statement
TG_ARGV[]	Data type array of text	The arguments from the `CREATE TRIGGER` statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to `TG_NARGS`) return `NULL`.

For more information, see Trigger functions.

Use the `CREATE TRIGGER` command to define a new trigger. To create or replace a trigger on a table, the user must have the `TRIGGER` privilege on the table. The user must also have `EXECUTE` privilege on the trigger function.

```
CREATE TRIGGER last_changes
  BEFORE UPDATE
  ON books
  FOR EACH ROW
  EXECUTE PROCEDURE save_changes();
```

The `last_changes` trigger is created as row-level `BEFORE UPDATE` trigger on the `books` table that invokes the `save_changes` trigger function. For more information on the `CREATE TRIGGER` syntax, see the `CREATE TRIGGER` article.

Update the `books` table to fire the trigger:

```
UPDATE books SET price = 52 WHERE ID = 3;
```

Check the contents of the `books_log` table:

```
SELECT * FROM books_log;
```

The result:

```
 id | book_id | book_name | price |      change_date
----+---------+-----------+-------+------------------------
  1 |       3 | 1984      |    28 | 2024-04-04 14:38:07.951503
```

Modify a trigger

Use the `ALTER TRIGGER` command to modify the trigger.

The following example renames the trigger:

```
ALTER TRIGGER last_changes ON books RENAME TO last_changes_new_name;
```

Disable a trigger

To temporarily disable or enable a trigger, utilize the `ALTER TABLE` command.

Disable a trigger:

```
ALTER TABLE books DISABLE TRIGGER last_changes_new_name;
```

Enable a trigger:

```
ALTER TABLE books ENABLE TRIGGER last_changes_new_name;
```

Delete a trigger

To remove a trigger, use the `DROP TRIGGER` command:

```
DROP TRIGGER last_changes_new_name ON books;
```

To execute this command, the current user must be the owner of the table for which the trigger is defined.

List triggers

To list all triggers in the current database, use the `information_schema.triggers` system view:

```
SELECT event_object_table, trigger_name FROM information_schema.triggers;
```

The result:

```
 event_object_table | trigger_name
--------------------+--------------
 books              | last_changes
```

It is also possible to utilize the `pg_trigger` system catalog to list triggers:

```
SELECT via, tgrelid, tgparentid, tgname FROM pg_trigger;
```

The result:

```
  oid   | tgrelid | tgparentid |     tgname
--------+---------+------------+-----------------
 191265 |  190751 |          0 | last_changes
```

Trigger behavior

This section describes some trigger behavior that you may find important.

The AFTER trigger

The execution of an `AFTER` trigger can be deferred to the end of the transaction, rather than the end of the statement, if it was defined as a `CONSTRAINT` trigger. In all cases, a trigger is executed as part of the same transaction as the statement that triggered it, so if the statement or the trigger causes an error, the effects of both will be rolled back.

Partitioned and parent tables

A statement that targets a parent table in an inheritance or partitioning hierarchy does not cause the statement-level triggers of affected child tables to be fired. Only the parent table's statement-level triggers are fired. However, row-level triggers of any affected child tables will be fired.

If `UPDATE` on a partitioned table causes a row to move to another partition, it will be performed as `DELETE` from the original partition followed by `INSERT` into a new partition. In this case, all row-level `BEFORE UPDATE` and `BEFORE DELETE` triggers are fired on the original partition. Then, all row-level `BEFORE INSERT` triggers are fired on the destination partition. You should be aware of the possibility of unexpected results when all of these triggers process the row being moved. Row-level `AFTER DELETE` and `AFTER INSERT` triggers are applied but `AFTER UPDATE` triggers are not applied because the `UPDATE` has been converted to `DELETE` and `INSERT`. The statement-level `DELETE` and `INSERT` triggers are not fired, even if rows are moved. Only `UPDATE` triggers defined on the target table of the `UPDATE` statement will be fired.

Return values of trigger functions

Functions invoked by statement-level triggers should return `NULL`. Trigger functions called by row-level triggers can return a table row (a value of the `RowTuple` type) to the calling executor. A row-level `BEFORE` trigger can return one of the following results:

- `NULL` — to skip the operation for the current row. This instructs the executor to not perform an `INSERT`, `UPDATE`, or `DELETE` operation on the specific table row that invokes the trigger.
- A table row — for `INSERT` and `UPDATE` triggers only, the returned row becomes the row that will be inserted or will replace the row being updated. This allows the trigger function to modify the row being inserted or updated.

If you do not plan to use the options mentioned above with a row-level `BEFORE` trigger, its trigger function must return the same row that is passed in – the new row for `INSERT` and `UPDATE` triggers, and the old row for `DELETE` triggers.

A row-level `INSTEAD OF` trigger should either return `NULL` to indicate that it did not modify any data from the view's base tables, or it should return the view row that was passed in (the new row for `INSERT` and `UPDATE` triggers, or the old row for `DELETE` triggers). A non-null return value signals that the trigger performed the necessary data modifications in the view. This will increment the counter for the number of rows affected by the command. For `INSERT` and `UPDATE` operations only, the trigger function can modify a new row before returning it.

The return value is ignored for row-level `AFTER` triggers, and so they can return `NULL`.

Generated columns

After defining a trigger, you can specify generated columns. Stored generated columns are computed after `BEFORE` triggers and before `AFTER` triggers. The generated values can be inspected or modified in `BEFORE` triggers. The expression of the row `OLD` variable contains the old generated value, but the new row does not yet contain the new generated value. Changes of a generated column in a `BEFORE` trigger are ignored and will be overwritten.

Trigger firing order

If more than one trigger is defined for the same event on the same relation, the triggers are fired in alphabetical order by trigger name. In the case of `BEFORE` and `INSTEAD OF` triggers, the possibly-modified row returned by each trigger becomes the input to the next trigger. If any `BEFORE` or `INSTEAD OF` trigger returns `NULL`, the operation for this row stops, and subsequent triggers are not fired for this row.

Trigger WHEN condition

A trigger definition can also contain a boolean `WHEN` condition that is checked to see whether the trigger should be fired. In row-level triggers, the `WHEN` condition can examine the old row, the new row, or both. Statement-level triggers can also include `WHEN` conditions, although the feature is not so useful for them. In a `BEFORE` trigger, the `WHEN` condition is evaluated just before the function is executed, so using `WHEN` is not significantly different from testing the same condition at the beginning of the trigger function. However, in an `AFTER` trigger, the `WHEN` condition is evaluated immediately after the row is updated, and it determines whether an event is queued to fire the trigger at the end of the statement. Therefore, when the `WHEN` condition in an `AFTER` trigger does not return `true`, there is no need to enlist either queue the event or to fetch this row at the end of the statement. This can significantly speed up statements that modify a large number of rows if a trigger with only rarely needs to fire on a few. `INSTEAD OF` triggers do not support the `WHEN` condition.

Cascading triggers

If a trigger function executes SQL commands, those commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger. For example, an `INSERT` trigger can execute a command that inserts an additional row into the same table, causing the `INSERT` trigger to be fired again. It is the programmer's responsibility to avoid endless recursion in such cases.

Trigger arguments

When defining a trigger, you can specify arguments. The purpose of including arguments in a trigger definition is to allow different triggers with similar requirements to call the same function. As an example, there could be a generalized trigger function that takes as its first argument a table name, and as its second argument a column name. This input data includes the type of trigger event (for example, `INSERT` or `UPDATE`) and arguments that are listed in the `CREATE TRIGGER` command. For row-level triggers, the input data also includes the new row for `INSERT` and `UPDATE` triggers and the old row for `UPDATE` and `DELETE` triggers.

Visibility of data changes

If you execute SQL commands in the trigger function, these SQL commands see the data changes according to the following data visibility rules:

- None of the changes made by a statement are visible to statement-level `BEFORE` triggers, but all modifications are visible to statement-level `AFTER` triggers.
- The data change (insertion, update, or deletion) causing the trigger to fire is not visible to SQL commands executed in a row-level `BEFORE` trigger because it has not yet occurred.
- SQL commands executed in a row-level `BEFORE` trigger see the effects of data changes for rows previously processed in the same command. This requires caution, since in SQL command that processes on multiple rows can access the rows in any order.
- A row-level `INSTEAD OF` trigger can see the effects of data changes made by previous `INSTEAD OF` trigger calls in the same command.
- When a row-level `AFTER` trigger is fired, all data changes made by the command are already complete, and are visible to the invoked trigger function.

If a trigger function is written in any of the standard procedural languages, the above statements are correct only if the function is declared as `VOLATILE`. Functions that are declared as `STABLE` or `IMMUTABLE` will not see changes made by the calling command.

User-defined functions and procedures

A reference guide

AD.PG

To Table of Contents

- Overview
  - User-defined functions
  - User-defined procedures
- Query language (SQL) functions
  - SQL functions on base types
  - SQL functions on composite types
  - SQL functions with output parameters
  - SQL procedures with output parameters
  - SQL functions with variable numbers of arguments
  - SQL functions returning sets
  - SQL functions returning a table
- Procedural language functions

Overview

User-defined PostgreSQL functions and procedures are designated to perform custom tasks. They are usually used to handle specific scenarios.

User-defined functions

ADP.PG.PostgreSQL provides the following types of functions:

- [Query language functions](#) functions written in SQL. The type of function is determined by the last query in the function.
- [Procedural language functions](#) functions written in other languages besides SQL and C. These other languages are generically called procedural languages (PL). For example, PL/pgSQL, PL/Tcl, PL/Perl. Procedural languages are not built into the PostgreSQL server; they are offered by loadable modules.
- Internal functions are built-in internal routines (C) that are statically linked to the ADP.PG.PostgreSQL server. All internal functions are declared during the initialization of the database cluster (initdb) and use `CREATE FUNCTION` to create an additional alias name for an internal function. [Internal functions](#)
- [C-language functions](#) are compiled into machine code. These functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand. The dynamic loading feature is what distinguishes "C language" functions from "internal" functions. The actual coding conventions are essentially the same for both. See [C-language functions](#).

Every type of function can take base and composite types as arguments (parameters). Also, functions can return a base type, a composite type, or a set of bases and composite values.

You can use the `TRUSTED PostgreSQL` command to define a new function. `ALTER FUNCTION` changes the definition of a function, and `DROP FUNCTION` removes a function. `SHOW FUNCTION` displays the source code of a function. All these commands can be used with the same name and different argument lists. See [Function overloading](#).

User-defined procedures

A procedure is a database object similar to a function. The key differences are:

- Procedures are defined by the `CREATE PROCEDURE` command.
- Procedures do not return a value. Therefore, `CREATE PROCEDURE` does not contain the `RETURNS` clause. However, procedures can return data via output parameters.
- A function is called as part of a query or DML command. a procedure is called in isolation using the `CALL` command.
- A procedure can execute a list of basic operations or other procedures. [Procedures](#)
- Functions can execute transactions (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, or whatever) via `RETURNS TABLE(isolation)`. In this case, all invoker must `SET search_path` explicitly.
- Procedures can execute transactions control statements (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, or whatever) via `CALL` command.

The body of a function or procedure is a sequence of statements separated by semicolon. A procedure after the last statement is optional. SQL statements can include the `SELECT`, queries, data-modification queries (`INSERT`, `UPDATE`, and `DELETE`), and other SQL commands. You cannot use the transaction control commands (for example, `COMMIT`, `SAVEPOINT`) in the body of a function or procedure. However, `EXECUTE` in `SQL` functions. The last statement must be `SELECT`, `INSERT`, `UPDATE`, or `DELETE` command for the `RETURNING` clause.

It is also possible to create an SQL function that performs actions but does not return a value. To do this, define a function as returning `void`. For example, the function below removes rows with negative values from the `quantities` table:

```
CREATE FUNCTION delete_bases() RETURNS void AS $$
    DELETE FROM bases
    WHERE quantity < 0;
$$ LANGUAGE SQL;

SELECT delete_bases();
```

You can also execute this code as a procedure:

```
CREATE PROCEDURE delete_bases_proc() AS $$
    DELETE FROM bases
    WHERE quantity < 0;
$$ LANGUAGE SQL;

CALL delete_bases_proc();
```

The syntax of the `CREATE FUNCTION` and `CREATE PROCEDURE` commands requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see [Lexical structure](#)) for the string constant. If you choose to use the escape string syntax `E''`, you must double the single quote marks (`''`) and [backslashes](#) (`\`) to escape symbols in the body of the function (see [String constants](#)).

Query language (SQL) functions

SQL functions execute an arbitrary list of SQL statements and return the result of the last query in the list. Use the `CREATE FUNCTION` command to define a function.

In the simple case, the first row of the last query's result will be returned. Ensure that the first row of a multi-row result is only provided if you use `ORDER BY` at the last query's result. Otherwise, the first row is unpredictable.

If the last query happens to return no rows at all, the null value will be returned. To do this, specify the function return type as `SETOF sometype` or declare it as `RETURNS TABLE(isolation)` parameter. In this case, all rows of the last query's result will be returned.

The body of a SQL function must be a list of SQL statements separated by semicolon. A function after the last statement is optional. SQL statements can include the `SELECT`, queries, data-modification queries (`INSERT`, `UPDATE`, and `DELETE`), and other SQL commands. You cannot use the transaction control commands (for example, `COMMIT`, `SAVEPOINT`) in the body of a function or procedure. However, `EXECUTE` in `SQL` functions. The last statement must be `SELECT`, `INSERT`, `UPDATE`, or `DELETE` command for the `RETURNING` clause.

It is also possible to create an SQL function that performs actions but does not return a value. To do this, define a function as returning `void`. For example, the function below removes rows with negative values from the `quantities` table:

```
CREATE FUNCTION delete_bases() RETURNS void AS $$
    DELETE FROM bases
    WHERE quantity < 0;
$$ LANGUAGE SQL;

SELECT delete_bases();
```

You can also execute this code as a procedure:

```
CREATE PROCEDURE delete_bases_proc() AS $$
    DELETE FROM bases
    WHERE quantity < 0;
$$ LANGUAGE SQL;

CALL delete_bases_proc();
```

The syntax of the `CREATE FUNCTION` and `CREATE PROCEDURE` commands requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see [Lexical structure](#)) for the string constant. If you choose to use the escape string syntax `E''`, you must double the single quote marks (`''`) and [backslashes](#) (`\`) to escape symbols in the body of the function (see [String constants](#)).

SQL functions on base types

The simplest SQL function has no arguments and returns a base type:

```
CREATE FUNCTION my_function() RETURNS integer AS $$
    SELECT 1;
$$ LANGUAGE SQL;

SELECT my_function();
```

The result:

```
 my_function
-------------
           1
```

The following SQL function takes base types as arguments:

```
CREATE FUNCTION my_add(a integer, b integer) RETURNS integer AS $$
    SELECT a + b;
$$ LANGUAGE SQL;

SELECT my_add(123, 123) AS answer;
```

The result:

```
 answer
--------
    246
```

Arguments of an SQL function can be referenced in the function body using either names or numbers. The function above can be rewritten without argument names:

```
CREATE FUNCTION my_add(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

The function below performs calculations and returns the updated balance as the result of the last `SELECT` command:

```
CREATE FUNCTION functionaccount_id integer, income numeric) RETURNS numeric AS $$
    UPDATE accounts
    SET balance = balance + income
    WHERE account_id = functionaccount_id;
    SELECT balance FROM accounts WHERE account_id = functionaccount_id;
$$ LANGUAGE SQL;
```

In this example, the name of the first argument is `account_id`, and the name of a table column is the same — `account_id`. To refer to the argument in the `UPDATE` command, use the argument name with the function name: `functionaccount_id`. Otherwise, you can specify a different argument name.

The `functionaccount_id` function can rewritten with the `RETURNING` clause:

```
CREATE FUNCTION functionaccount_id integer, income numeric) RETURNS numeric AS $$
    UPDATE accounts
    SET balance = balance + income
    WHERE account_id = functionaccount_id
    RETURNING balance;
$$ LANGUAGE SQL;
```

> **NOTE**
>
> If the first `SELECT` or `RETURNING` clause in an SQL function does not return the functions declared result type, PostgreSQL will cast the value to the required type if that is possible with an implicit or assignment cast. Otherwise, you must explicitly cast the result. For more information, see [type conversion](#) and [CREATE CAST](#).

Functions can be declared with default values for some or all of their input arguments. Default values are used when a function is called with an insufficient number of arguments. Since arguments can only be omitted from the end of the argument list, all parameters after a parameter with a default value will also receive their default values. For example:

```
CREATE FUNCTION foo(a int DEFAULT 1, b int DEFAULT 2)
RETURNS int AS $$
    SELECT a * b;
$$ LANGUAGE SQL;

SELECT foo(), foo(8), foo(8, 10);
```

The result is `2`, `16`.

You can call it with a single argument `8`:

```
SELECT foo(8);
```

The result is `16`.

SQL functions on composite types

When writing functions with composite SQL types, it is necessary to specify not only the argument but also which attribute (field) of the argument should be used.

For example, create the following table:

```
CREATE TABLE books (
    id BIGSERIAL PRIMARY KEY,
    title VARCHAR(UNICODE) NOT NULL,
    author VARCHAR(UNICODE) NOT NULL,
    pages integer
);

INSERT INTO books (title, author, pages) VALUES
('Mrs. Lieutenant', 'Life', 259),
('To the Lighthouse',),
('I'll Give a Passingbell',),
('The Blue Lagoon',);
```

Define a function that adds the tax to a price:

```
CREATE FUNCTION add_tax(books) RETURNS numeric AS $$
    SELECT $1.price * 1.1;
$$ LANGUAGE SQL;

SELECT title, add_tax(books) AS price_with_tax
    FROM books;
```

The result:

```
          title          | price_with_tax
-------------------------+----------------
 Mrs. Dalloway           |           60.0
 To the Lighthouse       |           54.0
 I'll Kill a Mockingbird |           37.4
 The Great Gatsby        |           33.0
```

The required field of the argument table is specified after dot (`.`) - `$1.price`. To select the entire row of a table as a composite value, the asterisk (`*`) is used - `books.*` or `SomeArgumentName.*` argument.

SQL functions with output parameters

An alternative way of describing a function's results is to define it with output parameters. Output parameters provide a convenient way of creating functions that return several columns. For example:

```
CREATE FUNCTION function_sql(a int, b int, OUT sum int, OUT product int)
    SELECT a + b, a * b;
$$ LANGUAGE SQL;

SELECT * FROM function_sql(5,2);
```

The result:

```
 sum | product
-----+---------
   7 |      10
```

Note, that the code above creates an anonymous composite type for the function's return value. It has the same result as the following code:

```
CREATE TYPE my_type AS (sum int, product int);

CREATE FUNCTION function_sql(int, int) RETURNS my_type AS $$
    SELECT $1 + $2, $1 * $2;
$$ LANGUAGE SQL;

SELECT * FROM function_sql(5,2);
```

Output parameters are not included in the calling argument list. PostgreSQL considers only the function's input parameters to define the function's calling signature. That means also that only the input parameters matter when referencing the function for purposes such as dropping it. We could drop the above function with one of the following commands:

```
DROP FUNCTION function_sql(a int, b int, OUT sum int, OUT product int);
DROP FUNCTION function_sql(int, int);
```

SQL functions parameters can be marked as follows:

- `IN` - input parameter; a part of the calling argument list
- `OUT` - output parameter; a part of the function's return value
- `INOUT` - parameter serves as both an input parameter (part of the calling argument list) and an output parameter (part of the result list).
- `VARIADIC` - specifies that the function can accept a variable number of arguments. See [SQL functions with variable numbers of arguments](#).

SQL procedures with output parameters

Output parameters are also supported in procedures but they work a bit differently. Output parameters must be included in the argument list in the `CALL` command. For example, `function1` from the example above can be rewritten as follows:

```
CREATE PROCEDURE procedure1(int, int, income numeric, OUT new_balance numeric) AS $$
    UPDATE accounts
    SET balance = balance + income
    WHERE account_id = account_id
    RETURNING balance;
$$ LANGUAGE SQL;
```

To call this procedure, you must supply an argument corresponding to the `OUT` parameter. Typically, `NULL` is used:

```
CALL procedure1(1, 200.0, NULL);
```

SQL functions with variable numbers of arguments

SQL functions can be declared to accept a variable number of arguments. All optional arguments must be of the same data type. The optional arguments are passed to the function as an array. To do this, declare last parameter as an array type and mark it as `VARIADIC`. For example, the following function finds the maximum of an arbitrary number of numbers:

```
CREATE FUNCTION max_value(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT max(arr[i]) FROM generate_subscripts(arr, 1) AS gs(i);
$$ LANGUAGE SQL;

SELECT max_value(10,5,3,4,7);
```

The result:

```
 max_value
-----------
        10
```

SQL functions returning sets

When an SQL function returns `SETOF sometype`, each row of its output is returned as an element of the result set. It is often used with the `FROM` clause. For example:

```
CREATE TABLE employees (id int, report_to int, name text);
INSERT INTO employees VALUES (1, 0, 'Sara');
INSERT INTO employees VALUES (2, 1, 'Mary');
INSERT INTO employees VALUES (3, 1, 'Bill');
INSERT INTO employees VALUES (4, 2, 'Jane');
INSERT INTO employees VALUES (5, 2, 'Laura');

CREATE FUNCTION get_employees(int) RETURNS SETOF employees AS $$
    SELECT * FROM employees WHERE report_to = $1;
$$ LANGUAGE SQL;

SELECT * FROM get_employees(1);
```

The result:

```
 id | report_to | name
----+-----------+------
  2 |         1 | Mary
  3 |         1 | Bill
```

It is frequently useful to construct a query's result by invoking a set-returning function multiple times, with the parameters for each invocation coming from successive rows of a table or subquery. Use the `LATERAL` key word for this. For example, consider the following `tree` table:

```
 name  | parent
-------+--------
 Parent |
 Child1 | Parent
 Child2 | Parent
 Child3 | Parent
 Child4 | Child1
 Child5 | Child1
```

Create a function that selects children:

```
CREATE FUNCTION get_children(text) RETURNS SETOF text AS $$
    SELECT name FROM tree WHERE parent = $1
$$ LANGUAGE SQL STABLE;
```

Execute the following query:

```
SELECT name, child FROM tree, LATERAL get_children(name) AS child;
```

The result:

```
 name  | child
-------+--------
 Parent | Child1
 Parent | Child2
 Parent | Child3
 Child1 | Child4
 Child1 | Child5
```

> **NOTE**
>
> If the last command of a function is `INSERT`, `UPDATE`, or `DELETE` with `RETURNING`, the command will be executed to completion, even if the function is not declared with `SETOF(type)`. Any rows not produced by the `RETURNING` clause are simply ignored. The same rule applies to data-modification [triggers](#).

SQL functions returning a table

There is another way to declare a function as returning a set. You can use the `RETURNS TABLE(isolation)` clause. This is equivalent to using one or more `OUT` parameters and declaring the function as returning `SETOF sometype`. For example:

```
CREATE TABLE sb_table AS SELECT * FROM generate_series(1, 5);
INSERT INTO sb_table VALUES (2, 10), (3, 15), (4, 20), (5, 25);

CREATE FUNCTION function_sql(a int) RETURNS TABLE(sum int, product int) AS $$
    SELECT a + b, a * b FROM sb_table;
$$ LANGUAGE SQL;

SELECT * FROM function_sql(5);
```

The result:

```
 sum | product
-----+---------
   6 |       5
   7 |      10
   8 |      15
   9 |      20
  10 |      25
```

Procedural language functions

The standard ADP.PG.PostgreSQL distribution includes four procedural languages: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python.

Consider a function written in PL/pgSQL. All it does is to multiply by two the value supplied by the `CREATE FUNCTION` command:

```
CREATE FUNCTION my_function(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

PL/pgSQL is a block-structured language. The complete text of a function body must be a block. A block is defined as:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

A `label` is required if you want to identify the block for use in an `EXIT` statement or to qualify the names of the variables declared in the block. All variables used in a block must be declared in the `DECLARE` section of the block. See [declarations](#).

The following function retrieves the `passwd` column and returns the result of the password checking:

```
CREATE FUNCTION check_passwd(name TEXT, pass TEXT) RETURNS BOOLEAN AS $$
DECLARE passwd BOOLEAN;
BEGIN
    SELECT (pwd = $2) INTO passwd
    FROM   pwds
    WHERE  username = $1;

    RETURN passwd;
END
$$ LANGUAGE plpgsql
```

As in SQL functions, arguments can be referenced by their numbers.

For more information on PL/pgSQL functions, see PL/pgSQL - SQL Procedural Language.

Transaction isolation levels

Eugenia Kuzmina

AD.PG

Contents

To Table of Contents

Read committed isolation level

Repeatable read isolation level

Serializable isolation level

The SQL standard defines four levels of transaction isolation. The most strict is Serializable. According to the SQL standard, any concurrent execution of a set of Serializable transactions should produce the same effect as running them one at a time in some order. The other three levels are defined in terms of phenomena. These phenomena are the result of interactions between concurrent transactions. Some of them must not occur at a particular level of transaction isolation. None of these phenomena are possible at the Serializable level.

The following phenomena may occur:

- **Dirty read** – a transaction reads data written by a concurrent uncommitted transaction.
- **Non-repeatable read** – a transaction re-reads data it previously read and finds that the data has been modified by another transaction (that was committed since the initial read).
- **Phantom read** – a transaction re-executes a query that returns a set of rows for some condition and discovers that the set of rows satisfying the condition has changed due to another recently committed transaction.
- **Serialization anomaly** – the result of successfully committing a group of transactions is inconsistent with all possible orderings of running these transactions one at a time.

The transaction isolation levels are described in the table below. `Not allowed in ADPG/PostgreSQL` is used for phenomena that are allowed in the SQL standard but are prohibited in ADPG/PostgreSQL.

Isolation level	Dirty read	Non-repeatable read	Phantom read	Serialization anomaly
Read uncommitted	Not allowed in ADPG/PostgreSQL	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Not allowed in ADPG/PostgreSQL	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

In ADPG/PostgreSQL, you can request any of four standard transaction isolation levels, but internally only three distinct isolation levels are implemented. The PostgreSQL Read uncommitted mode behaves like Read committed. This is because it is the only way to map the standard isolation levels to the PostgreSQL multiversion concurrency control architecture.

The table also shows that the PostgreSQL Repeatable read implementation does not allow phantom reads. This is acceptable under the SQL standard because the standard specifies which anomalies must not occur at certain isolation levels. More stringent restrictions are acceptable.

IMPORTANT

Some PostgreSQL data types and functions have special rules regarding transactional behavior. In particular, changes made to a *sequence* (and therefore the counter of a column declared using `serial`) are immediately visible to all other transactions and are not rolled back if the transaction that made the changes aborts.

The default isolation level in ADPG/PostgreSQL is Read committed. To confirm this, run the following `psql` command:

```
SHOW TRANSACTION ISOLATION LEVEL;
```

The result:

```
 transaction_isolation
-----------------------
 read committed
(1 row)
```

To select the desired transaction isolation level, use the SET TRANSACTION command.

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
...
COMMIT;
```

It is also possible to set the transaction isolation level in the BEGIN and START TRANSACTION commands.

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
...
COMMIT;

START TRANSACTION ISOLATION LEVEL SERIALIZABLE;
...
COMMIT;
```

Read committed isolation level

As mentioned above, Read committed is the default isolation level in ADPG/PostgreSQL. When a transaction uses this isolation level, a `SELECT` query (without a `FOR UPDATE/SHARE` clause) sees only data committed before the query began. It never accesses either uncommitted data or changes committed during query execution by concurrent transactions. Essentially, a `SELECT` query sees a snapshot of the database at the time the query starts executing. However, `SELECT` sees the results of previous updates performed within its own transaction, even if the results have not yet been committed. Also note that two successive `SELECT` commands can get different data, even they are executed within a single transaction, if other transactions commit changes after the first `SELECT` starts and before the second `SELECT` starts.

The `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they only find target rows that were committed when the command started. However, a target row can be updated, deleted, or locked by another concurrent transaction by the time it is found. In this case, the update will wait for the first updating transaction to commit or roll back if it is still in progress. If the first updater rolls back, the second updater can proceed to update the originally found row. If the first updater commits, the second updater will attempt to apply its operation to the updated version of the row. It will ignore the row if the first updater deleted it. The search condition of the command (the `WHERE` clause) is re-evaluated to see if the updated row still matches the search condition. If so, the second modifying transaction continues with the modified version of the row. In the case of `SELECT FOR UPDATE` and `SELECT FOR SHARE`, the updated version of the row is locked and returned to the client.

`INSERT` with `ON CONFLICT DO UPDATE` clause behaves similarly. In Read committed mode, each row proposed for insertion will be either inserted or updated. If a conflict is caused by another transaction whose result is not yet visible to `INSERT`, the `UPDATE` clause will act on that row, even though possibly no version of that row is conventionally visible to the command.

In Read committed mode, when you issue an `INSERT` with an `ON CONFLICT DO NOTHING` clause, the row may not be added as a result of another transaction whose effect is not visible in the snapshot of the `INSERT` command.

Because of the above rules, it is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects on other rows in the database. This behavior makes the Read committed mode unsuitable for commands that involve complex search conditions. However, you can use it for simpler cases. For example, consider updating bank balances — transferring money from one account to another with the following transactions:

```
BEGIN;
UPDATE accounts SET balance = balance - 1000.00 WHERE acc_number = 5432;
UPDATE accounts SET balance = balance + 1000.00 WHERE acc_number = 1234;
COMMIT;

BEGIN;
UPDATE accounts SET balance = balance - 1000.00 WHERE acc_number = 4543;
UPDATE accounts SET balance = balance + 1000.00 WHERE acc_number = 1234;
COMMIT;
```

If these two transactions concurrently try to change the balance of the `1234` account, the second transaction must operate on the modified version of the row:

```
BEGIN;
UPDATE accounts SET balance = balance - 1000.00 WHERE acc_number = 5432;

/* Run from another session:
BEGIN;
UPDATE accounts SET balance = balance - 1000.00 WHERE acc_number = 4543;
UPDATE accounts SET balance = balance + 1000.00 WHERE acc_number = 1234;
COMMIT;
*/

UPDATE accounts SET balance = balance + 1000.00 WHERE acc_number = 1234;
COMMIT;
```

Since each command only affects a specific row, if the command sees a modified version of the row, this will not lead to inconsistency problems.

More complex scenarios can produce undesirable results in Read committed mode. For example, consider a `DELETE` command operating on data that is being both added and removed from its restriction criteria by another command. For example, `website` is a two-row table with `website.hits` that equal `9` and `10`. Perform the following transactions:

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- Run from another session: DELETE FROM website WHERE hits = 10;
COMMIT;
```

`DELETE` will have no effect, even though there is a row with `hits` equaling `10` before and after `UPDATE`. This occurs because the pre-update row value `9` is skipped, and when `UPDATE` completes and `DELETE` obtains a lock, the new row value is no longer `10` but `11`, which does not match the criteria.

The partial transaction isolation provided by Read committed mode is suitable for many applications. This mode is fast and easy to use, but it is not suitable for all situations. Applications that perform complex queries and changes may require a more rigorously consistent view of the database than the Read committed mode provides.

Repeatable read isolation level

In Repeatable read mode, only the data that was committed before the transaction started is visible, but uncommitted data and changes made by other transactions while the transaction was in progress are not visible. However, the query sees the effects of previous updates executed within its own transaction, even though they are not yet committed. This prevents all the phenomena described in the phenomena table except for serialization anomalies.

This level differs from Read committed in that a query in a Repeatable read transaction sees a snapshot of the data at the start of the first statement in the transaction (transaction control commands are not taken into account). Successive `SELECT` commands in the same transaction see the same data – they do not see changes made and committed by other transactions after the current transaction began.

Applications using this level must be prepared to retry transactions due to serialization failures.

The `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave like `SELECT` when searching for target rows. They only find target rows that were committed at the time the transaction began. However, by the time they are found, these target rows may already have been modified, deleted, or locked by another concurrent transaction. In this case, a transaction in Repeatable read mode waits for the first transaction that changes data to be committed or rolled back (if it is still in progress). If the first updater rolls back, the Repeatable read transaction can proceed with updating the originally found row. If the first updater commits and updates or deletes the row, not just locks it, the Repeatable read transaction will be rolled back with the message: `ERROR: could not serialize access due to concurrent update`. A Repeatable read transaction cannot modify or lock rows changed by other transactions after the Repeatable read transaction began. When an application receives this error message, it should abort the current transaction and try to retry it from the beginning. Note that only updating transactions might need to be retried. Read-only transactions will never have serialization conflicts.

The Repeatable read mode provides a rigorous guarantee that each transaction sees a completely stable view of the database. However, this view will not necessarily always be consistent with some serial execution of concurrent transactions of the same level.

The Repeatable read isolation level is implemented using a technique known as **Snapshot isolation**.

Serializable isolation level

The Serializable isolation level provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions, as if transactions are executed one after another, serially, rather than concurrently. However, as with the Repeatable read level, applications must be prepared to retry transactions due to serialization failures.

This isolation mode works in the same way as Repeatable read, only it additionally monitors for conditions under which the result of serializable transactions executed in parallel may not be consistent with the result of the same transactions executed in turn. This monitoring does not introduce any blocking beyond that present in Repeatable read, but it creates some additional overhead. If a serialization anomaly is detected, the serialization fails.

As an example, consider `table1` that initially contains the following data:

```
 id | category | value
----+----------+-------
  1 | a        |    10
  2 | a        |    20
  3 | b        |   300
  4 | b        |   400
```

Suppose that the serializable transaction 1 calculates the sum of values in the `a` category and inserts it in the table as a value of the `b` category:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO table1(id, category, value) VALUES
(5,'b',
(SELECT SUM(value) FROM table1 WHERE category = 'a'));
```

The serializable transaction 2 calculates the sum of values in the `b` category and inserts it in the table as a value of the `a` category:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
INSERT INTO table1(id, category, value) VALUES
(6,'a',
(SELECT SUM(value) FROM table1 WHERE category = 'b'));
```

Now, if you execute `COMMIT` of the transaction 1, it is committed without errors. Then, if you run `COMMIT` of the transaction 2, the following error occurs:

```
ERROR: could not serialize access due to read/write dependencies among transactions
```

This is because if the transaction 1 had executed before the transaction 2, the transaction 2 would have computed the sum `730`, not `700`, and similarly the other order would have resulted in a different sum computed by the transaction 1.

When relying on Serializable transactions to prevent anomalies, it is important that any data read from a permanent table not be considered valid until the transaction which read it has successfully committed. This is true even for read-only transactions, except that data read within a deferrable read-only transaction is known to be valid as soon as it is read, because such a transaction waits until it can acquire a snapshot guaranteed to be free from such problems before starting to read any data. In all other cases, applications should not rely on the results of reading data in a transaction that has not been committed. In case of error and rollback, applications must retry the transaction until it completes successfully.

To fully guarantee serializability, ADPG/PostgreSQL uses predicate locks, which are locks that allow you to determine when a write would affect the result of a previous read from a concurrent transaction if that write was executed first. These locks do not result in actual data locks and therefore cannot cause transaction deadlocks in any way. They help identify and mark dependencies between concurrent Serializable level transactions, which in certain combinations can lead to serialization anomalies. Read committed or Repeatable read transactions, on the contrary, to ensure data integrity, must either lock entire tables, which will prevent users from accessing these tables, or use `SELECT FOR UPDATE` or `SELECT FOR SHARE`, which not only can block other transactions but also create additional load on the disk.

Predicate locks are based on data actually accessed by a transaction. They are displayed in the `pg_locks` system view with a mode value of `SIReadLock`. The particular locks acquired during the execution of a query depend on the query plan, and multiple fine-grained locks (e.g., tuple locks) may be combined into fewer coarser-grained locks (e.g., page locks) to save memory spent on lock tracking.

When used correctly, serializable transactions can simplify application development. It is important that an environment, which uses this technique, has a generalized way of handling serialization failures (which always return with an `SQLSTATE` value of `'40001'`), because it will be very hard to predict exactly which transactions need to be rolled back to prevent serialization anomalies.

When using serializable transactions to manage concurrency, consider the following guidelines:

- Declare transactions as `READ ONLY` when possible.
- Control the number of active connections, using a connection pool if needed. This is always an important performance consideration, but it can be particularly important in a busy system using serializable transactions.
- Contain no more commands in a single-transaction than are necessary to ensure integrity.
- Do not leave connections dangling 'idle in transaction' longer than necessary. The configuration parameter idle_in_transaction_session_timeout can be used to automatically disconnect lingering sessions.
- Eliminate explicit locks, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` where no longer needed due to the protections provided by Serializable transactions.
- When the system is forced to combine multiple page-level predicate locks into a single relation-level predicate lock due to memory pressure, the frequency of serialization failures may increase. You can avoid this by increasing the max_pred_locks_per_transaction, max_pred_locks_per_relation, and/or max_pred_locks_per_page parameters.
- A sequential scan will always necessitate a relation-level predicate lock. This can result in an increased rate of serialization failures. It may be helpful to encourage the use of index scans by reducing random_page_cost and/or increasing cpu_tuple_cost. However, here it is important to compare the gain from reducing the number of rollbacks and restarts of transactions with the loss from possible less efficient execution of queries.

You can specify parameters mentioned above in the **ADPG configurations** section of the ADPG service settings. See Configuration parameters.

The Serializable isolation level is implemented using a technique known as **Serializable snapshot isolation**, which builds on Snapshot isolation by adding checks for serialization anomalies.

Locks

6 minutes

Contents

- Table-level locks
- Row-level Locks
- The LOCK command
- Virtx locks
- Deadlocks
- Advisory locks

Try Table of Contents

Locks are used to synchronize processes. They provide various lock modes to control concurrent access to data in tables. Most PostgreSQL commands use locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. For example, `TRUNCATE` cannot safely be executed concurrently with other operations on the same table. It obtains an `ACCESS EXCLUSIVE` lock on the table. You can also implement application-controlled locking in situations where MVCC does not suit your requirements.

Based on the lock granularity, locks are grouped, the following locks can be distinguished:

- Table-level locks – lock a table.
- Row-level locks – lock the current row.
- **Page-level locks** – page-level shared/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a row is fetched or updated. Application developers generally do not need to be concerned with page-level locks.

Locks can also be:

- **exclusive** – can only be held by one transaction at a time.
- **shared** – can be held by multiple transactions.

Table-level locks

The table below shows the available lock modes and the contexts in which AGPLPostgreSQL uses them. You can also obtain any of these locks explicitly with the `LOCK` command as described below. All of these lock modes are table-level locks, even if the name contains the word "row"; the names of the lock modes are historical.

Name	Use of the lock
ACCESS SHARE (AccessShareLock)	The `SELECT` command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.
ROW SHARE (RowShareLock)	The `SELECT FOR UPDATE` and `SELECT FOR SHARE` commands acquire a lock of this mode on the target tables (in addition to `ACCESS SHARE` locks) on any other referenced tables. In general, this lock mode will be acquired by any command that `FOR UPDATE/FOR SHARE`.
ROW EXCLUSIVE (RowExclusiveLock)	The `UPDATE`, `DELETE`, and `INSERT` commands acquire this lock mode on the target table (in addition to `ACCESS SHARE` locks) on any other referenced tables. In general, this lock mode will be acquired by any command that modifies data in a table.
SHARE UPDATE EXCLUSIVE (ShareUpdateExclusiveLock)	This mode is acquired by `VACUUM` without `FULL`, `ANALYZE`, `CREATE INDEX CONCURRENTLY`, `CREATE STATISTICS`, `COMMENT ON`, `REINDEX CONCURRENTLY`, and certain `ALTER INDEX` and `ALTER TABLE` variants (full details are in the documentation of these commands). Protects a table against concurrent data changes. It is acquired by `CREATE INDEX` without `CONCURRENTLY`.
SHARE (ShareLock)	Protects a table against concurrent data changes. It is acquired by `CREATE INDEX` without `CONCURRENTLY`.
SHARE ROW EXCLUSIVE (ShareRowExclusiveLock)	Protects a table against concurrent data changes, and it is acquired by `ALTER TABLE` for some forms.
EXCLUSIVE (ExclusiveLock)	Allows only concurrent `ACCESS SHARE` locks—only reads from the table can proceed in parallel with a transaction holding this lock. `EXCLUSIVE` locks are acquired by `REFRESH MATERIALIZED VIEW CONCURRENTLY` commands.
ACCESS EXCLUSIVE (AccessExclusiveLock)	Guarantees that the transaction holding the lock is the only transaction that can access the table in any way. This lock is also the default lock mode for `LOCK TABLE` statement. Only an `ACCESS EXCLUSIVE` lock blocks a `SELECT` (without `FOR UPDATE/SHARE`) statement.

The main difference between one lock mode and another is the set of conflicting lock modes. See the conflicting table-level locks table. Two transactions cannot hold locks of conflicting modes on the same table at the same time. In other words, two transactions cannot hold locks of non-conflicting modes. Note that some lock modes are self-conflicting. For example, an `ACCESS EXCLUSIVE` lock cannot be held by more than one transaction at a time. Other modes are not self-conflicting. For example, an `ACCESS SHARE` lock may be held by multiple transactions. In the table below, the intersection of a row and a column with conflicting transactions is marked with the "X" symbol.

Conflicting table-level locks

Requested lock mode	Existing lock mode								
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE						X	X	X
SHARE UPDATE EXCLUSIVE				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

Once acquired, a lock is normally held until the end of the transaction. But if a lock is acquired after establishing a savepoint, the lock is released immediately if a rollback to the savepoint occurs.

Row-level Locks

In addition to table-level locks, AGPLPostgreSQL uses row-level locks. The types of locks and the cases in which AGPLPostgreSQL uses them are listed in the table below.

Row-level locks

Name	Use of the lock
FOR UPDATE	`FOR UPDATE` locks the rows returned by the `SELECT` statement as for update. This prevents them from being modified or locked by other transactions. The same happens if one of the commands `UPDATE`, `DELETE`, or `SELECT FOR SHARE` is executed. The affected rows will be locked until the current transaction ends. Within a `REPEATABLE READ` or `SERIALIZABLE` transaction, an error will be thrown if a row to be locked has changed since the transaction started. The `FOR UPDATE` lock mode is also acquired by any `DELETE` command on a row, and also by `UPDATE` commands that modify the values in certain columns. Currently, the columns affected by `UPDATE` are those for which a unique index can be used in a foreign key (partial indexes and expressional indexes are not considered).
FOR NO KEY UPDATE	Behaves similarly to `FOR UPDATE`, except that the lock acquired is weaker: this lock does not block `SELECT FOR KEY SHARE` commands that attempt to acquire a lock on the same rows. This lock mode is also acquired by any `UPDATE` that does not acquire a `FOR UPDATE` lock.
FOR SHARE	Behaves similarly to `FOR NO KEY UPDATE`, but it acquires a shared lock rather than an exclusive lock on each retrieved row. A shared lock prevents other transactions from performing the `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, or `SELECT FOR NO KEY UPDATE` on these rows, but it does not prevent them from performing `SELECT FOR SHARE` or `SELECT FOR KEY SHARE`.
FOR KEY SHARE	Behaves similarly to `FOR SHARE`, except that the lock is weaker: `SELECT FOR UPDATE` is blocked, but `SELECT FOR NO KEY UPDATE` is not. The lock blocks other transactions from performing `DELETE` or any `UPDATE` that changes the key values, whether or not they are part of the same unique index. `SELECT FOR NO KEY UPDATE` `SELECT FOR SHARE` `SELECT FOR KEY SHARE`

Note that a transaction can hold conflicting locks on the same row, even in different subtransactions; but two transactions can never obtain conflicting locks of the same row. Row-level locks do not affect data querying. They block only writers and other lockers to the same row. Like table-level locks, row-level locks are released at the transaction end or during a savepoint rollback. Like exclusive locks, shared locks are also released at the transaction end or during a savepoint rollback.

Conflicting row-level locks

Requested lock mode	Existing lock 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

PostgreSQL does not store information about modified rows in memory, so there is no limit on the number of rows that can be locked. However, locking a row might cause a disk write. For example, `SELECT FOR UPDATE` modifies selected rows to mark them locked, and so results in disk writes.

The LOCK command

The `LOCK` command gets a table-level lock. When acquiring locks for commands that reference tables, PostgreSQL always uses the least restrictive lock mode possible. You can use the `LOCK` command when you need more restrictive locking. For example, suppose that your application runs a transaction at the `READ COMMITTED` isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain `SHARE` lock mode on the table before querying it. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because `SHARE` mode conflicts with the `ROW EXCLUSIVE` lock acquired by writers, and your `LOCK TABLE name IN SHARE MODE` statement will wait until any concurrent holders of `ROW EXCLUSIVE` mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes in progress; nor can any begin until you release the lock.

Conversely, if you request a row-level lock then you should ensure that the table also contains the appropriate `LOCK` statement. If you write a transaction that `SELECT FOR UPDATE` rows, you should usually `LOCK` the table in `SHARE ROW EXCLUSIVE` mode. Failure to acquire an appropriate lock could result in deadlocks: if another transaction acquires a table lock and then modifies the table, it will wait for transactions holding row-level locks to commit, and vice versa.

`LOCK` ensures if necessary for any conflicting locks to be removed. The behavior when using `NOWAIT` and `SKIP LOCKED` is specified. `LOCK` does not want to acquire the desired lock if it cannot immediately. Postgres50L aborts the command and throws an error. The following code shows how to use `NOWAIT`:

```
BEGIN;
LOCK TABLE author IN SHARE MODE NOWAIT;
INSERT INTO author_id VALUES (1, 'Bob', public_awe*) VALUES
(DEFAULT, id FROM author
WHERE name = 'Herbert Wells'),
'The Time Machine',
1895);
COMMIT;
```

```
BEGIN;
LOCK TABLE bank IN SHARE ROW EXCLUSIVE MODE NOWAIT;
SELECT FROM bank WHERE author_id =1;
COMMIT;
```

Once the lock is acquired and the task until the end of current transaction. There is no `UNLOCK` command. Locks are always released at the end of the transaction.

When the `SELECT` command receives a lock or statistics information in the view definition query are also locked resources with the same lock mode.

Note that there is no `LOCK` in the SQL standard, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions.

View locks

To examine a list of the currently outstanding locks in a database server, use the `pg_locks` system view.

Create the `accounts` table for demonstration purposes:

```
-- first session
BEGIN;
CREATE TABLE accounts(
    acc_number integer PRIMARY KEY,
    balance numeric
);
COMMIT;

INSERT INTO accounts VALUES (1234,1000.00), (5432,2000.00), (2154,3000.00);
```

It is convenient to see locks for a specific session. To do this, obtain ID of the server process attached to the current session using the `pg_backend_pid()` function (see System information functions and operators).

```
-- first session
SELECT pg_backend_pid();
```

Result:

```
 pg_backend_pid
----------------
           4155
```

Start a transaction and update a row in the table:

```
-- first session
BEGIN;
UPDATE accounts SET balance = balance + 100.00WHERE acc_number = 1234;
```

Query `pg_locks` using the session ID (`pid` = `4155`):

```
-- first session
SELECT locktype, relation::regclass, transactionid AS xid, mode, granted
    FROM pg_locks WHERE pid = 4155;
```

Result:

```
   locktype    | relation |  xid  |       mode       | granted
---------------+----------+-------+------------------+---------
 relation      | pg_locks |       | AccessShareLock  | t
 relation      | accounts_pkey |       | RowExclusiveLock | t
 relation      | accounts |       | RowExclusiveLock | t
 virtualxid    |          | 2/5   | ExclusiveLock    | t
 transactionid |          | 250496 | ExclusiveLock    | t
```

Where

- `locktype` – type of the lockable object: see View events of locktype
- `relation` – relation name obtained by casting OID from the `relation` field to the `regclass` type
- `mode` – lock mode. Its names given in the Table-level locks table
- `xid` – transaction ID
- `granted` – name the value of `t` if the lock is held, otherwise `f` (the lock is awaited).

Start the second session and a new transaction:

```
-- second session
BEGIN;
SELECT pg_backend_pid();
```

Result:

```
 pg_backend_pid
----------------
          24764
```

Try to create an index:

```
-- second session
CREATE INDEX ON accounts(acc_number);
```

Query `pg_locks` using the process ID of the second session (`24764`) from the first session:

```
-- first session
SELECT locktype, relation::regclass, transactionid AS xid, mode, granted
    FROM pg_locks WHERE pid = 24764;
```

As a result, you can see that the second session transaction waits a `SHARE` lock:

```
   locktype    | relation | xid |     mode      | granted
---------------+----------+-----+---------------+---------
 virtualxid    |          | 3/3 | ExclusiveLock | t
 relation      | accounts |     | ShareLock     | f
```

Terminate the transaction from the first session and check the result:

```
-- first session
COMMIT;
SELECT locktype, relation::regclass, transactionid AS xid, mode, granted
    FROM pg_locks WHERE pid = 24764;
```

The second session transaction acquired a `SHARE` lock to create an index:

```
   locktype    | relation | xid |     mode      | granted
---------------+----------+-----+---------------+---------
 virtualxid    |          | 3/3 | ExclusiveLock | t
 relation      | accounts |     | ShareLock     | t
 virtualxid    |          | 2/5 | ExclusiveLock | t
 transactionid |          | 250497 | ExclusiveLock | t
```

Deadlocks

Deadlocks is a situation where two (or more) transactions hold locks in such a way that they mutually block each other. Frequent use of explicit locks can increase the likelihood of deadlocks.

For example, if **transaction 1** acquires an exclusive lock on **Table A**, and then tries to acquire an exclusive lock on **Table B**, while **transaction 2** has already exclusive locked **Table B** and now wants to exclusive lock **Table A**, then neither one can proceed. AGPLPostgreSQL automatically detects deadlock situations and resolves them so that only one of the transactions can complete. The server will randomly choose one transaction to abort and roll back the other transaction so that it can proceed.

Note that deadlocks can also occur as the result of row-level locks. So, they can occur even if explicit locking is not used. Consider the case in which two concurrent transactions modify a table. The first transaction executes:

```
UPDATE accounts SET balance = balance + 1000.00 WHERE acc_number = 1234;
```

It acquires a row-level lock on the row with `acc_number` = `1234`. Then the second transaction executes:

```
UPDATE accounts SET balance = balance + 500.00 WHERE acc_number = 5432;
UPDATE accounts SET balance = balance + 100.00 WHERE acc_number = 1234;
```

The first `UPDATE` statement successfully acquires a row-level lock on the specified row (with `acc_number` = `5432`) and updates that row. However, the second `UPDATE` statement finds that the row it is attempting to update has been locked, so it waits for the transaction that acquired the lock to complete. The second transaction is now waiting for the first transaction to complete before it continues execution.

Then, the first transaction executes:

```
UPDATE accounts SET balance = balance - 1000.00 WHERE acc_number = 5432;
```

The first transaction attempts to acquire a lock on the specified row, but it cannot. The second transaction already holds such a lock. So it waits for the second transaction to complete. Thus, the first transaction is blocked on the second transaction, and the second transaction is blocked on the first. A deadlock occurs. AGPLPostgreSQL detects this situation and aborts one of the transactions.

Typically, the best way to prevent deadlocks is to ensure that all applications accessing the database request locks on multiple objects in a consistent order. In the example above, if both transactions had updated the rows in the same order, no deadlock would have occurred.

Also, locks should be ordered so that the first lock acquired on an object is the most restrictive mode. If such an order cannot be ensured in advance, then deadlocks can be handled on the fly by retrying transactions that abort due to deadlocks.

Advisory locks

AGPLPostgreSQL provides a means for creating locks that have application-defined meanings. These are called **advisory locks**. It is up to the application to use them correctly. Advisory locks can be useful for locking strategies that do not follow the MVCC model.

There are two ways to acquire an advisory lock, at the session level and the transaction level. Once a session level advisory lock is held until explicitly released or the session ends. Unlike standard lock requests, session-level advisory lock requests do not honor transaction semantics: a lock acquired in a transaction, even if the transaction is rolled back, will be retained in the session. Similarly, releasing a lock removes it effect even if the transaction in which it was performed is later aborted. A lock can be acquired multiple times by its owning process; for each completed lock request there must be a corresponding unlock request before the lock is released.

Transaction level lock requests, on the other hand, behave more like regular lock requests: they are automatically released at the end of the transaction, and there is no explicit unlock operation. This behavior often helps avoid unnecessary calls, since it may not be necessary to think about releasing locks at all. Session-level and transaction-level lock requests for the same advisory lock identifier will block each other in the expected manner. If a session already holds a given advisory lock, additional requests by it will always succeed, even if other sessions are awaiting the lock. This statement is true regardless of whether the held lock (session or transaction) was first acquired.

For example, obtain an exclusive session-level advisory lock. Functions that manipulate advisory locks use numeric identifiers. As a possible example, you can use the hash code from the name:

```
BEGIN;
SELECT pg_advisory_lock(hashtext('custom_name'));
COMMIT;

SELECT pg_advisory_unlock(hashtext('custom_name'));
```

The functions that manipulate advisory locks are described in the Advisory lock functions section.

For example you can create an advisory lock to create a function that processes a multi-user execution:

```
CREATE FUNCTION sample_ad(I NUMERIC) RETURNS VOID AS $$
BEGIN
    PERFORM pg_advisory_lock(I);
    LOOP
        EXECUTE 'UPDATE pgbench_accounts
            SET balance = balance + 1 ';  -- execute recurse commands
    END LOOP;
    PERFORM pg_advisory_unlock(I);
END;
$$ LANGUAGE plpgsql
```

When using advisory locks, especially in queries with aggregate `UNION` or `LIMIT` statements, it is important to remember that the resulting locks may depend on the order in which the SQL expressions are evaluated. In the following example, `LIMIT` will not necessarily be applied before calling the `pg_advisory_lock` function. The application may acquire locks for more rows than it needs to return, even though the final result will be limited:

```
SELECT pg_advisory_lock(q) FROM table WHERE id = 12345 LIMIT 100;
```

The following query solves this problem:

```
SELECT pg_advisory_lock(q) FROM
(
SELECT id FROM table WHERE id = 12345 LIMIT 100
) AS q;
```

Both advisory locks and regular locks are stored in a shared memory pool. Its size is defined by the configuration parameters `max_locks_per_transaction` and `max_connections`. You can set these parameters on the **Clusters** → **AGPG cluster** → **Services** → **AGPG** → **Primary configuration** tab in KONTUR UI. Use the **AGPG configurations** section to set `max_locks_per_transaction` and `max_connections` parameters. If this memory is insufficient, the server will not be able to issue new locks.

Like all locks in PostgreSQL, a complete list of advisory locks currently held by any session can be found in the `pg_locks` system view. The `locktype` field contains the value `advisory` for them.

Example of working with tables via psql

Contents

- Overview
- Step 1. Create a database
- Step 2. Create a table
- Step 3. Get information about the table
- Step 4. Insert new data into the table
- Step 5. Select the table data
  - Select all data
  - Use WHERE
  - Sort the results
  - Group the results
  - Join tables
  - Combine SELECT with CREATE and INSERT
- Step 6. Update the table data
- Step 7. Delete the table data
- Step 8. Alter the table
  - Rename the table
  - Rename the column
  - Add the index
- Step 9. Drop the table

Overview

The easiest way to work with PostgreSQL tables is to use the `psql` program. Note that it allows you to enter data queries quite directly in PostgreSQL, and also from scripts. Let's create a table with a list of users, insert some data into it, and then perform various operations on the table data. Let's assume that the PostgreSQL server is running on the local machine (`localhost`), the server port is the default one (`5432`), and the user with the superuser rights is `postgres`.

```
$ sudo su - postgres

$ psql
```

The `psql` commands are with the `\` character. All subsequent commands should be entered at the prompt of the PostgreSQL server:

```
postgres=#
\set PROMPT1 '%n@%m %~%R%# '
postgres@localhost ~=#
```

Step 1. Create a database

To create a new database, the `CREATE DATABASE` statement is used. Let's create the database `mydb`, and build the users with the permissions `john` and `jane`:

```
postgres@localhost ~=# CREATE DATABASE mydb;
CREATE DATABASE
postgres@localhost ~=# CREATE USER john WITH PASSWORD 'mypassword';
CREATE ROLE
postgres@localhost ~=# CREATE USER jane WITH PASSWORD 'mypassword';
CREATE ROLE
postgres@localhost ~=# GRANT ALL PRIVILEGES ON DATABASE mydb TO john;
GRANT
postgres@localhost ~=# GRANT ALL PRIVILEGES ON DATABASE mydb TO jane;
GRANT
postgres@localhost ~=#
```

The following keywords create the new database `mydb`, the user `john` with the password `mypassword`, and the user `jane` with the password `mypassword`. Also, the last two keywords grant the user `john` and the user `jane` all the privileges on the database `mydb`.

```
postgres@localhost ~=# \c mydb
You are now connected to database "mydb" as user "postgres".
mydb@localhost ~=#
```

The following keywords connect the new database `mydb` as the superuser `postgres`.

```
mydb@localhost ~=# \l
                                  List of databases
   Name    |  Owner   | Encoding |  Collate   |   Ctype    |   Access privileges
-----------+----------+----------+------------+------------+-----------------------
 mydb      | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 postgres  | postgres | UTF8     | en_US.utf8 | en_US.utf8 |
 template0 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
 template1 | postgres | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
           |          |          |            |            | postgres=CTc/postgres
(4 rows)

mydb@localhost ~=#
```

The following keywords show the list of the databases.

```
mydb@localhost ~=# \du
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 jane      |                                                            | {}
 john      |                                                            | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}

mydb@localhost ~=#
```

The following keywords show the list of the users and their roles.

Step 2. Create a table

To create a new table, you should use the `CREATE TABLE` statement. Let's create the table `users`, which will consist of the columns `id`, `name`, `age`, `address`, and `salary`:

```
mydb@localhost ~=# CREATE TABLE users (
mydb@localhost ~=# id INT PRIMARY KEY NOT NULL,
mydb@localhost ~=# name TEXT NOT NULL,
mydb@localhost ~=# age INT NOT NULL,
mydb@localhost ~=# address CHAR(50),
mydb@localhost ~=# salary REAL);
CREATE TABLE
mydb@localhost ~=#
```

The following keywords create the table `users`. Each column in the table is defined by its name, data type, and optional attributes. For example, the column `id` is defined as the `INT` data type, and the `PRIMARY KEY` and `NOT NULL` constraints are applied to it. The column `name` is defined as the `TEXT` data type, and the `NOT NULL` constraint is applied to it. The column `address` is defined as the `CHAR(50)` data type. The column `salary` is defined as the `REAL` data type.

```
mydb=# \dt
        List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | users | table | postgres
(1 row)

mydb=#
```

The following keywords show the list of the tables.

Step 3. Get information about the table

To get information about the table, you should use the `\d` command. Let's get the following commands:

- `\d TABLE_NAME` - to show the list of the columns about the columns of the `TABLE_NAME` table, their data types, ability to store the `NULL` values, and default data;
- `\d+ TABLE_NAME` - to show the same information as the `\d TABLE_NAME` command, and also the storage and description for the columns.

```
mydb=# \d users
                                    Table "public.users"
  Column |     Type      | Collation | Nullable |  Default
---------+---------------+-----------+----------+-----------
 id      | integer       |           | not null |
 name    | text          |           | not null |
 age     | integer       |           | not null |
 address | character(50) |           |          |
 salary  | real          |           |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)

mydb=#
```

The following keywords show the information about the columns and their data types.

```
mydb=# \d+ users
                                                    Table "public.users"
  Column |     Type      | Collation | Nullable |  Default | Storage  | Compression | Stats target | Description
---------+---------------+-----------+----------+----------+----------+-------------+--------------+-------------
 id      | integer       |           | not null |          | plain    |             |              |
 name    | text          |           | not null |          | extended |             |              |
 age     | integer       |           | not null |          | plain    |             |              |
 address | character(50) |           |          |          | extended |             |              |
 salary  | real          |           |          |          | plain    |             |              |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)

mydb=#
```

The following keywords show the information about the columns and their data types, and also their storage and description.

Step 4. Insert new data into the table

To insert new data into the table, you should use the `INSERT` statement. Let's add the table rows, using a name, address (you can also use the `NULL` value), and salary for each user. Also, we'll not specify the `id` column value, because it is the `PRIMARY KEY` column, and the PostgreSQL server will add the correct value for each row automatically:

```
mydb=# INSERT INTO users (name, address, salary) VALUES ('Ben', 'Texas', 2000);
INSERT 0 1
mydb=# INSERT INTO users (name, address, salary) VALUES ('Kate', 'Arizona', 1500);
INSERT 0 1
mydb=# INSERT INTO users (name, address, salary) VALUES ('Mike', 'Florida', 1800);
INSERT 0 1
mydb=#
```

The result of each of the commands looks like as follows:

```
INSERT 0 1
```

You can also insert the row into the `users` table, but this time, without the value of the `address` field (the `NULL` value):

```
mydb=# INSERT INTO users (name, age, salary) VALUES ('Nick', 25, 1700);
INSERT 0 1
mydb=# SELECT * FROM users;
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
  2 | Kate |  25 | Arizona                                            |   1500
  3 | Mike |  35 | Florida                                            |   1800
  4 | Nick |  25 |                                                    |   1700
(4 rows)

mydb=#
```

The result of the command:

```
INSERT 0 1
```

To see all the data in the table, you should use the `SELECT` statement. Let's get the following command:

```
mydb=# SELECT * FROM users;
```

The result of the command:

```
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
  2 | Kate |  25 | Arizona                                            |   1500
  3 | Mike |  35 | Florida                                            |   1800
  4 | Nick |  25 |                                                    |   1700
(4 rows)
```

Step 5. Select the table data

To select data from the table, you should use the `SELECT` statement. Let's select the data from the `users` table. Let's get the following command:

```
mydb=# SELECT * FROM users;
```

The result of the command:

```
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
  2 | Kate |  25 | Arizona                                            |   1500
  3 | Mike |  35 | Florida                                            |   1800
  4 | Nick |  25 |                                                    |   1700
(4 rows)

mydb=#
```

Select all data

The following query will select all the data from the `users` table:

```
SELECT * FROM users;
```

Here are the `id` and `name` columns from the `users` table:

```
SELECT id, name FROM users;
```

Use WHERE

The next query selects rows of the table having salary is `2000`:

```
SELECT *
FROM users
WHERE salary = 2000;
```

The result:

```
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
(1 row)
```

Sort the results

The following query selects the data and partially sorts it by the values stored in the `name` column:

```
SELECT *
FROM users
ORDER BY name;
```

The result of partial sort:

```
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
  3 | Mike |  35 | Florida                                            |   1800
  4 | Nick |  25 |                                                    |   1700
  2 | Kate |  25 | Arizona                                            |   1500
(4 rows)
```

Group the results

The next query groups all records, where the `name` field has the duplicated values and the count of the found duplicated items (per name). Sorting by the sum performed, descending:

```
SELECT
    name,
    SUM(salary)
FROM users
GROUP BY name
ORDER BY SUM(salary) DESC;
```

The result of the query:

```
 name | sum
------+------
 Ben  | 2000
 Mike | 1800
 Nick | 1700
 Kate | 1500
(4 rows)
```

Join tables

The next example joins two tables: `customer` and `order`. In order to display products names instead of their identifiers stored in the table `order`:

```
SELECT
    order_id,
    customer.name AS customer,
    product.name AS product
FROM order
INNER JOIN customer ON order.customer_id = customer.id
INNER JOIN product ON order.product_id = product.id;
```

The result:

```
 order_id | customer | product
----------+----------+---------
        1 | Ben      | Phone
        2 | Kate     | Laptop
        3 | Mike     | Monitor
        4 | Nick     | Phone
        5 | Ben      | Laptop
(5 rows)
```

Combine SELECT with CREATE and INSERT

The following query creates the table `customer_backup` and copies to it the data from the table `customer` with `SELECT` and `INSERT` commands:

```
CREATE TABLE customer_backup AS SELECT * FROM customer;
INSERT INTO customer_backup SELECT * FROM customer;
```


NOTICE

Sometimes you may need to create a new table from the existing one. If you need to, you can use the `SELECT INTO` command and then the `CREATE TABLE` command.


```
CREATE TABLE temp_copy
AS SELECT * FROM users WHERE id = 1;
```

The result:

```
SELECT 1
```

The next query selects the rows in the table `temp_copy` and then drops the table `temp_copy` (removes it):

```
SELECT * FROM temp_copy;
DROP TABLE temp_copy;
```

The result:

```
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
(1 row)

DROP TABLE
```

You can check that the table will be the following command. The content of the table is identical:

```
mydb=# SELECT * FROM users_copy;
 id |  name  | age |                      address                       | salary
----+--------+-----+----------------------------------------------------+--------
  1 | Ben    |  30 | Texas                                              |   2000
  2 | Kate   |  25 | Arizona                                            |   1500
  3 | Mike   |  35 | Florida                                            |   1800
  4 | Nick   |  25 |                                                    |   1700
  5 | Ben    |  30 | Texas                                              |   2000
  6 | Kate   |  25 | Arizona                                            |   1500
  7 | Mike   |  35 | Florida                                            |   1800
  8 | Nick   |  25 |                                                    |   1700
(8 rows)

mydb=# SELECT * FROM customer_backup;
 id |  name  | age |                      address                       | salary
----+--------+-----+----------------------------------------------------+--------
  1 | Ben    |  30 | Texas                                              |   2000
  2 | Kate   |  25 | Arizona                                            |   1500
  3 | Mike   |  35 | Florida                                            |   1800
  4 | Nick   |  25 |                                                    |   1700
  5 | Ben    |  30 | Texas                                              |   2000
  6 | Kate   |  25 | Arizona                                            |   1500
  7 | Mike   |  35 | Florida                                            |   1800
  8 | Nick   |  25 |                                                    |   1700
(8 rows)

mydb=#
```

Step 6. Update the table data

To update the data in the table, you should use the `UPDATE` statement. Also, you should use the `SET` statement, that sets the new values for the columns of the `SELECT` rows, and `WHERE` statement, that specifies the rows for updating. Let's update the data for the table `users`, where the value of the `id` column is `3`:

```
mydb=# UPDATE users SET salary = 1000 WHERE id = 3;
UPDATE 1
mydb=#
```

The following keywords update the value of the `salary` field. The system returns a successful operation.

```
mydb=# SELECT * FROM users;
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
  2 | Kate |  25 | Arizona                                            |   1500
  4 | Nick |  25 |                                                    |   1700
  3 | Mike |  35 | Florida                                            |   1000
(4 rows)

mydb=#
```

The following keywords show the updated data.

NOTICE

Be careful with using the `UPDATE` statement. Missing the `WHERE` clause can lead to all rows in the table being updated.

Step 7. Delete the table data

To delete the data in the table, you should use the `DELETE` statement. Also, you should use the `WHERE` statement, that specifies the rows for deletion. Let's delete the row, where the value of the `id` column is `3`:

```
mydb=# DELETE FROM users WHERE id = 3;
DELETE 1
mydb=#
```

The following keywords delete the row. The system returns a successful operation.

```
mydb=# SELECT * FROM users;
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
  2 | Kate |  25 | Arizona                                            |   1500
  4 | Nick |  25 |                                                    |   1700
(3 rows)

mydb=#
```

The following keywords show the updated data.

NOTICE

Be careful with using the `DELETE` statement. Missing the `WHERE` clause can lead to all rows in the table being deleted.

Step 8. Alter the table

To change the table name, you should use the `ALTER TABLE` statement. Also, you should use the `RENAME` and `TO` keywords. Changes using one of the possible schema, and both the old and new table names. Let's rename the table `users` to `my_users`. Other data in the table will not change:

Rename the table

The next command renames the table `users` to `my_users`:

```
ALTER TABLE users RENAME TO my_users;
```

The result is as follows:

```
ALTER TABLE
```

To check the command result, you can run the `\dt` command. The table is renamed:

```
mydb=# \dt
        List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | my_users | table | postgres
(1 row)

mydb=#
```

To check the command result, you can run the `SELECT` statement. The table is renamed:

```
mydb=# SELECT * FROM my_users;
 id | name | age |                      address                       | salary
----+------+-----+----------------------------------------------------+--------
  1 | Ben  |  30 | Texas                                              |   2000
  2 | Kate |  25 | Arizona                                            |   1500
  4 | Nick |  25 |                                                    |   1700
(3 rows)

mydb=#
```

Rename the column

The next command renames the column `name` to `full_name`:

```
ALTER TABLE my_users RENAME name TO full_name;
```

The result is as follows:

```
ALTER TABLE
```

To check the command result, you can run the `SELECT` statement. The column is renamed:

```
mydb=# SELECT * FROM my_users;
 id | full_name | age |                      address                       | salary
----+-----------+-----+----------------------------------------------------+--------
  1 | Ben       |  30 | Texas                                              |   2000
  2 | Kate      |  25 | Arizona                                            |   1500
  4 | Nick      |  25 |                                                    |   1700
(3 rows)

mydb=#
```

Add the index

The next command adds the index `salary_index` to the column `salary`. The index is created in the table `my_users` and `ASC` and `DESC` clauses specify the sort order:

```
ALTER TABLE my_users
ADD CONSTRAINT salary_index UNIQUE (salary ASC, name);
```

The result is:

```
ALTER TABLE
```

To check the command result, you can run the `\d` command. The index is added:

```
mydb=# \d my_users
                                    Table "public.my_users"
  Column   |     Type      | Collation | Nullable |  Default
-----------+---------------+-----------+----------+-----------
 id        | integer       |           | not null |
 full_name | text          |           | not null |
 age       | integer       |           | not null |
 address   | character(50) |           |          |
 salary    | real          |           |          |
Indexes:
    "my_users_pkey" PRIMARY KEY, btree (id)
    "salary_index" UNIQUE CONSTRAINT, btree (salary, name)

mydb=#
```

Step 9. Drop the table

To remove the table permanently, you should use the `DROP TABLE` statement, that will drop the table data and the table itself. Let's remove the `users` table:

```
DROP TABLE users;
```

The result of execution of this is:

```
DROP TABLE
```

To check the command results, you can run the `\dt` command. The table is removed:

```
mydb=# \dt
        List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | my_users | table | postgres
(1 row)

mydb=#
```

Use tables and column aliases in queries

Evgeniia Kuzina

AD.PG

Contents

To Table of Contents

Table aliases

Column aliases

You can specify a temporary name for a table or a column. This name is called an **alias**. PostgreSQL assigns aliases during query execution and does not store them in a database or on a disk.

Create a table for examples:

```
CREATE TABLE employees (
employee_id serial PRIMARY KEY,
full_name VARCHAR NOT NULL,
hours INT,
rate  INT,
manager_id INT
);
```

Insert data to the table:

```
INSERT INTO employees (employee_id, full_name, hours, rate, manager_id)
  VALUES
  (1, 'James Wilson', 8, 500, NULL),
  (2, 'Mary Burton', 6, 450, 1),
  (3, 'Patricia Robinson', 7, 430, 1),
  (4, 'Robert Gray', 8, 480, 1),
  (5, 'Elizabeth Tucker', 7, 470, 2),
  (6, 'Joseph Lewis', 7, 470, 2),
  (7, 'William Ferguson', 7, 470, 2),
  (8, 'Linda Black', 6, 320, 3),
  (9, 'David Green', 6, 310, 3),
  (10, 'Daniel Gray', 5, 390, 5),
  (11, 'Mark Armstrong', 6, 320, 4),
  (12, 'Donald Carter', 9, 290, 7),
  (13, 'Elizabeth Collins', 8, 300, 7),
  (14, 'Paul Brown', 8, 280, 8),
  (15, 'Andrew Clarke', 8, 280, 8);
```

The result:

```
SELECT * FROM employees;

 employee_id |     full_name      | hours | rate | manager_id
-------------+--------------------+-------+------+------------
           1 | James Wilson       |     8 |  500 |
           2 | Mary Burton        |     6 |  450 |          1
           3 | Patricia Robinson  |     7 |  430 |          1
           4 | Robert Gray        |     8 |  480 |          1
           5 | Elizabeth Tucker   |     7 |  470 |          2
           6 | Joseph Lewis       |     7 |  470 |          2
           7 | William Ferguson   |     7 |  470 |          2
           8 | Linda Black        |     6 |  320 |          3
           9 | David Green        |     6 |  310 |          3
          10 | Daniel Gray        |     5 |  390 |          5
          11 | Mark Armstrong     |     6 |  320 |          4
          12 | Donald Carter      |     9 |  290 |          7
          13 | Elizabeth Collins  |     8 |  300 |          7
          14 | Paul Brown         |     8 |  280 |          8
          15 | Andrew Clarke      |     8 |  280 |          8
```

Table aliases

You can assign an alias to a table or complex table reference and use this alias in the current query. A table reference can be a derived table, for example, a subquery, JOIN construct, or complex combination of these options.

Use the following syntax to assign an alias:

```
FROM <table_reference> AS <alias>
```

or

```
FROM <table_reference> <alias>
```

Where:

- `<table_reference>` is a table name or complex table reference;
- `<alias>` is a unique identifier in the current query.

The `AS` keyword is optional.

Aliases are often used to assign short identifiers to long table names to improve the readability of queries. For example:

```
SELECT * FROM very_long_table_name1 a JOIN very_long_table_name2 b ON a.id = b.num_id;
```

The alias becomes a new table name in the current query. If the alias is assigned, the original table name cannot be used in the query. For example, the following query is invalid:

```
-- an incorrect query

SELECT * FROM table1 AS t WHERE table1.id > 276;
```

Although aliases are mostly used for convenience, they are necessary when a table is joined to itself, for example:

```
SELECT manager.employee_id, manager.full_name,
   subordinate.employee_id, subordinate.full_name, subordinate.manager_id
FROM employees AS manager
JOIN employees AS subordinate
ON manager.employee_id = subordinate.manager_id;
```

The result:

```
employee_id |     full_name      | employee_id |     full_name      | manager_id
------------+--------------------+-------------+--------------------+------------
          1 | James Wilson       |           2 | Mary Burton        |          1
          1 | James Wilson       |           3 | Patricia Robinson  |          1
          1 | James Wilson       |           4 | Robert Gray        |          1
          2 | Mary Burton        |           5 | Elizabeth Tucker   |          2
          2 | Mary Burton        |           6 | Joseph Lewis       |          2
          2 | Mary Burton        |           7 | William Ferguson   |          2
          3 | Patricia Robinson  |           8 | Linda Black        |          3
          3 | Patricia Robinson  |           9 | David Green        |          3
          5 | Elizabeth Tucker   |          10 | Daniel Gray        |          5
          4 | Robert Gray        |          11 | Mark Armstrong     |          4
          7 | William Ferguson   |          12 | Donald Carter      |          7
          7 | William Ferguson   |          13 | Elizabeth Collins  |          7
          8 | Linda Black        |          14 | Paul Brown         |          8
          8 | Linda Black        |          15 | Andrew Clarke      |          8
```

An alias is also required if the table reference is a subquery. See Subquery in the FROM clause for details.

You can use parentheses to resolve ambiguities. In the following example, the first statement assigns the `staff` alias to the second `employees` instance, but the second statement assigns the `staff` alias to the result of `CROSS JOIN`:

```
SELECT * FROM employees AS managers CROSS JOIN employees AS staff ...
SELECT * FROM (employees AS managers CROSS JOIN employees) AS staff ...
```

Also, aliases are not available outside parentheses. For example, the following query is valid:

```
SELECT a.* FROM table1 AS a JOIN table2 AS b ON a.id = b.num_id;
```

However, the `a` alias is not visible outside parentheses in the query below:

```
-- an incorrect query

SELECT a.* FROM (table1 AS a JOIN table2 AS b ON a.id = b.num_id) AS c;
```

Column aliases

A column alias allows you to assign a temporary name to a column or an expression in the `SELECT` statement list. The column alias exists only during the query execution.

Aliases usually consist of lowercase letters. If you need to use uppercase letters, special characters, spaces, or keywords, enclose an alias in double quotes `"`.

Use the following syntax to assign an alias:

```
SELECT <column_name> AS <alias> FROM <table>;
```

or

```
SELECT <column_name> <alias> FROM <table>;
```

Where:

- `<column_name>` is a name of a column;
- `<alias>` is a unique identifier for a column;
- `<table>` is a table name.

The `AS` keyword is optional.

Example:

```
SELECT employee_id AS id, full_name AS "full name" FROM employees LIMIT 3;
```

The result:

```
 id |     full name
----+--------------------
  1 | James Wilson
  2 | Mary Burton
  3 | Patricia Robinson
```

Aliases are displayed in the column headers.

You can set an alias for an expression:

```
SELECT full_name "full name", hours*rate sum FROM employees LIMIT 3;
```

The result:

```
     full name      | sum
--------------------+------
 James Wilson       | 4000
 Mary Burton        | 2700
 Patricia Robinson | 3010
```

Alternatively, you can use the following syntax to assign aliases to columns and a table simultaneously:

```
FROM <table_reference> [AS] <alias> ( <column1_alias> [, <column2_alias> [, ...]] )
```

Where:

- `<table_reference>` is a table name or complex table reference;
- `<alias>` is a table alias;
- `<column1..n_alias>` are column aliases.

If you specify fewer column aliases than the number of columns in the table, the remaining columns retain their original names.

Example:

```
SELECT * FROM employees AS managers ( id, "full name") LIMIT 3;
```

The result:

```
 id |     full name      | hours | rate | manager_id
----+--------------------+-------+------+------------
  1 | James Wilson       |     8 |  500 |
  2 | Mary Burton        |     6 |  450 |          1
  3 | Patricia Robinson |     7 |  430 |          1
```

JOIN clause

Contents

To Table of Contents

- CROSS JOIN
- INNER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN
- JOIN conditions
  - ON
  - USING
  - NATURAL
- Combine JOIN clauses

The `JOIN` clause combines data from two tables according to the rules of the specified `JOIN` type. The general `JOIN` syntax is:

```
<table> <JOIN_type> <table2> [ <join_condition> ]
```

The syntax of `JOIN` clauses that process string matches is:

```
<table1> { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN <table2> ON <boolean_expression>
<table1> { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN <table2> USING ( <join_column_list> )
<table1> NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN <table2>
```

`INNER` and `OUTER` are optional words in all forms. The default value is `INNER`. However, if the `LEFT`, `RIGHT`, or `FULL` modifier is specified without `INNER` or `OUTER`, PostgreSQL executes the `JOIN` clause as `OUTER JOIN`.

The `ON` and `USING` clauses control the evaluation of the `JOIN` condition. Also, the `NATURAL` keyword implicitly sets the `JOIN` condition. The condition determines, which rows of the two source tables corresponds to each other.

To see how different types of `JOIN` operate with data, create two tables:

```
CREATE TABLE books (
    id SERIAL PRIMARY KEY,
    title VARCHAR (256),
    cover_id INT
);

INSERT INTO books (title, cover_id) VALUES
('Hyperion', 1),
('1984', 2),
('Dune', 3),
('The Time Machine', 4),
('Mrs. Dalloway', 5);

CREATE TABLE covers (
    id SERIAL PRIMARY KEY,
    name VARCHAR (32)
);

INSERT INTO covers (name) VALUES
('Paperback'),
('Hardcover'),
('Hardcover with dust jacket'),
('Leatherette cover');
```

The `books` table contains the `cover_id` field. This field is a foreign key that refers to the primary key in the `covers` table and serves for defining relationships between tables.

CROSS JOIN

`CROSS JOIN` resulted table consists of all possible combinations of rows from `table1` and `table2` (the Cartesian product). The set of the joined table columns includes all `table1` columns followed by all `table2` columns. If the tables have **N** and **M** rows respectively, the joined table has **N** * **M** rows. `CROSS JOIN` is equivalent to `INNER table1 INNER JOIN table2 ON TRUE`.

Example:

```
SELECT * FROM books CROSS JOIN covers;
```

The result:

```
 id |      title       | cover_id | id |              name
----+------------------+----------+----+--------------------------------
  1 | Hyperion         |        1 |  1 | Paperback
  2 | 1984             |        2 |  1 | Paperback
  3 | Dune             |        3 |  1 | Paperback
  4 | The Time Machine |        4 |  1 | Paperback
  5 | Mrs. Dalloway    |        5 |  1 | Paperback
  1 | Hyperion         |        1 |  2 | Hardcover
  2 | 1984             |        2 |  2 | Hardcover
  3 | Dune             |        3 |  2 | Hardcover
  4 | The Time Machine |        4 |  2 | Hardcover
  5 | Mrs. Dalloway    |        5 |  2 | Hardcover
  1 | Hyperion         |        1 |  3 | Hardcover with dust jacket
  2 | 1984             |        2 |  3 | Hardcover with dust jacket
  3 | Dune             |        3 |  3 | Hardcover with dust jacket
  4 | The Time Machine |        4 |  3 | Hardcover with dust jacket
  5 | Mrs. Dalloway    |        5 |  3 | Hardcover with dust jacket
  1 | Hyperion         |        1 |  4 | Leatherette cover
  2 | 1984             |        2 |  4 | Leatherette cover
  3 | Dune             |        3 |  4 | Leatherette cover
  4 | The Time Machine |        4 |  4 | Leatherette cover
  5 | Mrs. Dalloway    |        5 |  4 | Leatherette cover
(20 rows)
```

INNER JOIN

The `INNER JOIN` examines each row in `table1` and checks if `table2` contains a row that satisfies the `JOIN` condition. If a corresponding row is found, the `INNER JOIN` creates a new row that combines columns from both tables and adds it to the result set.

Example:

```
SELECT books.id, books.title, covers.name cover FROM books
  INNER JOIN covers ON books.cover_id = covers.id;
```

The result contains books `id`, `title` and `cover` type. The `Mrs. Dalloway` book is not included in the result since for its `cover_id` value (`5`) a corresponding row in `covers` is not found.

```
 id |      title       |            cover
----+------------------+-----------------------------
  1 | Hyperion         | Paperback
  2 | 1984             | Hardcover
  3 | Dune             | Hardcover with dust jacket
  4 | The Time Machine | Hardcover with dust jacket
```

LEFT OUTER JOIN

The `LEFT OUTER JOIN` clause performs the `INNER JOIN` operation first. After that, each `table1` row, which is excluded from the `INNER JOIN` result, is added to the `LEFT OUTER JOIN` result with null values in the `table2` columns. The joined table always has at least one row for each `table1` row.

Example:

```
SELECT books.id, books.title, covers.name cover FROM books
  LEFT OUTER JOIN covers ON books.cover_id = covers.id;
```

The result contains a row with the `Mrs. Dalloway` book, in which the `covers.name` column with the `cover` alias has the null value:

```
 id |      title       |            cover
----+------------------+-----------------------------
  1 | Hyperion         | Paperback
  2 | 1984             | Hardcover
  3 | Dune             | Hardcover with dust jacket
  4 | The Time Machine | Hardcover with dust jacket
  5 | Mrs. Dalloway    |
```

RIGHT OUTER JOIN

The `RIGHT OUTER JOIN` clause performs the `INNER JOIN` operation first. After that, each `table2` row, which is excluded from the `INNER JOIN` result, is added to the `RIGHT OUTER JOIN` result with null values in the `table1` columns. The joined table always has one row for each `table2` row.

Example:

```
SELECT books.id, books.title, covers.name cover FROM books
  RIGHT OUTER JOIN covers ON books.cover_id = covers.id;
```

The result contains a row with `Leatherette cover`, in which the `books.id` and `books.title` columns have null values:

```
 id |      title       |            cover
----+------------------+-----------------------------
  1 | Hyperion         | Paperback
  2 | 1984             | Hardcover
  3 | Dune             | Hardcover with dust jacket
  4 | The Time Machine | Hardcover with dust jacket
    |                  | Leatherette cover
```

FULL OUTER JOIN

The `FULL OUTER JOIN` clause performs the `INNER JOIN` operation first. After that, each `table1` row, which is excluded from the `INNER JOIN` result, is added to the `FULL OUTER JOIN` result with null values in the `table2` columns. Then each `table2` row, which is excluded from the `INNER JOIN` result, is added to the `FULL OUTER JOIN` result with null values in the `table1` columns.

Example:

```
SELECT books.id, books.title, covers.name cover FROM books
  FULL OUTER JOIN covers ON books.cover_id = covers.id;
```

The result:

```
 id |      title       |            cover
----+------------------+-----------------------------
  1 | Hyperion         | Paperback
  2 | 1984             | Hardcover
  3 | Dune             | Hardcover with dust jacket
  4 | The Time Machine | Hardcover with dust jacket
  5 | Mrs. Dalloway    |
    |                  | Leatherette cover
```

JOIN conditions

The `JOIN` condition determines, which rows from two source tables match. You can use the `ON`, `USING`, or `NATURAL` clause to specify the condition.

ON

The `ON` clause enters a boolean value. A row from `table1` matches a row from `table2` if the `ON` expression evaluates to `true`.

Create two tables for test examples:

```
CREATE TABLE table1(
    num INT PRIMARY KEY,
    name VARCHAR (20)
);

INSERT INTO table1 (num, name) VALUES
    (1, 'a'),
    (2, 'b'),
    (3, 'c');

CREATE TABLE table2(
    num INT PRIMARY KEY,
    value VARCHAR (20)
);

INSERT INTO table2 (num, value) VALUES
    (1, 'value1'),
    (2, 'value2'),
    (3, 'value3');
```

`table1`:

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

`table2`:

```
 num | value
-----+--------
   1 | value1
   2 | value2
   3 | value3
```

Execute the following query with the `ON` condition:

```
SELECT * FROM table1 INNER JOIN table2 ON table1.num = table2.num;
```

The result:

```
 num | name | num | value
-----+------+-----+--------
   1 | a    |   1 | value1
   2 | b    |   2 | value2
   3 | c    |   3 | value3
```

The `JOIN` condition can also contain additional clauses that do not relate directly to `JOIN`.

Example:

```
SELECT * FROM table1 INNER JOIN table2
  ON table1.num = table2.num AND table2.value = 'value1';
```

The result:

```
 num | name | num | value
-----+------+-----+--------
   1 | a    |   1 | value1
```

USING

You can utilize the `USING` clause, when columns in both tables have the same names. `USING` takes a comma-separated list of the column names and creates a `JOIN` condition that includes an equal comparison for each column. For example, joining `table1` and `table2` with `USING (field1, field2)` produces the `JOIN` condition `ON table1.field1 = table2.field1 AND table1.field2 = table2.field2`.

The result of the `JOIN` clause with `USING` contains a single column for each of the listed column pairs in the listed order, followed by all remaining columns from `table1` and `table2`.

Example:

```
SELECT * FROM table1 INNER JOIN table2 USING (num);
```

The result:

```
 num | name | value
-----+------+--------
   1 | a    | value1
   2 | b    | value2
   3 | c    | value3
```

NATURAL

The `NATURAL` clause creates a `USING` list. It includes all matching column names that both input tables contain. As with `USING`, the result of the `JOIN` clause with `NATURAL` contains a single column for each of the listed column pairs in the listed order, followed by all remaining columns from `table1` and `table2`.

Example:

```
SELECT * FROM table1 NATURAL INNER JOIN table2;
```

The result:

```
 num | name | value
-----+------+--------
   1 | a    | value1
   2 | b    | value2
   3 | c    | value3
```

If tables do not have columns with matching names, the `NATURAL JOIN` clause produces the same result as `CROSS JOIN`.

Combine JOIN clauses

You can chain `JOIN` clauses of all types or nest `JOIN` to another `JOIN`. In the default order, `JOIN` clauses are executed from left to right. You can enclose `JOIN` clauses in parentheses to override the join order.

Create `table3` in addition to `table1` and `table2` for test examples:

```
CREATE TABLE table3(
    num INT PRIMARY KEY,
    name VARCHAR (20)
);

INSERT INTO table3 (id, value) VALUES
    (1, 'value1'),
    (2, 'value3');
```

`table1`:

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

`table2`:

```
 num | value
-----+--------
   1 | value1
   2 | value2
   3 | value3
```

`table3`:

```
 id | value
----+--------
  1 | value1
  2 | value3
```

Execute a `JOIN` query without and with parentheses and compare results. Run a query without parentheses:

```
SELECT * FROM table1 CROSS JOIN table2 INNER JOIN table3 USING(value);
```

First, PostgreSQL executes the `CROSS JOIN` clause: `SELECT * FROM table1 CROSS JOIN table2`. Its result is:

```
 num | name | num | value
-----+------+-----+--------
   1 | a    |   1 | value1
   1 | a    |   2 | value2
   1 | a    |   3 | value3
   2 | b    |   1 | value1
   2 | b    |   2 | value2
   2 | b    |   3 | value3
   3 | c    |   1 | value1
   3 | c    |   2 | value2
   3 | c    |   3 | value3
```

After that, PostgreSQL executes `INNER JOIN` on the `CROSS JOIN` resulted table and `table3`. Since `USING` contains the `value` field, this field is the first column in the resulted table.

The result:

```
 value | num | name | num | id
-------+-----+------+-----+----
 value1 |   1 | a    |   1 |  1
 value3 |   2 | b    |   3 |  2
 value1 |   3 | c    |   1 |  1
```

Add parentheses to the query:

```
SELECT * FROM table1 CROSS JOIN (table2 INNER JOIN table3 USING(value));
```

First, PostgreSQL executes the `INNER JOIN` clause: `SELECT * FROM table2 INNER JOIN table3 USING(value)`. Its result is:

```
 value | num | id
-------+-----+----
 value1 |   1 |  1
```

After that, PostgreSQL executes `CROSS JOIN` on the `table1` and the `INNER JOIN` resulted table.

The result:

```
 num | name | value | num | id
-----+------+-------+-----+----
   1 | a    | value1 |   1 |  1
   2 | b    | value1 |   1 |  1
   3 | c    | value1 |   1 |  1
```

Subqueries

A PostgreSQL Tutorial

Contents

Overview

Scalar subquery

Subquery with comparison operators

Subquery with EXISTS

Subquery with IN

Subquery with NOT IN

Subquery with ANY/SOME

Subquery with ALL

Single-row comparison

Subquery in the FROM clause

Overview

A PostgreSQL subquery is an SQL query nested inside a larger query. A subquery can be nested inside a `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `SET`, or `DO` statement or inside another subquery. A subquery is also called an `inner query` or `inner select`, while a statement containing a subquery is also called an `outer query` or `outer select`. A subquery can also use any of the comparison operators, such as `=`, `>`, `<`, or others or multiple-row operators, such as `IN`, `ANY`, or `ALL`. A subquery is also known as a `nested query`.

For example, we have the following table `books`:

```
 id |              title              | author_id | public_year |  genre   | evaluation | price | total_sales
----+---------------------------------+-----------+-------------+----------+------------+-------+-------------
  1 | Mrs. Dalloway                   |         1 |        1925 | novel    |        7.83 |  8.68 |     4215000
  2 | To the Lighthouse               |         1 |        1927 | novel    |        8.08 |  7.99 |     3860000
  3 | To Kill a Mockingbird           |         2 |        1960 | novel    |        9.18 |  7.50 |     7210000
  4 | The Great Gatsby                |         3 |        1925 | novel    |        7.23 |  7.90 |     5110000
  5 | The Lord of the Rings           |         4 |        1954 | fantasy  |        9.49 |  9.89 |     5672000
  6 | The Hobbit, or There and Back Again |         4 |        1937 | fantasy  |        9.32 |  5.99 |     3567000
  7 | Bat and Peace                   |         5 |        1869 | novel    |        9.08 |  5.99 |     5354000
  8 | Hyperion                        |         6 |        1990 | sci-fi   |        9.44 |  8.69 |     4412200
  9 | The Time Machine                |         7 |        1895 | sci-fi   |        8.11 |  4.99 |     4444000
```

You can use subqueries with the clauses of the same game as a term of the form with `id = 1`. A subquery `SELECT name FROM books WHERE id = 1` determines the game.

```
SELECT id, title, genre
FROM books
WHERE genre = (SELECT genre FROM books WHERE id = 1);
```

The result:

```
 id |              title              | genre
----+---------------------------------+----------
  1 | Mrs. Dalloway                   | novel
  2 | To the Lighthouse               | novel
  3 | To Kill a Mockingbird           | novel
  4 | The Great Gatsby                | novel
  7 | Bat and Peace                   | novel
```

A subquery is executed before its parent query, and the subquery result is passed to the parent query. In this example, the subquery returns `'novel'`, and the parent query returns books of the `'novel'` genre. The subquery from this example is a scalar one.

Scalar subquery

A scalar subquery is a subquery that returns only one column or expression and returns one row, a single scalar value.

In the example below, the subquery `SELECT MAX(total) FROM orders WHERE customer_id = 17` returns the maximum value of the `total` field for the customer with `id = 17`.

```
INSERT INTO order_statistic (customer_id, max_sum) VALUES (
    17,
    (SELECT MAX(total) FROM orders WHERE customer_id = 17)
)
RETURNING *;
```

The result:

```
 customer_id | max_sum
-------------+---------
          17 |  5000.00
```

Subquery with comparison operators

You can use subquery before or after comparison operators. The type of subqueries can return at most one value. The value can be the result of an arithmetic expression or a function. PostgreSQL compares the subquery result value with the value on the other side of the comparison operator. You can use the following comparison operators:

Operator	Description
`=`	Equal to
`>`	Greater than
`>=`	Greater than or equal to
`<`	Less than
`<=`	Less than or equal to
`<>`	Not equal to
`!=`	Not equal to. Not safe enough for. This operator performs an equality comparison, but returns `TRUE` if one of the operands is `NULL` and both operands are `NULL` and `NULL` rather than `NULL` if one operand is `NULL`.

The following query displays books with the price higher than the average price:

```
SELECT id, title, price
FROM books
WHERE price >
(SELECT AVG(price) FROM books);
```

The result:

```
 id |              title              | price
----+---------------------------------+-------
  4 | The Great Gatsby                |  7.90
  5 | The Lord of the Rings           |  9.89
  6 | The Hobbit, or There and Back Again |  5.99
  7 | Bat and Peace                   |  5.99
```

Subquery with EXISTS

The `EXISTS` expression syntax:

```
EXISTS (subquery)
```

The `EXISTS` argument is a subquery as a `SELECT` statement. The subquery is executed to determine whether it returns any rows. If it returns at least one row, the result of `EXISTS` is `true`. If subquery returns no rows, the result of `EXISTS` is `false`.

The subquery can refer to variables from the parent query.

The subquery is executed until at least one row is found. Since the result depends on whether any rows are returned, the output list is not significant. A common coding convention is to write all `EXISTS` tests in the form `EXISTS (SELECT 1 WHERE ...)`.

For example, we have the table `authors`. A written row looks `books` mentioned above:

```
 id |      name      |   country
----+----------------+-------------
  1 | Margot Lee     | USA
  2 | F. Scott Fitzgerald | USA
  3 | J.R.R. Tolkien | Great Britain
  4 | Herbert Wells  | Great Britain
  5 | Leo Simeon     | Russia
  6 | Herbert Wells  | USA
```

Execute the following query:

```
SELECT id, title, author_id
FROM books b
WHERE EXISTS (SELECT 1 FROM authors WHERE id = b.author_id);
```

The result contains books whose authors are in the `authors` table:

```
 id |              title              | author_id
----+---------------------------------+-----------
  1 | The Great Gatsby                |         3
  2 | Bat and Peace                   |         5
  3 | The Hobbit, or There and Back Again |         4
  4 | The Lord of the Rings           |         4
  5 | To Kill a Mockingbird           |         2
  6 | To Kill a Mockingbird           |         2
  7 | Hyperion                        |         6
  8 | The Time Machine                |         7
```

Subquery with IN

The `IN` expression syntax:

```
row_expression IN (subquery)
```

`row_expression` must return the same number of columns as the `SELECT` statement. The subquery is executed to determine whether it returns any rows. If it returns at least one row, the result of `IN` is `true`. If any equal subquery row is found, the result is `TRUE`, if no equal row is found, an `IS NULL` subquery returns `NULL`, if one row is found, an `IS NULL` subquery returns `FALSE`.

The subquery can refer to variables from the parent query.

`row_expression` and `subquery` must return comparable data types. Similar to `UNION`, `INTERSECT`, and `EXCEPT`, implicit type conversions are attempted to make unlike types comparable.

The subquery is executed until a row is found. Since the result depends on whether any rows are returned, the output list is not significant. A common coding convention is to write all `IN` tests in the form `EXISTS (SELECT 1 WHERE ...)`.

For example, we have the table `authors`. A written row looks `books` mentioned above:

```
 id |      name      |   country
----+----------------+-------------
  1 | Margot Lee     | USA
  2 | F. Scott Fitzgerald | USA
  3 | J.R.R. Tolkien | Great Britain
  4 | Herbert Wells  | Great Britain
  5 | Leo Simeon     | Russia
  6 | Herbert Wells  | USA
```

Execute the following query:

```
SELECT id, title, author_id
FROM books b
WHERE EXISTS (SELECT 1 FROM authors WHERE id = b.author_id);
```

The result contains books whose authors are in the `authors` table:

```
 id |              title              | author_id
----+---------------------------------+-----------
  1 | The Great Gatsby                |         3
  2 | Bat and Peace                   |         5
  3 | The Hobbit, or There and Back Again |         4
  4 | The Lord of the Rings           |         4
  5 | To Kill a Mockingbird           |         2
  6 | To Kill a Mockingbird           |         2
  7 | Hyperion                        |         6
  8 | The Time Machine                |         7
```

Subquery with IN

The `IN` expression syntax:

```
row_expression IN (subquery)
```

`row_expression` must return the same number of columns as the `row_constructor` expression contains.

Example

```
SELECT id, title, author_id
FROM books
WHERE author_id IN
(SELECT id FROM authors WHERE country='USA');
```

The result:

```
 id |              title              | author_id
----+---------------------------------+-----------
  1 | To Kill a Mockingbird           |         2
  2 | The Hobbit, or There and Back Again |         4
  3 | The Lord of the Rings           |         4
  4 | Hyperion                        |         6
  5 | The Time Machine                |         7
```

The `IN` is equivalent to `= ANY`.

Subquery with NOT IN

The `NOT IN` expression syntax:

```
row_is_expression NOT IN (row_is_subquery)
```

`row_is_expression` must return the same number of columns as the `row_is_subquery`.

`row_is_expression` and `row_is_subquery` must return the same number of columns as the `row_is_subquery`.

The following query selects books of non-American authors:

```
SELECT id, title, author_id
FROM author_id NOT IN
(SELECT id FROM authors WHERE country='USA');
```

The result:

```
 id |              title              | author_id
----+---------------------------------+-----------
  1 | Mrs. Dalloway                   |         1
  2 | To the Lighthouse               |         1
  3 | The Great Gatsby                |         3
  4 | The Hobbit, or There and Back Again |         4
  5 | Bat and Peace                   |         5
  6 | Hyperion                        |         6
  7 | The Time Machine                |         7
```

`NOT IN` is equivalent to `!= ALL`.

The left part of the `NOT IN` expression can be a row constructor:

```
row_constructor NOT IN (row_is_subquery)
```

`row_is_subquery` must return the same number of columns as the `row_constructor` expression contains.

Example

```
SELECT FROM books id, books.title FROM books
WHERE author_id NOT IN
(SELECT id FROM authors WHERE country='USA');
```

The result:

```
                row
-----------------------------------
(1,"Mrs. Dalloway")
(2,"To the Lighthouse")
(3,"The Great Gatsby")
(4,"The Hobbit, or There and Back Again")
(5,Bat and Peace")
(6,(Hyp)
(7,"The Time Machine")
```

Subquery with ANY/SOME

`SOME` is a synonym for `ANY`. The `EXISTS` expression syntax:

```
<expression> <operator> ANY (subquery)
<expression> <operator> SOME (subquery)
```

`subquery` must return one column. The `<expression>` is evaluated and compared to each row of the subquery result using `<operator>`. `<operator>` must return a boolean result. The result of `ANY` is `TRUE` if any `TRUE` result is obtained. The result is `FALSE` if the result of found, or `UNKNOWN` if neither `TRUE` nor `FALSE` is obtained. If no rows are returned, the result is `FALSE`.

Note: If `<operator>` does not return `TRUE` and at least one subquery row returns `NULL`, and `ANY` result is `NULL`, not `FALSE`.

Rewrite the example with `IN` as follows:

```
SELECT id, title FROM books
WHERE author_id = ANY
(SELECT id FROM authors
 WHERE country='USA');
```

The result is the same:

```
 id |              title
----+---------------------------------
  1 | To Kill a Mockingbird
  2 | The Great Gatsby
  3 | Hyperion
```

The left part of the `ANY/SOME` expression can be a row constructor:

```
row_constructor <operator> ANY (subquery)
row_constructor <operator> SOME (subquery)
```

`row_subquery` must return the same number of columns as the `row_constructor` expression contains.

Example

```
SELECT FROM books id, books.title FROM books
WHERE author_id = ANY
(SELECT id FROM authors WHERE country='USA');
```

The result:

```
                row
-----------------------------------
(1,"To Kill a Mockingbird")
(2,"The Great Gatsby")
(3,Hyperion)
```

Subquery with ALL

The `ALL` expression syntax:

```
<all_expression> <operator> ALL (<all_subquery>)
```

`<all_subquery>` must return one column. The `<all_expression>` is evaluated and compared to each row of the subquery result using `<operator>`. `<operator>` must return a boolean result. The result of `ALL` is `TRUE` if all rows return true or if `<all_subquery>` returns no rows. The result is `FALSE` if any false result is found. The result is `NULL` if no comparisons with a subquery row result in `FALSE` or `TRUE`, and at least one comparison returns `NULL`.

`ALL ALL` is equivalent to `NOT IN`.

Rewrite the example with `ALL` as follows:

```
SELECT id, title FROM books
WHERE author_id <> ALL
(SELECT id FROM authors WHERE country='USA');
```

The result is the same:

```
 id |              title
----+---------------------------------
  1 | Mrs. Dalloway
  2 | To the Lighthouse
  3 | The Lord of the Rings
  4 | The Hobbit, or There and Back Again
  5 | Bat and Peace
  6 | (Hyp)
  7 | The Time Machine
```

The left part of the `ALL` expression can be a row constructor:

```
row_constructor <operator> ALL (<all_subquery>)
```

`<all_subquery>` must return the same number of columns as the `row_constructor` expression contains.

Example

```
SELECT FROM books id, books.title FROM books
WHERE author_id <> ALL
(SELECT id FROM authors WHERE country='USA');
```

The result:

```
                row
-----------------------------------
(1,"Mrs. Dalloway")
(2,"To the Lighthouse")
(3,"The Lord of the Rings")
(4,"The Hobbit, or There and Back Again")
(5,Bat and Peace")
(6,(Hyp)
(7,"The Time Machine")
```

Single-row comparison

The single-row comparison has the following syntax:

```
row_constructor <operator> (subquery)
```

`row_constructor` must return the same number of columns as the `row_constructor` expression contains. `(subquery)` cannot return more than one row. If it returns no rows, the result is `NULL`. You can use `<`, `>`, `<=`, `>=`, `=`, or `<>` comparison operators. The `row_constructor` is evaluated and compared to the single subquery result row using `<operator>`.

Example

```
SELECT last_name FROM employees
  WHERE (employee_hours, employee_id) =
    (SELECT employee_hours, employee_id FROM departments
      WHERE location = 'Moscow');
```

The result:

```
 last_name
-----------
 Ivanov
```

Subquery in the FROM clause

An SQL statement with subquery in the `FROM` clause has the following syntax:

```
SELECT <expression> FROM (<subquery>) AS <alias>
```

Every table in the `FROM` clause must have an alias. All columns in the subquery `SELECT` list must have unique names.

The following query shows books published after 1950 with a percentage of total sales:

```
SELECT id, book_id, title, percentage
FROM
  (SELECT id AS book_id, title AS book_title, public_year AS year, ROUND(total_sales/(SELECT SUM(total_sales) FROM books)*100,2) AS percentage FROM books) AS t1
WHERE year > 1950;
```

The result:

```
 book_id |          book_title          | percentage
---------+------------------------------+------------
       1 | Mrs. Dalloway                |       5.26
       2 | To the Lighthouse            |       4.81
       3 | To Kill a Mockingbird        |       8.76
       4 | The Great Gatsby             |       6.15
       5 | The Lord of the Rings        |       6.88
       6 | (Hyp)                        |       5.33
       7 | The Hobbit, or There and Back Again |       4.45
       8 | Hyperion                     |       7.22
```

Combine queries (UNION, INTERSECT, EXCEPT)

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

The UNION example

The INTERSECT example

The EXCEPT example

Combine set operators

You can use the `UNION`, `INTERSECT`, and `EXCEPT` set operators to combine the results of two queries. These operators have the following syntax:

```
<query1> UNION [ALL] <query2>
<query1> INTERSECT [ALL] <query2>
<query1> EXCEPT [ALL] <query2>
```

Where:

- `<query1>` and `<query2>` are queries to combine. The queries must be union compatible. They must return the same number of columns and the corresponding columns must have compatible data types, as described in the following article: UNION.
- `UNION` adds the result of `<query2>` to the result of `<query1>`. The order, in which `UNION` return rows, is not determined.
- `INTERSECT` returns all rows that are in both the `<query1>` and `<query2>` results.
- `EXCEPT` returns all rows that are in the `<query1>` result but not in the `<query2>` result.
- `ALL` specifies that duplicate rows should be included in the result. If the query does not contain the `ALL` modifier, the `UNION`, `INTERSECT`, and `EXCEPT` operators eliminate duplicate rows.

To see how the set operators work, create three tables:

```
CREATE TABLE book_store1(
    id INT PRIMARY KEY,
    title VARCHAR (100)
);

CREATE TABLE book_store2(
    id INT PRIMARY KEY,
    title VARCHAR (100)
);

CREATE TABLE book_store3(
    id INT PRIMARY KEY,
    title VARCHAR (100)
);

INSERT INTO book_store1 (id, title) VALUES
(1, 'Hyperion'),
(2, '1984'),
(3, 'War and Peace'),
(4, 'The Time Machine');

INSERT INTO book_store2 (id, title) VALUES
(3, 'War and Peace'),
(4, 'The Time Machine'),
(5, 'Mrs. Dalloway' );

INSERT INTO book_store3 (id, title) VALUES
(1, 'Hyperion'),
(2, '1984'),
(5, 'Mrs. Dalloway' );
```

The UNION example

The following query uses `UNION` to join the `book_store1` and `book_store2` tables:

```
SELECT * from book_store1 UNION SELECT * from book_store2;
```

The result:

```
id |     title
---+-------------------
5 | Mrs. Dalloway
4 | The Time Machine
2 | 1984
3 | War and Peace
1 | Hyperion
```

Add the `ALL` modifier and compare results:

```
SELECT * from book_store1 UNION ALL SELECT * from book_store2;
```

The result contains duplicate rows:

```
 id |     title
----+-------------------
  1 | Hyperion
  2 | 1984
  3 | War and Peace
  4 | The Time Machine
  3 | War and Peace
  4 | The Time Machine
  5 | Mrs. Dalloway
```

The INTERSECT example

The following query uses `INTERSECT` to return all rows that are in both the `book_store1` and `book_store2` tables:

```
SELECT * from book_store1 INTERSECT SELECT * from book_store2;
```

The result:

```
 id |     title
----+-------------------
  4 | The Time Machine
  3 | War and Peace
```

The EXCEPT example

The query below uses `EXCEPT` to returns all rows that are in `book_store1` but not in the `book_store2` table.

```
SELECT * from book_store1 EXCEPT SELECT * from book_store2;
```

The result:

```
 id |  title
----+---------
  2 | 1984
  1 | Hyperion
```

Combine set operators

You can combine set operators in a single query.

For example, the following query first adds rows from the `book_store1` and `book_store2` tables to the result, and then excludes `book_store3` rows:

```
SELECT * from book_store1 UNION SELECT * from book_store2
    EXCEPT SELECT * from book_store3;
```

The result:

```
 id |     title
----+-------------------
  4 | The Time Machine
  3 | War and Peace
```

Parentheses determine the order, in which operators are evaluated. Without parentheses, `UNION` and `EXCEPT` are executed from left to right. The expressions below are equivalent:

```
<query1> UNION <query2> EXCEPT <query3>

(<query1> UNION <query2>) EXCEPT <query3>
```

`INTERSECT` has a higher priority than `UNION` and `EXCEPT`. The expressions below are equivalent:

```
<query1> UNION <query2> INTERSECT <query3>

<query1> UNION (<query2> INTERSECT <query3>)
```

Execute the following query with `INTERSECT`:

```
SELECT * from book_store1 UNION SELECT * from book_store2
    INTERSECT SELECT * from book_store3;
```

PostgreSQL executes the `INTERSECT` first, its result is:

```
 id |     title
----+-------------------
  5 | Mrs. Dalloway
```

After that, `UNION` is executed to combine the result above with the `book_store1` rows.

The result:

```
 id |     title
----+-------------------
  5 | Mrs. Dalloway
  4 | The Time Machine
  2 | 1984
  3 | War and Peace
  1 | Hyperion
```

Add parentheses to the query:

```
(SELECT * from book_store1 UNION SELECT * from book_store2)
    INTERSECT SELECT * from book_store3;
```

PostgreSQL executes the `UNION` first, its result is:

```
 id |     title
----+-------------------
  5 | Mrs. Dalloway
  4 | The Time Machine
  2 | 1984
  3 | War and Peace
  1 | Hyperion
```

After that, `INTERSECT` returns all rows that are in both the previous result and `book_store3` table.

The result:

```
 id |     title
----+---------------
  2 | 1984
  1 | Hyperion
  5 | Mrs. Dalloway
```

If you need to use additional clauses (for example, `LIMIT`) in input queries, enclose queries in parentheses. If you use additional clauses in queries without parentheses, a syntax error may occur or a clause can be applied to the set operator output rather than an input query. For example, the following query is correct, but PostgreSQL applies `LIMIT 10` to the `UNION` result:

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10;
```

This query is equivalent to the following expression:

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10;
```

To apply `LIMIT` to the second query, use parentheses:

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10);
```

Common table expressions

Ji Ayesha Kumar

Contents

To Table of Contents

The SELECT statement in CTE

Recursive queries

- Search order
- Cycle Detection

CTE materialization

Data-modifying statements

A **common table expression (CTE)** is a temporary result of an SQL statement that can be referenced in another SQL statement. CTEs allow you to split complex SQL queries into simpler parts.

CTE has the following syntax:

```
WITH <CTE_name> (<column_list>) AS (
<CTE_query_definition>
)
<statement>;
```

In the statement above, the `WITH` clause creates CTE using the following:

- `<CTE_name>` is an identifier for CTE.
- `<column_list>` is an optional parameter that contains a list of CTE columns.
- `<CTE_query_definition>` specifies an SQL statement to populate CTE.
- `<statement>` is an SQL expression that uses CTE.

You can also define several CTEs with a single `WITH` clause.

The SELECT statement in CTE

The example below displays sale totals per book in the top sale store. The `WITH` clause defines two auxiliary statements named `store_sales` and `top_stores`. The `store_sales` output is used in `top_stores`, and the `top_stores` output is used in the parent `SELECT` query.

```
WITH store_sales AS (
    SELECT store, SUM(amount) AS total_sales
    FROM orders
    GROUP BY store
), top_stores AS (
    SELECT store
    FROM store_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM store_sales)
)
SELECT store,
       book,
       SUM(quantity) AS book_quantity,
       SUM(amount) AS book_sales
FROM orders
WHERE store IN (SELECT store FROM top_stores)
GROUP BY store, book;
```

If rewrite this query without `WITH`, it will be a complex statement with two levels of nested SELECT subqueries.

Recursive queries

Recursive queries are used to process hierarchical or tree-structured data. You can use the `RECURSIVE` modifier in a `WITH` query to refer to the query's own output.

A recursive CTE has the following syntax:

```
WITH RECURSIVE <CTE_name> AS(
    <CTE_query_definition> -- non-recursive term
    UNION [ALL]
    <CTE_query_definition> -- recursive term
)
```

A recursive `WITH` query contains a non-recursive term, `UNION` or `UNION ALL`, and a recursive term. Only the recursive term can contain a reference to the query's own output.

PostgreSQL executes a recursive query in the following sequence:

1. Execute the non-recursive term to create the base result set (R0).
2. Execute the recursive term with R1 as an input to return the result set R1+1 as the output.
3. Repeat step 2 until an empty set is returned.
4. Return the final result set that is a `UNION` or `UNION ALL` of the result sets R0, R1,... Rn.

Create a new table to demonstrate a recursive query:

```
CREATE TABLE employees (
employee_id serial PRIMARY KEY,
full_name VARCHAR NOT NULL,
manager_id INT
);
```

Insert data to the table

```
INSERT INTO employees (employee_id, full_name, manager_id)
    VALUES
    (1, 'James Wilson', NULL),
    (2, 'Mary Burton', 1),
    (3, 'Patricia Robinson', 1),
    (4, 'Robert Gray', 1),
    (5, 'Elizabeth Tucker', 2),
    (6, 'Joseph Lewis', 2),
    (7, 'William Ferguson', 2),
    (8, 'Linda Black', 3),
    (9, 'David Green', 3),
    (10, 'Donald Gray', 4),
    (11, 'Mark Armstrong', 4),
    (12, 'Donald Carter', 7),
    (13, 'Elizabeth Collins', 7),
    (14, 'Paul Brown', 8),
    (15, 'Andrew Clarke', 8);
```

The following recursive query returns all the subordinates of the manager with `employee_id` equal to `2`:

```
WITH RECURSIVE subordinates (employee_id, full_name, manager_id) AS (
    SELECT employee_id, manager_id, full_name
    FROM employees WHERE employee_id = 2
    UNION
        SELECT e.employee_id, e.manager_id, e.full_name
        FROM employees e
    INNER JOIN subordinates s ON s.employee_id = e.manager_id
)
SELECT * FROM subordinates;
```

The non-recursive term returns the base result set (R0):

```
employee_id | manager_id |  full_name
-------------+------------+---------------
          2 |          1 | Mary Burton
```

The recursive term returns the subordinates of the employee from the first result. This is the `JOIN` result of the `employees` table and the `subordinates` CTE. The first iteration of the recursive term returns the following result:

```
employee_id | manager_id |    full_name
-------------+------------+-----------------
          5 |          2 | Elizabeth Tucker
          6 |          2 | Joseph Lewis
          7 |          2 | William Ferguson
```

The second iteration uses the result set above as the input, and returns the result set:

```
employee_id | manager_id |    full_name
-------------+------------+-----------------
         12 |          7 | Donald Carter
         13 |          7 | Elizabeth Collins
         10 |          5 | Daniel Gray
```

The third iteration returns an empty result set because employees with `employee_id` 10, 12, and 13 do not have subordinates.

The final result set is the union of all result sets in the first and second iterations generated by the non-recursive and recursive terms.

```
employee_id | manager_id |    full_name
-------------+------------+-----------------
          2 |          1 | Mary Burton
          5 |          2 | Elizabeth Tucker
          6 |          2 | Joseph Lewis
          7 |          2 | William Ferguson
         12 |          7 | Donald Carter
         13 |          7 | Elizabeth Collins
         10 |          5 | Daniel Gray
```

Search order

If you need to get the results in the depth-first or breadth-first order, you can use the `SEARCH` clause.

Add `SEARCH DEPTH FIRST BY` to display the result in the depth-first order. As a result, a column that contains the path to the row is added to the output.

```
WITH RECURSIVE subordinates(employee_id, manager_id, full_name) AS (
    SELECT employee_id, manager_id, full_name
    FROM employees WHERE employee_id = 2
    UNION
        SELECT e.employee_id, e.manager_id, e.full_name
        FROM employees e
    INNER JOIN subordinates s ON s.employee_id = e.manager_id
) SEARCH DEPTH FIRST BY employee_id SET ordercol
SELECT * FROM subordinates ORDER BY ordercol;
```

The result:

```
employee_id | manager_id |    full_name     |    ordercol
-------------+------------+------------------+-----------------
          2 |          1 | Mary Burton      | {(2)}
          5 |          2 | Elizabeth Tucker | {(2,(5))}
         10 |          5 | Daniel Gray      | {(2,(5),(10))}
          6 |          2 | Joseph Lewis     | {(2,(6))}
          7 |          2 | William Ferguson | {(2,(7))}
         12 |          7 | Donald Carter    | {(2,(7),(12))}
         13 |          7 | Elizabeth Collins | {(2,(7),(13))}
```

Use `SEARCH BREADTH FIRST BY` to display the result in the breadth-first order. As a result, a column that contains the nested level of the recursive query is added to the output.

```
WITH RECURSIVE subordinates(employee_id, manager_id, full_name) AS (
    SELECT employee_id, manager_id, full_name
    FROM employees WHERE employee_id = 2
    UNION
        SELECT e.employee_id, e.manager_id, e.full_name
        FROM employees e
    INNER JOIN subordinates s ON s.employee_id = e.manager_id
) SEARCH BREADTH FIRST BY employee_id SET ordercol
SELECT * FROM subordinates ORDER BY ordercol;
```

The result:

```
employee_id | manager_id |    full_name     | ordercol
-------------+------------+------------------+----------
          2 |          1 | Mary Burton      | {0,2}
          5 |          2 | Elizabeth Tucker | {1,5}
          6 |          2 | Joseph Lewis     | {1,6}
          7 |          2 | William Ferguson | {1,7}
         10 |          5 | Daniel Gray      | {2,10}
         12 |          7 | Donald Carter    | {2,12}
         13 |          7 | Elizabeth Collins | {2,13}
```

Cycle Detection

When working with recursive queries, it is important to make sure that the recursive part of the query eventually returns no items, otherwise the query loops indefinitely. You can use the `CYCLE` clause to detect cycles.

```
WITH RECURSIVE subordinates(employee_id, manager_id, full_name) AS (
    SELECT employee_id, manager_id, full_name
    FROM employees WHERE employee_id = 2
    UNION
        SELECT e.employee_id, e.manager_id, e.full_name
        FROM employees e
    INNER JOIN subordinates s ON s.employee_id = e.manager_id
) CYCLE employee_id SET is_cycle USING path
SELECT * FROM subordinates;
```

The `CYCLE` clause contains the list of columns to track for cycle detection (`employee_id` - in the example above) a name of column that shows whether a cycle is detected (`is_cycle`), and the name of another column that tracks the path (`path`). The `is_cycle` and `path` columns are added to the query output.

The result:

```
employee_id | manager_id |    full_name     | is_cycle |      path
-------------+------------+------------------+----------+-----------------
          2 |          1 | Mary Burton      | f        | {(2)}
          5 |          2 | Elizabeth Tucker | f        | {(2),(5)}
          6 |          2 | Joseph Lewis     | f        | {(2),(6)}
          7 |          2 | William Ferguson | f        | {(2),(7)}
         12 |          7 | Donald Carter    | f        | {(2),(7),(12)}
         13 |          7 | Elizabeth Collins | f        | {(2),(7),(13)}
         10 |          5 | Daniel Gray      | f        | {(2),(5),(10)}
```

TIP

A query can include both `SEARCH` and `CYCLE` clauses. The cycle path column is computed in the same way as the depth-first ordering column. The `SEARCH DEPTH FIRST BY` and `CYCLE` clauses share common internal computations. To avoid it, use the `CYCLE` clause and order by the `path` column if you need to apply the breadth-first order, specify both `SEARCH` and `CYCLE` clauses.

CTE materialization

You can define whether the result of the CTE subquery (starting with `AS`) is **materialized**. It means that the subquery is executed separately from the parent query and only once during its execution, and its result is written to a temporary table stored in memory. It can be useful if this subquery contains resource intensive calculations.

Use the `MATERIALIZED` modifier to force the subquery separate calculation from the parent query:

```
WITH employees_data AS MATERIALIZED (
    SELECT * FROM employees
)
SELECT full_name FROM employees_data WHERE employee_id = 5;
```

The following subqueries are materialized by default:

- subqueries from recursive CTE queries;
- subqueries that are referenced more than once.

Since a materialized subquery is executed and optimized separately from the parent query it may cause the performance problems. That is why the default behavior was changed starting with PostgreSQL 12. In recent PostgreSQL versions, CTE subqueries referenced once are not materialized. They are merged into the parent queries and the optimizer processes both query simultaneously. For example, the optimizer can combine conditions from two queries into one filter to search by index.

You can use the `NOT MATERIALIZED` modifier to force the joint optimization of the subquery and parent query.

```
WITH employees_data AS NOT MATERIALIZED (
    SELECT * FROM employees
)
SELECT (
    SELECT full_name FROM employees_data WHERE employee_id = 7),
    (SELECT full_name FROM employees_data WHERE employee_id = 3);
```

Here, the `NOT MATERIALIZED` modifier has no effect on recursive CTE queries, they are always materialized.

Data-modifying statements

You can use the `INSERT`, `UPDATE` and `DELETE` statements in CTEs. It allows you to perform multiple different operations in the same query. The example below moves two rows from `employees` to `retired_employees`:

```
WITH moved_rows AS (
    DELETE FROM employees
    WHERE
        employee_id > 6 OR
        employee_id < 5
    RETURNING *
)
INSERT INTO retired_employee
SELECT * FROM moved_rows;
```

`DELETE` removes the specified rows from `employees`, returns their content by the `RETURNING` clause. The parent query reads the subquery output and inserts it into `retired_employee`.

Data-modifying statements in `WITH` should contain the `RETURNING` clause to refer to their result sets in the parent query.

Note that the CTE statement execution order is not determined. The query below returns not modified data from the `employees` table, because the `UPDATE` statement is executed later than `SELECT`:

```
WITH employees_data AS (
    UPDATE employees SET manager_id = 1 WHERE employee_id =10
    RETURNING *
)
SELECT * FROM employees;
```

To get the modified data, refer to the subquery result:

```
WITH employees_data AS (
    UPDATE employees SET manager_id = 1 WHERE employee_id =10
    RETURNING *
)
SELECT * FROM employees_data;
```

Recursive self-references in data-modifying statements are not allowed. You can include data-modifying statement in the parent query. The following recursive query deletes all subordinates of the manager with the id `2` and they manager from `employees_data`:

```
WITH RECURSIVE subordinates (employee_id, full_name, manager_id) AS (
    SELECT employee_id, manager_id, full_name
    FROM employees WHERE employee_id = 2
    UNION
        SELECT e.employee_id, e.manager_id, e.full_name
        FROM employees e
    INNER JOIN subordinates s ON s.employee_id = e.manager_id
)
DELETE FROM employees
  WHERE employee_id in (SELECT employee_id FROM subordinates);
```

Aggregate functions

Eugenia Kuzina

AD.PG

Contents

To Table of Contents

Overview

Aggregate expression syntax

Overview

Aggregate functions compute a single result from a set of input values. For example, you can compute the count, sum, average, maximum, and minimum over a set of rows.

Example:

```
SELECT MIN(public_year) FROM book;
```

The result:

```
 min
------
 1869
```

To utilize an aggregate function in the `WHERE` clause, use a subquery. The following example returns the title of the oldest book:

```
SELECT title FROM book WHERE public_year = (SELECT min(public_year) FROM book);
```

The result:

```
     title
---------------
 War and Peace
```

You can also combine aggregate functions with the `GROUP BY` clauses. For example, we can get the oldest book of each genre:

```
SELECT genre, min(public_year) public_year FROM book GROUP BY genre;
```

The result:

```
      genre      | public_year
-----------------+-------------
 science fiction |        1895
 fantasy         |        1937
 novel           |        1869
```

You can use the `HAVING` clause to filter the result set:

```
SELECT genre, min(public_year) public_year
    FROM book GROUP BY genre HAVING min(public_year) < 1900;
```

The result:

```
      genre      | min
-----------------+------
 science fiction | 1895
 novel           | 1869
```

> **NOTE**
>
> There is a fundamental difference between the `WHERE` and `HAVING` clauses. `WHERE` selects input rows before evaluating groups and aggregate functions, it determines which rows are included in aggregate calculations. `HAVING` selects group rows after groups and aggregate functions are calculated. The `WHERE` clause cannot contain aggregate functions, and the `HAVING` clause always contains aggregate functions.
> If you need to use additional conditions, add them to the `WHERE` clause, not `HAVING`. This approach avoids aggregation calculations for rows that do not meet the additional conditions.

To obtain the full list of predefined aggregate functions, refer to the following link: Aggregate functions.

Aggregate expression syntax

The syntax of an aggregate expression can be one of the following:

1. The first form of aggregate expression invokes the aggregate function once for each input row according to the specified expression:

```
<aggregate_name> (<expression> [ , ... ] [<order_by_clause>] )
    [ FILTER ( WHERE <filter_clause> ) ]
```

   Where:

   - `<aggregate_name>` is an aggregate function;
   - `<expression>` is a value expression that does not contain any aggregate expression or window function call;
   - `<order_by_clause>` is a clause that specifies the sort order (optional);
   - `<filter_clause>` is a clause that defines a filter (optional).

   Example:

```
SELECT MIN(public_year) FROM book;
```

2. The second form has the `ALL` modifier. Since `ALL` is used by default, the second form is the same as the first.

```
<aggregate_name> (ALL <expression> [ , ... ] [<order_by_clause>] )
    [ FILTER (WHERE <filter_clause>) ]
```

3. The third form invokes the aggregate function once for each distinct value of the expression (or distinct set of values, for multiple expressions).

```
<aggregate_name> (DISTINCT <expression> [ , ... ] [<order_by_clause>] )
    [ FILTER (WHERE <filter_clause>) ]
```

   The following statement returns the number of distinct book titles:

```
SELECT COUNT (DISTINCT(book_title)) FROM orders;
```

4. The fourth form invokes the aggregate function once for each row. Since no particular input value is specified, this form only applies to the `COUNT(*)` function.

```
<aggregate_name> ( * ) [ FILTER (WHERE <filter_clause>) ]
```

5. The fifth form is used with ordered-set aggregate functions (for example, `array_agg`, `string_agg`, `rank`, and `percentile`). The result of these functions depends on the order of the input rows. The `order_by_clause` parameter is required for these functions.

```
<aggregate_name> ( [ <expression> [ , ... ] ] )
    WITHIN GROUP ( <order_by_clause>)
    [FILTER (WHERE <filter_clause>)]
```

   The `ORDER BY` clause has the same syntax as a query-level `ORDER BY` clause, but cannot contain output-column numbers and names.

   Example:

```
SELECT array_agg(field1 ORDER BY field2 DESC) FROM table1;
```

   If you call multiple-argument aggregate functions, `ORDER BY` must be placed after all aggregate arguments:

```
SELECT string_agg(field1, ',' ORDER BY field1) FROM table1;
```

   If `DISTINCT` is specified in addition to `ORDER BY`, all the `ORDER BY` expressions must match regular arguments of the aggregate function. You cannot sort by an expression that is not included in the DISTINCT list.

   For an ordered-set aggregate function, you can put `order_by_clause` inside `WITHIN GROUP(…)`. The expressions in `order_by_clause` are evaluated once per input row as regular aggregate arguments, sorted according to `order_by_clause` requirements, and passed to the aggregate function as input arguments. When `order_by_clause` is not in `WITHIN GROUP`, it is not treated as an aggregate function argument. The argument expressions preceding `WITHIN GROUP`, if any, are called direct arguments to distinguish them from the aggregated arguments listed in the `order_by_clause`. Direct arguments are evaluated once per aggregate call. They can contain variables only if these variables are grouped by `GROUP BY`. Direct arguments are typically used for a single value per aggregation calculation (for example, percentile fractions). If the direct argument list is empty, write `()`.

   Example:

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
```

For all types of aggregate expression listed above, `filter_clause` is an optional parameter. If `FILTER` is specified, only the input rows for which the `filter_clause` evaluates to `true` are passed to the aggregate function.

We have the following table:

```
 id |              book_title              | public_year
----+--------------------------------------+-------------
  1 | Mrs. Dalloway                        |        1925
  2 | To the Lighthouse                    |        1927
  3 | To Kill a Mockingbird                |        1960
  4 | The Great Gatsby                     |        1925
  5 | The Lord of the Rings                |        1955
  8 | War and Peace                        |        1869
  7 | The Hobbit, or There and Back Again  |        1937
  9 | Hyperion                             |        1989
 10 | The Time Machine                     |        1895
  6 | 1984                                 |        1949
```

Execute query with the specified `FILTER`:

```
SELECT
    COUNT(*) AS unfiltered,
    COUNT(*) FILTER (WHERE public_year < 1900) AS filtered
FROM book;
```

The result:

```
 unfiltered | filtered
------------+----------
         10 |        2
```

Most aggregate functions ignore NULL input, and rows where one or more expressions evaluate to NULL are discarded. This is true for all built-in aggregate functions unless otherwise specified. For example, `COUNT(*)` returns the total number of input rows; `COUNT(field1)` returns the number of input rows where `field1` is not null, and `COUNT(DISTINCT field1)` returns the number of distinct non-null values of `field1`.

Window functions

Evgeniia Kozina

To Table of Contents

Window functions perform a calculation across a set of rows. They do not return a single output row as aggregate functions, window functions include some additional information in the selection result.

The simplified window function syntax can be written as follows:

```
<function_name> OVER (<windows>)
```

Where:

- `<function_name>` is a window function name;
- `<windows>` is an expression that describes the set of rows to process and the processing order.

For information on the extended window function syntax, refer to Window Function Calls.

If you need to pass all result rows to a window function, use empty parentheses `()`. For example, the following query adds row numbers to the result:

```
SELECT id, title, evaluation,
       row_number() OVER () AS numbers
FROM books;
```

The result:

```
 id |                 title                  | evaluation | numbers
----+----------------------------------------+------------+---------
  1 | Mrs. Dalloway                          |       7.82 |       1
  2 | To the Lighthouse                      |       8.45 |       2
  3 | To Kill a Mockingbird                  |       7.48 |       3
  4 | The Great Gatsby                       |       9.23 |       4
  5 | The Lord of the Rings                  |       9.49 |       5
  6 | 1984                                   |       8.17 |       6
  7 | The Hobbit, or There and Back Again    |       9.32 |       7
  8 | War and Peace                          |       9.69 |       8
  9 | Hyperion                               |       9.46 |       9
 10 | The Time Machine                       |       8.12 |      10
```

You can add the `ORDER BY` clause to the window function to change the processing order:

```
SELECT id, title, genre, evaluation,
       row_number() OVER (ORDER BY evaluation DESC) AS rating
FROM books
ORDER BY id;
```

The result:

```
 id |                 title                  |  genre  | evaluation | rating
----+----------------------------------------+---------+------------+--------
  1 | Mrs. Dalloway                          | novel   |       7.82 |      9
  2 | To the Lighthouse                      | novel   |       8.45 |      6
  3 | To Kill a Mockingbird                  | novel   |       7.48 |     10
  4 | The Great Gatsby                       | novel   |       9.23 |      5
  5 | The Lord of the Rings                  | fantasy |       9.49 |      2
  6 | 1984                                   | sci-fi  |       8.17 |      7
  7 | The Hobbit, or There and Back Again    | fantasy |       9.32 |      4
  8 | War and Peace                          | novel   |       9.69 |      1
  9 | Hyperion                               | sci-fi  |       9.46 |      3
 10 | The Time Machine                       | sci-fi  |       8.12 |      8
```

In the example above, a query-level sort (`ORDER BY id`) is added to the query. PostgreSQL first evaluates the window function results and then sorts its results with the query results according to `ORDER BY id`. `ORDER BY` at the window level and `ORDER BY` at the query level are applied correctly and do not interfere with each other.

You can use the `PARTITION BY` clause to divide rows into groups or **partitions** that share the same values of the `PARTITION BY` expression. The window function calculates a row value across the rows of the same partition. The following example calculates a book rating across books of the same genre:

```
SELECT id, title, genre, evaluation,
       row_number() OVER (PARTITION BY genre ORDER BY evaluation DESC) AS rating
FROM books
ORDER BY genre;
```

The result:

```
 id |                 title                  |  genre  | evaluation | rating
----+----------------------------------------+---------+------------+--------
  5 | The Lord of the Rings                  | fantasy |       9.49 |      1
  7 | The Hobbit, or There and Back Again    | fantasy |       9.32 |      2
  8 | War and Peace                          | novel   |       9.69 |      1
  4 | The Great Gatsby                       | novel   |       9.23 |      2
  2 | To the Lighthouse                      | novel   |       8.45 |      3
  1 | Mrs. Dalloway                          | novel   |       7.82 |      4
  3 | To Kill a Mockingbird                  | novel   |       7.48 |      5
  9 | Hyperion                               | sci-fi  |       9.46 |      1
  6 | 1984                                   | sci-fi  |       8.17 |      2
 10 | The Time Machine                       | sci-fi  |       8.12 |      3
```

If you omit `PARTITION BY`, a window function is evaluated for all rows.

The list of predefined window functions is available at the following link: General-purpose window functions.

You can also use aggregate functions `sum`, `count` and others as window functions. If `ORDER BY` is not specified in the window clause of an aggregate function, the function is calculated for the entire partition and its result is written to all partition rows.

For example, we have the table `orders`:

```
 id | customer_id |  total
----+-------------+---------
  1 |          17 | 3500.00
  2 |          22 | 1000.00
  4 |          46 | 3300.00
 11 |          38 | 2000.00
 12 |          17 | 1750.00
  3 |          17 | 5000.00
  5 |          17 | 4700.00
  6 |          17 | 2000.00
  7 |          46 | 1900.00
  8 |          22 | 2100.00
  9 |          22 | 1400.00
 10 |          13 | 1400.00
 13 |          46 | 4600.00
 14 |          13 | 2250.00
```

The query below calculates the sum of orders for each customer:

```
SELECT id, customer_id,
    sum(total) OVER (PARTITION BY customer_id) as sum
FROM orders;
```

The result:

```
 id | customer_id |   sum
----+-------------+----------
 10 |          13 |  3650.00
 14 |          13 |  3650.00
  6 |          17 | 16950.00
 12 |          17 | 16950.00
  5 |          17 | 16950.00
  3 |          17 | 16950.00
  1 |          17 | 16950.00
  2 |          22 |  4500.00
  9 |          22 |  4500.00
  8 |          22 |  4500.00
 11 |          38 |  2000.00
  7 |          46 |  9800.00
  4 |          46 |  9800.00
 13 |          46 |  9800.00
```

If `ORDER BY` is specified, the function is calculated for rows from the beginning of the partition to the current row. If the next row contains the same value of the field specified in the `ORDER BY` clause, it is included in the calculation. This set of rows is called **window frame**. Some window functions operate with window frame rows, rather than with the entire partition. For example, the `sum` function works with window frames. Add `ORDER BY` to the example above:

```
SELECT id, customer_id,
    sum(total) OVER (PARTITION BY customer_id ORDER BY id) as sum
FROM orders;
```

The result:

```
 id | customer_id |   sum
----+-------------+----------
 10 |          13 |  1400.00
 14 |          13 |  3650.00
  1 |          17 |  3500.00
  3 |          17 |  8500.00
  5 |          17 | 13200.00
  6 |          17 | 15200.00
 12 |          17 | 16950.00
  2 |          22 |  1000.00
  8 |          22 |  3100.00
  9 |          22 |  4500.00
 11 |          38 |  2000.00
  4 |          46 |  3300.00
  7 |          46 |  5200.00
 13 |          46 |  9800.00
```

The result contains the running total for each partition.

You can use window functions in the `SELECT` and `ORDER BY` clause. They cannot be included in the `GROUP BY`, `HAVING`, `WHERE`, and other clauses, because window functions are calculated after the processing of these clauses. Also, window functions are executed after non-window aggregate functions. You can include an aggregate function call in the arguments of a window function.

To filter or group rows after a window function calculation, use a subquery. The following query utilizes subquery to apply a filter and display rows where `sum` less than `5000`:

```
SELECT id, customer_id, sum
 FROM
   (SELECT id, customer_id, sum(total) OVER (PARTITION BY customer_id ORDER BY id) as sum
      FROM orders
   )AS orders_with_sum
 WHERE sum < 5000;
```

The result:

```
 id | customer_id |   sum
----+-------------+----------
 10 |          13 |  1400.00
 14 |          13 |  3650.00
  1 |          17 |  3500.00
  2 |          22 |  1000.00
  8 |          22 |  3100.00
  9 |          22 |  4500.00
 11 |          38 |  2000.00
  4 |          46 |  3300.00
```

You can use several window functions in a single query:

```
SELECT id, customer_id,
    sum(total) OVER (PARTITION BY customer_id ORDER BY id) as sum,
    round(avg(total) OVER(PARTITION BY customer_id),2) as avg
FROM orders;
```

The result:

```
 id | customer_id |   sum    |   avg
----+-------------+----------+---------
 10 |          13 |  1400.00 | 1825.00
 14 |          13 |  3650.00 | 1825.00
  1 |          17 |  3500.00 | 3390.00
  3 |          17 |  8500.00 | 3390.00
  5 |          17 | 13200.00 | 3390.00
  6 |          17 | 15200.00 | 3390.00
 12 |          17 | 16950.00 | 3390.00
  2 |          22 |  1000.00 | 1500.00
  8 |          22 |  3100.00 | 1500.00
  9 |          22 |  4500.00 | 1500.00
 11 |          38 |  2000.00 | 2000.00
  4 |          46 |  3300.00 | 3266.67
  7 |          46 |  5200.00 | 3266.67
 13 |          46 |  9800.00 | 3266.67
```

If multiple window functions have the same `OVER` clause, you can name it and place separately with the `WINDOW` keyword. Rewrite the query above as follows:

```
SELECT id, customer_id,
    sum(total) OVER (w ORDER BY id) as sum,
    round(avg(total) OVER(w),2) as avg
FROM orders
WINDOW w AS (PARTITION BY customer_id);
```

The result:

```
 id | customer_id |   sum    |   avg
----+-------------+----------+---------
 10 |          13 |  1400.00 | 1825.00
 14 |          13 |  3650.00 | 1825.00
  1 |          17 |  3500.00 | 3390.00
  3 |          17 |  8500.00 | 3390.00
  5 |          17 | 13200.00 | 3390.00
  6 |          17 | 15200.00 | 3390.00
 12 |          17 | 16950.00 | 3390.00
  2 |          22 |  1000.00 | 1500.00
  8 |          22 |  3100.00 | 1500.00
  9 |          22 |  4500.00 | 1500.00
 11 |          38 |  2000.00 | 2000.00
  4 |          46 |  3300.00 | 3266.67
  7 |          46 |  5200.00 | 3266.67
 13 |          46 |  9800.00 | 3266.67
```

JSON/JSONB data types

Contents

To Table of Contents

- JSON and JSONB data types
  - JSON and JSONB encoding
  - JSONB converting
- JSON and JSONB syntax
- Store JSON data in a table
- Use operators to query JSON data
- Use JSON operators in the WHERE clause
- Contain and exist operators
- Apply aggregate functions to JSON data
- PostgreSQL functions to process the JSON and JSONB types
  - json_each/jsonb_each
  - json_object_keys/jsonb_object_keys
  - json_typeof/jsonb_typeof
  - json_extract_path/jsonb_extract_path
  - jsonb_pretty

PostgreSQL offers two types to store JSON data defined in RFC 7159: `JSON` and `JSONB`. To implement efficient query mechanisms for these data types, PostgreSQL also provides the `JSONPATH` data type.

JSON and JSONB data types

Both `JSON` and `JSONB` store values with JSON values. `JSONB` stores data in a plain text format, as a copy of the input text. `JSONB` stores data in a decomposed binary format. These formats have the following differences:

- `JSON` is faster to input, because there is no conversion process.
- `JSONB` operations run significantly faster than `JSON`. `JSONB` operations, processing functions reparse data on each execution, while `JSON` stores parsed values.
- `JSONB` supports indexes. `JSON` does not.

`JSON` is compatible with the JSON standard specification and PostgreSQL, it is used to store and retrieve JSON data. If you do a lot of operations in PostgreSQL, or use indexing on JSON data, use `JSONB`. For more information on indexes, refer to Indexing to efficiently query JSON data stored in the `JSONB` format.

`JSONB` also preserves only the last value in key-value pairs when duplicate keys are passed, unlike the `JSON` object. If a JSON object keeps the values the same way more than once, all the key-value pairs are kept. The processing functions consider the last value as the operative one. `JSONB` does not preserve white spaces, the order of object keys, and does not keep duplicate object keys passed in the input. In the case of duplicate keys, only the last value is kept.

JSONB converting

When converting text to `JSONB`, the primitive types are mapped onto native PostgreSQL types, as shown in the table below. `JSONB` has some minor additional constraints, corresponding to restrictions of the PostgreSQL data types. For example, `JSONB` rejects numbers that are outside the range of the PostgreSQL `numeric` data type.

JSON primitive type	PostgreSQL type	Notes
string	text	`JSONB` does not allow `\u0000` and Unicode escapes corresponding to encoding not supported by the database encoding
number	numeric	The `NaN` and `Infinity` values are not allowed
boolean	boolean	Only lowercase `true` and `false` spellings are accepted
null	–	This is not SQL `NULL`

JSON and JSONB syntax

The following JSON representations are valid for the `JSON` and `JSONB` types:

- Scalar JSON primitive value. Primitive values can be numbers, quoted strings, `true`, `false`, or `null`. Example:

```
SELECT '5'::json;
SELECT '"test string"'::jsonb;
```

- Array of zero or more elements. Array can contain elements of different types. Example:

```
SELECT '[1, 2, "string element", null]'::json;
```

- Objects that contain pairs of keys and values. Key is a quoted string. Example:

```
SELECT '{"key": "key", "balance": 7.77, "active": false}'::jsonb;
```

- Nested arrays and objects. Example:

```
SELECT '{"key": [true, "key"], "key2": {"a": 1, "b": null}}'::json;
```

Store JSON data to a table

To store JSON data to a table, use a `JSONB` or `JSON` column. For example, we create a new table `book_orders` with a `JSONB` column:

```
CREATE TABLE book_orders (
    id serial NOT NULL PRIMARY KEY,
    content json NOT NULL
);
```

To insert data into a table, ensure that data is in a valid JSON format. The following code adds new rows to the `book_orders` table:

```
INSERT INTO book_orders (content)
VALUES ('{ "customer": "John Johnson", "items": {"product": "Diapers", "qty": 3},
          { "customer": "Anne Antonov", "items": {"product": "Toy Car", "qty": 2},
          { "customer": "Andrew Anderson", "items": {"product": "Toy Train", "qty": 1}');
```

Use operators to query JSON data

You can use the `SELECT` statement to get JSON data in the same way as you operate with other data types:

```
SELECT content FROM book_orders;
```

The result:

```
                                                content
-------------------------------------------------------------------------------------------------
 {"items": {"qty": 3, "book": "War and Peace"}, "customer": "Anne Antonov"}
 {"items": {"qty": 2, "book": "1984"}, "customer": "Andrew Antonov"}
 {"items": {"qty": 1, "book": "Hyperion"}, "customer": "Sarah Johnson"}
```

PostgreSQL also provides operators to work with JSON data types. For example, the `->` operator returns a JSON object by key. The following query uses the `->` operator to get all customers as JSON objects:

```
SELECT content -> 'customer' AS customer
FROM book_orders;
```

The result:

```
   customer
---------------
 "John Johnson"
 "Anne Antonov"
 "Sarah Johnson"
```

The `->>` operator uses the `TEXT` operator to get all customers in the text form:

```
SELECT content ->> 'customer' AS customer
FROM book_orders;
```

The result:

```
   customer
---------------
 John Johnson
 Anne Antonov
 Sarah Johnson
```

Since the `TEXT` operator returns a JSON object, you can use it with the `->` operator to retrieve a specific node. For example, the following statement returns all books:

```
SELECT content -> 'items' ->> 'book' AS book
FROM book_orders;
```

The result:

```
    book
------------
 War and Peace
 1984
 Hyperion
```

`content -> 'items'` returns `items` as JSON objects. `content ->'items' ->> 'book'` returns all books as text.

It is also possible to get a JSON object at the specified path. To do this, use the `#>` operator:

The following statement returns JSON objects at the path `'{items,book}'`:

```
SELECT content #> '{items, book}' AS books
FROM book_orders;
```

The result:

```
    books
-------------
 "War and Peace"
 "1984"
 "Hyperion"
```

You can also return JSON objects at the specified path as text. To do this, use the `#>>` operator:

```
SELECT content #>> '{items, book}' AS books
FROM book_orders;
```

The result:

```
    books
-------------
 War and Peace
 1984
 Hyperion
```

Use JSON operators in the WHERE clause

You can include JSON operators in the `WHERE` clause to filter rows. For example, the following query returns a customer who bought the "War and Peace" book:

```
SELECT content ->> 'customer' AS customer
FROM book_orders
WHERE content ->> 'items' ->> 'book' = 'War and Peace';
```

The result:

```
  customer
--------------
 Anne Brown
```

Utilize a type cast in the `WHERE` clause to use comparison operators. In the query below, the type cast is used to convert the `qty` value into the `INTEGER` type and compare it with `3`:

```
SELECT content ->> 'customer' AS customer,
content ->> 'items' ->> 'book' AS book
FROM book_orders
WHERE CAST ( content ->> 'items' ->> 'qty' AS INTEGER ) = 3;
```

The result:

```
  customer   |    book
-------------+-------------
 Jacob Johnson | Hyperion
```

Contain and exist operators

It is possible to rewrite the first `WHERE` clause example with the additional JSONB operator `@>` that determines whether the first JSON value contains the second one:

```
SELECT content ->> 'customer' AS customer
FROM book_orders
WHERE content -> 'items' -> 'book' @> '"War and Peace"'::jsonb;
```

The result is the same:

```
  customer
--------------
 Anne Brown
```

The `@>` operator takes into account the element nesting level.

For example, the following statement returns `false`:

```
SELECT '[1, 2, 3]'::jsonb @> '[[1, 2]]'::jsonb;
```

However, the example below returns `false`:

```
SELECT '[1, 2, [1, 2]]'::jsonb @> '[1, 2]'::jsonb;
```

It is necessary to explicitly specify the nesting level to get the `TRUE` result:

```
SELECT '[1, 2, [1, 2]]'::jsonb @> '[[1, 2]]'::jsonb;
```

The `?` operator is a variation of `@>`. It determines whether a `JSONB` object contains the specified string as an object key or array element at the top level. The following statement returns `true`:

```
SELECT content ? 'customer'
FROM book_orders;
```

For more information on additional `JSONB` operators, refer to additional JSONB operators.

Apply aggregate functions to JSON data

PostgreSQL 9.5 introduced `json_agg` and `jsonb_agg` aggregate functions to JSON data. For example, the following statement returns minimum, maximum, average, and the total quantity of books. The statement uses the `->` operator to get the `qty` value and `CAST` to cast the text data type to an appropriate type for aggregate functions.

```
SELECT
    MIN(CAST (content -> 'items' ->> 'qty' AS INTEGER)) AS min,
    MAX (CAST (content -> 'items' ->> 'qty' AS INTEGER)),
    AVG (CAST (content -> 'items' ->> 'qty' AS INTEGER)),
    SUM (CAST (content -> 'items' ->> 'qty' AS INTEGER))
FROM book_orders;
```

The result:

```
 min | min | avg | sum
-----+-----+-----+-----
   1 |   3 | 2.0 |   6
```

PostgreSQL functions to process the JSON and JSONB types

PostgreSQL provides a set of functions to process `JSON` / `JSONB` data, some of them are described below.

json_each/jsonb_each

The `json_each` and `jsonb_each` functions expand the top-level JSON object into a set of key-value pairs.

The function have the following syntax:

```
json_each ([some] a anlyst record [key text, value json]
jsonb_each ([some] a anlyst record [key text, value jsonb]
```

Example:

```
SELECT * jsonb_each(content)
FROM book_orders;
```

The result:

```
                    jsonb_each
---------------------------------------------------
 (items,"{""qty"": 3, ""book"": ""War and Peace""}")
 (customer,"""Anne Brown""")
 (items,"{""qty"": 2, ""book"": ""1984""}")
 (customer,"""Anne Antonov""")
 (items,"{""qty"": 1, ""book"": ""Hyperion""}")
 (customer,"""Sarah Johnson""")
```

To get a set of key-value pairs as text, use the `jsonb_each_text` or `json_each_text` function:

json_object_keys/jsonb_object_keys

The `json_object_keys` and `jsonb_object_keys` functions return a set of keys of the top-level JSON object.

The function have the following syntax:

```
json_object_keys([some] a anlyst text
jsonb_object_keys([some] a anlyst text
```

The statement below displays a set of keys for the `items` objects:

```
SELECT jsonb_object_keys (content->'items')
FROM book_orders;
```

The result:

```
 jsonb_object_keys
-------------------
 qty
 book
 qty
 book
 qty
 book
```

json_typeof/jsonb_typeof

The `json_typeof` and `jsonb_typeof` functions return a type of the top-level JSON value as a string. They can return the `number`, `boolean`, `null`, `object`, `array`, and `string` values.

The function have the following syntax:

```
json_typeof([some] a anlyst text
jsonb_typeof([some] a anlyst text
```

The statement below displays the type of the `qty` objects:

```
SELECT jsonb_typeof (content->'items'->'qty')
FROM book_orders;
```

The result:

```
 jsonb_typeof
--------------
 number
 number
 number
```

json_extract_path/jsonb_extract_path

The `jsonb_extract_path` and `json_extract_path` functions extract JSON objects at the specified path. It is equivalent to the `#>` operator.

The functions have the following syntax:

```
json_extract_path ( <from_json> json, <VARIADIC path_elems text[] > ) type
jsonb_extract_path ( <from_json> jsonb, <VARIADIC path_elems text[] > ) jsonb
```

The statement below displays `qty` objects:

```
SELECT jsonb_extract_path (content, 'items', 'qty')
FROM book_orders;
```

The result:

```
 jsonb_extract_path
--------------------
 3
 2
 1
```

jsonb_pretty

The `jsonb_pretty` function converts a JSON object to the prettу-printed, indented text.

The function has the following syntax:

```
jsonb_pretty ([some] a text
```

The following example displays the `items` objects:

```
SELECT jsonb_pretty(content->'items')
FROM book_orders;
```

The result:

```
    jsonb_pretty
-------------------------
 {                      +
     "qty": 3,          +
     "book": "War and Peace"+
 }
 {                      +
     "qty": 2,          +
     "book": "1984"     +
 }
 {                      +
     "qty": 1,          +
     "book": "Hyperion" +
 }
```

JSONPath

1 quadrillion IDs

Contents

Table of Contents

- JSONPath functions
  - jsonb_path_exists
  - jsonb_path_match
  - jsonb_path_query
  - jsonb_path_query_array
  - jsonb_path_query_first
  - JSONPath functions with the _tz suffix
  - The vars argument
  - The silent argument
- JSONPath operators
  - The match operator @@
  - The existence check operator @?
- The strict and lax modes

The `JSONPath` type implements support for the SQL/JSON path language in PostgreSQL to efficiently query JSON data stored in the `JSONB` format. `JSONPath` is a binary representation of the parsed SQL/JSON path expression that specifies the items to be retrieved by the path engine.

`JSONPath` uses the following JavaScript convention syntax:

- The dot (`.`) is used for member access.
- Square brackets (`[]`) are used for array access.
- SQL/JSON arrays are 0-relative, unlike regular SQL arrays that start from 1.

Enclose an SQL/JSON path expression in single quotes to include it in an SQL query. To use a single-quote character in a value, double it. For example: `'$.name.\"a b\"'`.

Some forms of path expression require string literals. These embedded string literals must be surrounded by double quotes. These string literals can contain backslash escapes. To write a double quote, use `\"`. To write a backslash, type `\\`. Other special backslash sequences that can be used in JSON literals: `\b`, `\f`, `\n`, `\r`, `\t` for ASCII control characters, `\uNNNN` and `\uNNNNN` unicode escapes (where `N` represents one hex digits, but hex digits are only from zero to six hex digits respectively).

A path expression can consist of the following path elements:

- Path variables:
  - `$` is the current context in the filter expression with `@` specifies the array to process and is used in a filter. Example: `$.items`
  - `@` is the current context in the filter expression (the result of path evaluation in filter expressions). Example: `$.items[*] ? (@.qty > 2)`
  - `$variable` as a named variable. Its value can be set by the parameter `vars`. Example: `'$[*] ? (@ > $x)'::jsonb, '{"x": 2}'`
- A filter expression that starts with a question mark (`?`) and contains a condition in parentheses:

```
? (condition)
```

It operates similar to the `WHERE` clause.

- `JSONPath` operators and methods listed in the following section: SQL/JSON Path Operators and Methods.
- Parentheses that are used to specify filter expressions or define the order of path evaluation.
- Accessor operators that are listed in the table below:

Accessor operator	Description	Example
Any "$username"	A member accessor that returns an object member with the specified key. If the key name matches a named variable starting with `$` or does not meet the JavaScript rules for an identifier, it must be enclosed in double quotes	$.items
*	A wildcard member accessor that returns the values of all members located at the top level of the current object	$.items.*
**	A recursive wild-card member accessor that processes all levels of the JSON hierarchy of the current object and returns all the member values, regardless of their nesting level	$.**
**{level} **{start_level to end_level}	This member accessor selects only the specified levels of the JSON hierarchy. A nesting level is an integer value. The `0` level corresponds to the current object. To access the lowest nesting level, use the `last` keyword.	$.**{1} $.**{1 to 3}
[start_index to end_index]	An array element accessor that can be used in two forms: `index` or `start_index to end_index`. The first form returns a single array element by its index. The second form returns an array including the elements that correspond to the `start_index` and `end_index`. The specified index can be an integer or an expression returning a single numeric value. An array index is zero-based for integers. The `0` index corresponds to the first array element. You can also use the `last` keyword to denote the last array element	$.a[0] $.a[3 to 5]
[*]	A wildcard array element accessor that returns all array elements	$.items[*]

You can use `JSONPath` syntax to create simpler queries than the SQL syntax allows.

JSONPath functions

The `JSONPath` data type implements `JSONB` data. All three functions have `vars` and `silent` optional arguments. `vars` allows you to pass variables into a JSONPath expression. `silent` determines whether error handling is suppressed. These arguments are described below.

Create the `book_orders` table and insert data for tests.

```
CREATE TABLE book_orders (
  id serial NOT NULL PRIMARY KEY,
  content jsonb NOT NULL
);

INSERT INTO book_orders (content)
VALUES ('{ "supplier": "Jacob Johnson", "items": {"book": "Hyperion","qty": 1, "price": 2222 },
        { "supplier": "Adam Brown", "items": {"book": "War and Peace","qty": 2, "price": 3111 },
        { "supplier": "Andrew Nelson", "items": {"book": "1984","qty": 4, "price": 3111 }};

CREATE TABLE demand(name text, max_price int);

INSERT INTO demand VALUES ('Jane', 25), ('Alex', 27), ('Ben', 33);
```

jsonb_path_exists

```
jsonb_path_exists ( <target> jsonb, <path> jsonpath [, vars jsonb [, silent boolean ]] ) → boolean
```

The function checks whether `path` returns any item for the `target` JSON value.

Examples:

```
SELECT jsonb_path_exists('{"a": 1}', '$.a');
-- returns true

SELECT jsonb_path_exists('{"a": 1}', '$.b');
-- returns false

SELECT jsonb_path_exists('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ == $min && @ <= $max)', '{"min":2, "max":4}');
-- returns true
```

The following query determines if the JSON data stored in the `content` column of the `book_orders` table contains strings that have the `Hyperion` value:

```
SELECT jsonb_path_exists(content, '$.** ? (@ == "Hyperion")')
FROM book_orders WHERE id =1;

 jsonb_path_exists
-------------------
 t
```

`$.**` iterates through the JSON value hierarchy of the `content` field and returns all element values, regardless of their nesting level. `@` refers to the current element value in the filter expression.

The result:

```
 jsonb_path_exists
-------------------
 true
 false
 false
```

jsonb_path_match

```
jsonb_path_match ( <target> jsonb, <path> jsonpath [, vars jsonb [, silent boolean ]] ) → boolean
```

The function returns the result of the `path` predicate check for the `target` JSON value. The function takes into account only the first item of the result. If the result is not `boolean`, `jsonb_path_match` returns `NULL`.

Examples:

```
SELECT jsonb_path_match('{"a": 1}', '$.a == 1');
-- returns true

SELECT jsonb_path_match('{"a": 1}', '$.a == 2');
-- returns false

SELECT jsonb_path_match('{"a":[1,2,3,4,5]}', 'exists($.a[*] ? (@ == $min && @ <= $max))', '{"min":2, "max":4}');
-- returns true
```

The following query also determines if the `content` column of the `book_orders` row contains strings that have the `Hyperion` value:

```
SELECT jsonb_path_match(content, 'exists($.** ? (@ == "Hyperion"))')
FROM book_orders WHERE id =1;

 jsonb_path_match
------------------
 t
```

The result:

```
 jsonb_path_match
------------------
 true
 false
 false
```

jsonb_path_query

```
jsonb_path_query ( <target> jsonb, <path> jsonpath [, vars jsonb [, silent boolean ]] ) → setof jsonb
```

The function returns all JSON items that `path` extracts from the `target` JSON value.

Examples:

```
SELECT jsonb_path_query('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > 2)');
-- returns 3, 4, 5

SELECT jsonb_path_query('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > $x)', '{"x": 2}');
-- returns 3, 4, 5
```

The following query returns `items` where the `qty` field more than 2:

```
SELECT jsonb_path_query(content, '$.items[*] ? (@.qty > 2)')
FROM book_orders;
```

The result:

```
          jsonb_path_query
-------------------------------------
 {"qty": 4, "book": "Hyperion", "price": 2222}
 {"qty": 4, "book": "1984", "price": 3111}
```

jsonb_path_query_array

```
jsonb_path_query_array ( <target> jsonb, <path> jsonpath [, vars jsonb [, silent boolean ]] ) → jsonb
```

The function returns all JSON items that `path` extracts from the `target` JSON value as a JSON array.

Examples:

```
SELECT jsonb_path_query_array('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > 2)');
-- returns [3, 4, 5]

SELECT jsonb_path_query_array('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > $x)', '{"x": 2}');
-- returns [3, 4, 5]
```

jsonb_path_query_first

```
jsonb_path_query_first ( <target> jsonb, <path> jsonpath [, vars jsonb [, silent boolean ]] ) → jsonb
```

The function returns the first JSON item that `path` extracts from the `target` JSON value. `jsonb_path_query_first` returns `null` if there are no results.

Examples:

```
SELECT jsonb_path_query_first('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > 2)');
-- returns 3

SELECT jsonb_path_query_first('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > $x)');
-- returns 3
```

JSONPath functions with the _tz suffix

PostgreSQL provides the following functions with the `_tz` suffix: `jsonb_path_exists_tz`, `jsonb_path_match_tz`, `jsonb_path_query_tz`, `jsonb_path_query_array_tz` and `jsonb_path_query_first_tz`. These functions operate as their counterparts described above without the `_tz` suffix, but support operations with timezone-aware (`TIMESTAMP WITH TIME ZONE`) data and comparisons.

The example below requires creation of the date-only value `2022-08-02` as a timestamp with time zone. The result depends on the current `TimeZone` setting. Because of this dependency, PostgreSQL marks these functions as stable, which means that you cannot use these functions in indexes.

```
SELECT jsonb_path_exists_tz('["2022-08-01 22:08:08 -08"]', '$[*] ? (@.datetime() < "2022-08-02".datetime())');
-- returns true

SELECT jsonb_path_exists_tz('["2022-08-01 12:08:08"]', '$[*] ? (@.datetime() < "2022-08-02".datetime())');
-- returns true
```

The vars argument

The `vars` argument allows you to pass variables into a `JSONPath` expression. It must be a JSON object, and its fields must contain named values to be substituted into the `JSONPath` expression.

Execute the following query with the `vars` argument to find books with a price lower than `max_price` specified for `JCase` in the `demand` table:

```
SELECT jsonb_path_query(content, '$.items[*] ? (@.price < $max_price)',
  vars => json_build_object('max_price', demand.max_price))
  FROM book_orders, demand WHERE name = 'Alex';
```

The `vars` argument passes the result of the query that determines `max_price` specified for `JCase` in the `jsonb_path_query` function.

The result:

```
          jsonb_path_query
-------------------------------------
 {"qty": 1, "book": "Hyperion", "price": 2222}
 {"qty": 2, "book": "War and Peace", "price": 3111}
```

The silent argument

`silent` suppresses error handling. Execute the following query in which the `strict` modifier changes the error handling mode:

```
SELECT jsonb_path_query('[]', 'strict $.a');
```

An error occurs:

```
ERROR: jsonpath member accessor can only be applied to an object
```

Set `silent` → `true`:

```
SELECT jsonb_path_query('[]', 'strict $.a', silent => true);
```

The error is not displayed. The result:

```
 jsonb_path_query
------------------
(0 rows)
```

JSONPath operators

The match operator @@

The `@@` operator returns the result of the `JSONPath` predicate check for the specified JSON value. PostgreSQL takes into account only the first item of the result. If the result is not boolean, `@@` returns `NULL`.

The following query determines if the `content` column of the `book_orders` table contains strings that have the `Hyperion` value:

```
SELECT content @@ '$.** == "Hyperion"' AS hyperion FROM book_orders;
```

The result:

```
 hyperion
----------
 t
 null
 null
```

The existence check operator @?

The `@?` operator determines whether a `JSONPath` expression returns any item for the specified JSON value.

The following query adds `@?` to select books with the `price` value more than 25.

```
SELECT jsonb_path_query(content, '$.items.book')
FROM book_orders WHERE content @? '$.items[*] ? (@.price > 25)';
```

The result:

```
 jsonb_path_query
------------------
 "War and Peace"
 "1984"
```

The strict and lax modes

When you query JSON data, the `JSONPath` expression may not match the actual JSON data structure. An attempt to access a non-existent member or element is a structural error. `JSONPath` expressions have two modes of handling structural errors:

- `lax` (default) - the path engine implicitly adapts the queried data to the specified path. Any remaining structural errors are suppressed and converted to empty SQL/JSON sequences.
- `strict` - if a structural error occurs, an error is raised.

Use the `lax` and `strict` modifiers to switch modes.

To see how these modes operate, search the `$x` key in a JSON value that does not contain this key. In the `lax` mode, the result is `false`:

```
SELECT jsonb '{"a":1}' @? 'lax $.b ? (@ > 1)';
```

The result:

```
 jsonb_path_query
------------------
 false
```

In the `strict` mode, it returns `NULL`:

```
SELECT jsonb '{"a":1}' @? 'strict $.b ? (@ > 1)';
```

The result:

```
 jsonb_path_query
------------------
 null
```

In the `lax` mode, an array with a complex hierarchy [1,2,[3,[4]]] is processed as [1,2,3,[4]]:

```
SELECT jsonb '[1,2,[3,[4,5]]]' @? 'lax $[*] ? (@ == 5)';
```

The result in `true`.

In the `strict` mode, the query returns `false`:

```
SELECT jsonb '[1,2,[3,[4,5]]]' @? 'strict $[*] ? (@ == 5)';
```

Replace `@` with `@[*]` to get the `true` result in the `strict` mode:

```
SELECT jsonb '[1,2,[3,[4,5]]]' @? 'strict $[*] ? (@[*] == 5)';
```

For more information on modes, see Strict and lax modes.

XML data type

Contents

Create XML values
- The schema function
- PostgreSQL-specific conversion

Convert XML to a character string

Create XML content elements
- xmlcomment
- xmlelement
- xmlforest
- xmlpi
- xmlroot
- xmltext

Check XML value properties
- IS DOCUMENT / IS NOT DOCUMENT
- XMLEXISTS
- xml_is_well_formed

Process XML data
- Extract XML values
- Store and retrieve XML values

Export tables to XML

To Table of Contents

You can use the `XML` data type to store XML data. As for the data type, PostgreSQL validates the input data structure and provides functionality to perform type-safe operations.

To use the `XML` data type effectively, you should be familiar with the XML standard and related concepts such as [XPath](https://www.postgresql.org/docs/current/functions-xml.html#FUNCTIONS-XML-PROCESSING).

NOTE

The PostgreSQL `XML` data type uses libxml2 parsing functions for validation. For more information, refer to the [libxml2 documentation](http://xmlsoft.org/).

Create XML values

You can use the `xmlparse` function to generate values of the `XML` data type. To pass those values, create a new table and use `XML` column.

```
CREATE TABLE xmltest (id INT4, xmldata XML, content TEXT);
```

The xmlparse function

The `xmlparse` function processes the `XML` input values data. It has the following syntax:

```
xmlparse ( { DOCUMENT | CONTENT } value )
```

Put the required document into the `value` argument. Use the `DOCUMENT` or `CONTENT` keyword to specify the argument type. `DOCUMENT` checks whether the input value has a single root node, while `CONTENT` checks for well-formed content.

Example

```
INSERT INTO xmltest (id, xmldata)
    VALUES (1, xmlparse (DOCUMENT '<?xml version="1.0"?><book><title>Manual for QEMU</title><chapter>HTML Viewer</chapter></book>'));
INSERT INTO xmltest (id, xmldata)
    VALUES (2, xmlparse (CONTENT 'abc<foo>bar</foo><bar>foo</bar>'));
```

The result in the table:

```
id |                    xmldata
---+-----------------------------------------------
 1 | <?xml version="1.0"?><book><title>Manual for QEMU</title><chapter>HTML Viewer</chapter></book>
 2 | abc<foo>bar</foo><bar>foo</bar>
```

PostgreSQL-specific conversion

The following syntax allows you to convert PostgreSQL data types to `XML`:

```
xmlparse ( content_expression )
```

The `content_expression` value specifies which PostgreSQL data type elements to convert into the `XML` format. For example:

```
SELECT xmlparse (CONTENT 'abc<foo>bar</foo><bar>foo</bar>');
```

Example

```
SELECT xmlparse (CONTENT '
    <book>
        <title>Manual for QEMU</title>
        <chapter>HTML Viewer</chapter>
    </book>');
```

The result in the table:

```
                  xmlparse
-----------------------------------------------
 <book>                                       +
     <title>Manual for QEMU</title>           +
     <chapter>HTML Viewer</chapter>           +
 </book>
```

Convert XML to a character string

To convert a value from the `XML` data type, use the `xmlserialize` function:

```
xmlserialize ( { DOCUMENT | CONTENT } value AS type )
```

`DOCUMENT` and `CONTENT` keywords work the same as in the `xmlparse` function. `value` specifies the `XML` value to convert. `type` specifies the character or binary string data type to convert `value` into.

Example

```
SELECT xmlserialize (CONTENT xmldata AS text )
FROM xmltest;
```

The result:

```
                  xmlserialize
-----------------------------------------------
 <book>                                       +
     <title>Manual for QEMU</title>           +
     <chapter>HTML Viewer</chapter>           +
 </book>
 abc<foo>bar</foo><bar>foo</bar>
```

According to the SQL standard, the `xmlserialize` function is the only way to convert `XML` to character types (or PostgreSQL data types you can treat like them).

```
SELECT CAST(xmldata AS text)
FROM xmltest;
```

When a character string value is cast to the `XML` type with the `xmlparse` or `xmlserialize` functions, the `DOCUMENT` or `CONTENT` keywords determine how to parse the string. This can be done with specifying the type modifier when the string value is cast to `XML`, because in this case there is no way to pass it through the `xmlparse` or `xmlserialize` function. For example:

```
SET xmloption TO DOCUMENT;
SELECT 'abc'::xml;
SET xmloption TO CONTENT;
SELECT 'abc'::xml;
```

The result:

```
ERROR:  invalid XML document
DETAIL:  line 1: Start tag expected, '<' not found
abc
```

Create XML content elements

This section describes functions and predicates for creating XML content from SQL data. You can also use them to examine existing XML content.

xmlcomment

The `xmlcomment` function creates an XML comment with the specified text.

```
xmlcomment ( 'text' )
```

`text` is the content of the comment. It cannot contain `--` or end with a `-` character.

Example

```
SELECT xmlcomment ('This is a comment');
```

The result:

```
      xmlcomment
-----------------------
 <!--This is a comment-->
```

xmlconcat

The `xmlconcat` function concatenates a list of individual XML values to create a single value containing an XML content fragment.

```
xmlconcat ( value [, ...] )
```

Null values omit from the result. For example, `NULL` arguments of three or more arguments.

Example

```
SELECT xmlconcat('<abc/>', NULL, '<bar>foo</bar>');
```

The result:

```
      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

xmlelement

The `xmlelement` function creates an XML element with the given name, attributes, and content.

```
xmlelement ( NAME name [, XMLATTRIBUTES (attvalue [ AS attname ] [, ...])] [, content [, ...] ] )
```

Where:

- `name` and `attname` are identifiers.
- `XMLATTRIBUTES` clause generates attributes from the values in the specified `attname`.
- `attvalue` and `content` are expressions.

Example

```
SELECT xmlelement (name test, xmlattributes('1' as num), 'abc');
```

The result:

```
                 xmlelement
--------------------------------------------
 <test num="1">abc</test>
```

xmlforest

The `xmlforest` function creates an XML element (with the given name, attributes, and content) from multiple given values.

```
xmlforest ( content [ AS name ] [, ...] )
```

Where:

- `content` is an expression.
- `name` is an identifier.

Example

```
SELECT xmlforest(table_name, column_name)
FROM information_segment.columns
WHERE table_name = 'pg_settings';
```

The result:

```
                                 xmlforest
-----------------------------------------------------------------------
 <table_name>pg_settings</table_name><column_name>name</column_name>
 <table_name>pg_settings</table_name><column_name>setting</column_name>
 <table_name>pg_settings</table_name><column_name>unit</column_name>
 <table_name>pg_settings</table_name><column_name>category</column_name>
 <table_name>pg_settings</table_name><column_name>short_desc</column_name>
 <table_name>pg_settings</table_name><column_name>extra_desc</column_name>
 <table_name>pg_settings</table_name><column_name>context</column_name>
 <table_name>pg_settings</table_name><column_name>vartype</column_name>
 <table_name>pg_settings</table_name><column_name>source</column_name>
 <table_name>pg_settings</table_name><column_name>min_val</column_name>
 <table_name>pg_settings</table_name><column_name>max_val</column_name>
 <table_name>pg_settings</table_name><column_name>enumvals</column_name>
 <table_name>pg_settings</table_name><column_name>boot_val</column_name>
 <table_name>pg_settings</table_name><column_name>reset_val</column_name>
 <table_name>pg_settings</table_name><column_name>sourcefile</column_name>
 <table_name>pg_settings</table_name><column_name>sourceline</column_name>
 <table_name>pg_settings</table_name><column_name>pending_restart</column_name>
```

xmlpi

The `xmlpi` function creates an XML processing instruction.

```
xmlpi ( NAME name [, content] )
```

Where `name` is an identifier, and `content` is an expression. The expression must not contain the character sequence `?>`.

Example

```
SELECT xmlpi(name php, 'echo "hello world";');
```

The result:

```
              xmlpi
------------------------------------
 <?php echo "hello world";?>
```

xmlroot

The `xmlroot` function constructs the properties of the XML root node.

```
xmlroot ( value, VERSION { expression | NO VALUE } [, STANDALONE {YES|NO|NO VALUE} ] )
```

`VERSION` and `STANDALONE` settings are specified they replace values in the root node's version declaration.

Example

```
SELECT xmlroot(xmlparse(DOCUMENT '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);
```

The result:

```
                            xmlroot
----------------------------------------------------------------
 <?xml version="1.0" standalone="yes"?><content>abc</content>
```

xmlagg

The `xmlagg` function is an [aggregate function](https://www.postgresql.org/docs/current/functions-aggregate.html). It concatenates the input values across rows.

Call `xmlagg` like an [aggregate function](https://www.postgresql.org/docs/current/functions-aggregate.html), for example:

```
SELECT xmlagg(xmlelement (NAME test, xmlattributes(a)))
FROM test;
```

The result:

```
                    xmlagg
----------------------------------------------
 <test a="1" />
 <test a="2" />
 <test a="3" />
```

xmltext

The `xmltext` function creates an XML text node.

```
xmltext ( text )
```

Where `text` is the content of the text node. It cannot contain the `]]>` character sequence.

Example

```
SELECT xmltext('Hello World');
```

The result:

```
  xmltext
-------------
 Hello World
```

Check XML value properties

You can use the `IS DOCUMENT` and `IS NOT DOCUMENT` predicates to check whether a value is a document or a content fragment. It's useful when the data type is `XML`.

IS DOCUMENT / IS NOT DOCUMENT

```
value IS [ NOT ] DOCUMENT
```

Example

```
SELECT xmlparse (CONTENT 'abc<foo>bar</foo><bar>foo</bar>') IS DOCUMENT;
```

The result:

```
 ?column?
----------
 f
```

`NOT` is the same as `IS DOCUMENT`. Use `NOT` or `IS NOT DOCUMENT` to check `false` value is returned. `IS DOCUMENT` returns `false` or `IS NOT DOCUMENT` returns `true` if the argument is an XML content fragment.

```
SELECT xmlparse (DOCUMENT '<foo>bar</foo>') IS DOCUMENT;
```

The result:

```
 ?column?
----------
 t
```

XMLEXISTS

The `XMLEXISTS` function determines whether a specific [XPath](https://www.postgresql.org/docs/current/functions-xml.html#FUNCTIONS-XML-PROCESSING) expression returns any values.

```
XMLEXISTS ( 'xpath' PASSING [BY {REF|VALUE}] xml [BY {REF|VALUE}] )
```

Where `xpath` is an XPath expression, and `xml` is an XML value.

Example

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE
'<towns><town>Toronto</town><town>Ottawa</town></towns>');
```

The result:

```
 xmlexists
-----------
 t
```

xml_is_well_formed

The `xml_is_well_formed` function checks whether a string is a well-formed XML document.

```
xml_is_well_formed ( string )
```

`xml_is_well_formed` returns `true` if the string is a well-formed document, and `false` if it's not.

`xml_is_well_formed_document` verifies whether a string is a well-formed document, and `xml_is_well_formed_content` verifies whether a string is well-formed content.

Example

```
SELECT xml_is_well_formed('<abc/>');
```

The result:

```
 xml_is_well_formed
--------------------
 t
```

`xml_is_well_formed` is similar to the `xmlparse` function. `xmlparse` raises an error if the argument is not well-formed, while `xml_is_well_formed` returns `false`.

Process XML data

You can use the `SQL/XML` functions to process values of the `XML` data type. Also, you can use extensions, such as [PL/Perl](https://www.postgresql.org/docs/current/plperl.html).

Extract XML values

The `xpath` function allows you to query `XML` values corresponding to the XPath expression. If the XPath expression returns [XML fragment](https://www.postgresql.org/docs/current/functions-xml.html#FUNCTIONS-XML-PROCESSING), `xpath` returns values from all elements.

```
xpath ( xpath, xml [, nsarray ] )
```

Where:

- `xpath` is an XPath expression.
- `xml` is an XML value.
- `nsarray` is an array of text values to map prefixes and namespaces.

Example

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);
```

The result:

```
 xpath
--------
 {test}
```

Store and retrieve XML values

You can store `XML` values in tables and retrieve them. For example:

```
CREATE TABLE test (id int, xmldata xml);
INSERT INTO test VALUES (1, '<abc>abc</abc>');
SELECT xmldata FROM test;
```

The result:

```
                 xmldata
-----------------------------------------
 <abc>abc</abc>
```

Create a table based on an XML value

The `XMLTABLE` function maps the result of an XML value. You can specify the XML `PATH` clause.

The `XMLTABLE` command has the following syntax:

```
XMLTABLE (
    [ XMLNAMESPACES ( namespace_uri AS namespace_name [, ...] ), ]
    row_expression PASSING [BY {REF|VALUE}] document_expression [BY {REF|VALUE}]
    COLUMNS name { type [PATH column_expression] [DEFAULT default_expression] [NOT NULL | NULL]
                 | FOR ORDINALITY }
                 [, ...]
)
```

Where:

- `XMLNAMESPACES` specifies a comma-separated list of namespaces definitions, where `namespace_uri` is a text expression and `namespace_name` is a simple identifier. It specifies the namespaces used in the XML document and their aliases.
- `row_expression` is an XPath expression that specifies the XPath expression for each row.
- `document_expression` specifies the XML document.
- `BY REF` is accepted but ignored.
- `BY VALUE` specifies that each row processes the entire XML document.
- `COLUMNS` specifies the columns in the resulting table. Use `name type` to specify the column name and type. If the column type is not `XML`, `PATH` specifies the XPath expression that is used to extract the column value from the document. Otherwise, `name` is used as the XPath expression. `DEFAULT expression` specifies the value for the column whose XML expression returns no values.
- `FOR ORDINALITY` specifies that the column should be filled with sequence numbers, starting with 1.

Example

```
CREATE TABLE myschema.A (id int, xmldata XML);
INSERT INTO myschema.A VALUES
  (1, '<data><row id="1"><a>1</a><b>2</b><c>3</c></row><row id="2"><a>4</a><b>5</b><c>6</c></row></data>'),
  (2, '<data><row id="3"><a>7</a><b>8</b><c>9</c></row><row id="4"><a>10</a><b>11</b><c>12</c></row></data>');
SELECT xmldata.* FROM myschema.A,
  XMLTABLE('/data/row' PASSING A.xmldata COLUMNS id int PATH '@id', a int, b int, c int) AS xmldata;
```

The result:

```
 id | a  | b  | c
----+----+----+----
  1 |  1 |  2 |  3
  2 |  4 |  5 |  6
  3 |  7 |  8 |  9
  4 | 10 | 11 | 12
```

Use the [`xpath_table`](https://www.postgresql.org/docs/current/xml2.html) function to run the same query:

```
SELECT xpath_table.* FROM myschema.A,
  xpath_table('id', 'xmldata', 'A', '/data/row', 'id int PATH @id, a int, b int, c int') AS xpath_table;
```

The result:

```
 id | a  | b  | c
----+----+----+----
  1 |  1 |  2 |  3
  2 |  4 |  5 |  6
  3 |  7 |  8 |  9
  4 | 10 | 11 | 12
```

Export tables to XML

The following functions map `SQL` values based on relational tables to XML:

```
table_to_xml ( tbl regclass, nulls boolean, tableforest boolean, targetns text )
query_to_xml ( query text, nulls boolean, tableforest boolean, targetns text )
cursor_to_xml ( cursor refcursor, nulls boolean, tableforest boolean, targetns text )
```

- `table_to_xml` maps the content of the named table, passed in the `tbl` parameter. The `regclass` type accepts strings identical to the forms accepted by the `::regclass` cast (that is, optionally schema-qualified table names are recognized).
- `query_to_xml` executes the query whose text is passed in the `query` parameter and maps the result set.
- `cursor_to_xml` fetches the indicated number of rows from the cursor specified by the `cursor` parameter. This method is recommended if you want to dump a table with many rows to XML, because you may have to work with fairly large XML documents.

The `nulls` flag determines whether null values should be included in the output. `true` means that null values in columns are represented as `<columnname xsi:nil="true" />`, where `xsi` is the XML namespace prefix for XML schema instance. An appropriate `targetns` declaration is always added. `false` means that null values are omitted from the output.

The `tableforest` flag determines whether the XML output must contain a `<table_name>` tag for each row. `true` means that it must not contain it. `false` means that it must contain it.

Use the `targetns` parameter to specify the desired XML namespace. If you do not want a namespace, pass an empty string.

```
cursor_to_xmlschema ( cursor refcursor, nulls boolean, tableforest boolean, targetns text )
query_to_xmlschema ( query text, nulls boolean, tableforest boolean, targetns text )
table_to_xmlschema ( tbl regclass, nulls boolean, tableforest boolean, targetns text )
```

These functions produce an XML Schema document that describes the result structure of the corresponding functions above that produce XML data. Call these functions directly or combine them with `xmlconcat`.

To produce a self-contained document, combine the value with `xmlelement`. For example:

```
query_to_xml_and_xmlschema ( query text, nulls boolean, tableforest boolean, targetns text )
```

Example

```
SELECT table_to_xml('myschema.A', true, true, '');
```

The result:

```
<myschema.A xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <id>1</id>
    <xmldata>
      <data>
        <row id="1">
          <a>1</a>
          <b>2</b>
          <c>3</c>
        </row>
        <row id="2">
          <a>4</a>
          <b>5</b>
          <c>6</c>
        </row>
      </data>
    </xmldata>
  </row>
  <row>
    <id>2</id>
    <xmldata>
      <data>
        <row id="3">
          <a>7</a>
          <b>8</b>
          <c>9</c>
        </row>
        <row id="4">
          <a>10</a>
          <b>11</b>
          <c>12</c>
        </row>
      </data>
    </xmldata>
  </row>
</myschema.A>
```

Hstore data type

Lorenzo Alberton

Contents

To Table of Contents

Operations with hstore values

- Get value by key
- Add a key/value pair to an existing `hstore` value
- Update a key/value pair
- Add multiple key/value pairs
- Remove a key/value pair
- Determine if a key exists
- Check if a key/value pair exists
- Get rows that contain multiple specified keys
- Get all keys
- Get all values
- Convert hstore data to a set of records
- Convert hstore data to JSON/JSONB

Create indexes on the hstore field

The `hstore` data type allows you to store key-value pairs in a single value in PostgreSQL. `hstore` is useful when you cannot predefine keys, or keys can vary for different objects.

The hstore module implements the `hstore` data type. You need to create the `hstore` extension to use this type:

```
CREATE EXTENSION hstore;
```

The `hstore` value includes `key => value` pairs separated by commas. Keys and values are stored as text strings.

Create the `products` table with an `hstore` column to see how to work with this type:

```
CREATE TABLE products (
    id serial NOT NULL,
    name text NOT NULL,
    attributes hstore
);
```

In this table, different products can have different attributes. Add the following values to the table:

```
INSERT INTO products (name, attributes) VALUES (
  'The Time Machine',
  author => 'Herbert Wells',
  pages => 202,
  genre => 'sci-fi'),
('Hyperion',
  author => 'Dan Simmons',
  pages => 761,
  genre => 'sci-fi'),
('The Great Gatsby',
  author => 'F. Scott Fitzgerald',
  pages => 256,
  genre => 'novel',
  year => 1925),
('The Time Machine',
  author => 'H. G. Wells',
  number => 38,
  date => 'AD 12 2022'
);
```

The table content

```
 id |      name      |                                  attributes
----+----------------+-----------------------------------------------------------------------
  1 | The Time Machine | "genre"=>"sci-fi", "pages"=>"202", "author"=>"Herbert Wells"
  2 | Hyperion       | "genre"=>"sci-fi", "pages"=>"761", "author"=>"Dan Simmons"
  3 | The Great Gatsby | "genre"=>"novel", "pages"=>"256", "author"=>"F. Scott Fitzgerald"
  4 | People         | "date"=>"AD 12 2022", "number"=>"38", "publisher"=>"Time Inc"
```

The `hstore` type has the following syntax specifics:

- The order of `hstore` pairs is not significant and may not be reproduced on output.
- PostgreSQL ignores whitespaces between pairs or around the `=>` sign. Keys and values that include whitespace, commas, `=>` or `=` characters should be double-quoted. To include a double quote (`"`) or a backslash (`\`) in a key or value, escape it with a backslash (`\"`, `\\`).
- Each key should be unique.
- A value can contain SQL `NULL` `key => NULL`. The `NULL` keyword is not case-sensitive. Double-quote it to process as the `"NULL"` string value.

Operations with hstore values

PostgreSQL provides a variety of operators and functions to work with the `hstore` type. This section gives examples of typical operations that you can perform with this functionality. For more information, see the `hstore` operators and function, refer to the PostgreSQL documentation: hstore operators and functions.

Get value by key

You can use the `->` operator to retrieve a value of a specific key from an `hstore` column. If the key is missing, a corresponding row is not included in the result.

```
SELECT name, attributes->'author' as author
FROM products;
```

The result:

```
      name      |      author
----------------+-------------------
 The Time Machine | Herbert Wells
 Hyperion       | Dan Simmons
 The Great Gatsby | F. Scott Fitzgerald
```

You can also utilize the `?` operator in the `WHERE` clause. The following statement selects products of the `sci-fi` genre:

```
SELECT name, attributes->'author' as author
FROM products
WHERE attributes -> 'genre' = 'sci-fi';
```

The result:

```
      name      |     author
----------------+-----------------
 The Time Machine | Herbert Wells
 Hyperion       | Dan Simmons
```

Add a key/value pair to an existing `hstore` value

You can add a new key/value pair to existing rows. For this, use the `||` operator that concatenates two `hstore` values. The following statement adds `"freeshipping"=>"yes"` to all existing rows:

```
UPDATE products
SET attributes = attributes || '"freeshipping"=>"yes"'::hstore;
```

Update a key/value pair

To update an existing key/value pair, utilize the same `||` operator. For example, the following statement updates the `freeshipping` value of the `Hyperion` book with `no`:

```
UPDATE products
SET attributes = attributes || '"freeshipping"=>"no"'::hstore
WHERE name = 'Hyperion';
```

The table content:

```
 id |      name      |                                      attributes
----+----------------+---------------------------------------------------------------------------------
  1 | The Time Machine | "genre"=>"sci-fi", "pages"=>"202", "author"=>"Herbert Wells", "freeshipping"=>"yes"
  2 | Hyperion       | "genre"=>"sci-fi", "pages"=>"761", "author"=>"Dan Simmons", "freeshipping"=>"no"
  3 | The Great Gatsby | "genre"=>"novel", "pages"=>"256", "author"=>"F. Scott Fitzgerald", "freeshipping"=>"yes"
  4 | People         | "date"=>"AD 12 2022", "number"=>"38", "publisher"=>"Time Inc", "freeshipping"=>"yes"
```

Add multiple key/value pairs

To add several key/value pairs, use a two-dimensional array that contains keys and values. The following statement replaces the `attributes` field value in the row where `id = 4`:

```
UPDATE products SET attributes = hstore(ARRAY['pages', '32'],['media', 'ebook']) WHERE id = 4;
```

The query below adds values to the `attributes` field value in the row where `id = 4`:

```
UPDATE products SET attributes = attributes || hstore(ARRAY[['pages', '32'],['media', 'ebook']]) WHERE id = 4;
```

Remove a key/value pair

Use the `delete` function to remove a key/value pair from the `hstore` column:

```
UPDATE products
SET attributes = delete(attributes, 'freeshipping');
```

You can also utilize an array to delete several keys:

```
UPDATE products
SET attributes = delete(attributes, ARRAY['genre', 'pages']);
```

Determine if a key exists

Use the `?` operator to check if a specific key exists in the `hstore` column:

```
SELECT name, attributes -> 'publisher' as publisher
FROM products
WHERE attributes ? 'publisher';
```

The result:

```
  name  | publisher
--------+-----------
 People | Time Inc
```

Check if a key/value pair exists

To check if a key/value pair exists, utilize the `@>` operator that determines whether the left operand contains the right one. The following statement retrieves all rows where the `attributes` column contains `{"freeshipping"=>"yes"}`:

```
SELECT name, attributes -> 'freeshipping' as freeshipping
FROM products
WHERE attributes @> '"freeshipping"=>"yes"'::hstore;
```

The result:

```
      name      | freeshipping
----------------+--------------
 The Time Machine | yes
 The Great Gatsby | yes
 People         | yes
```

Get rows that contain multiple specified keys

Use the `?&` operator to get rows with the several specified keys. The following statement returns rows with the `genre` and `pages` keys:

```
SELECT name FROM products
WHERE attributes ?& ARRAY [ 'genre', 'pages' ];
```

The result:

```
      name
----------------
 The Time Machine
 The Great Gatsby
 Hyperion
```

Get all keys

Use the `akeys` function to get all keys from an `hstore` column as an array:

```
SELECT akeys (attributes) FROM products;
```

The result:

```
                akeys
-------------------------------------
 {genre,pages,author,freeshipping}
 {genre,pages,author,freeshipping}
 {date,number,publisher,freeshipping}
 {genre,pages,author,freeshipping}
```

Utilize the `skeys` function to get all keys as a set of records:

```
SELECT skeys (attributes) FROM products;
```

The result:

```
   skeys
-------------
 genre
 pages
 author
 freeshipping
 genre
 pages
 author
 freeshipping
 date
 number
 publisher
 freeshipping
 genre
 pages
 author
 freeshipping
```

Get all values

Use the `avals` function to get all values from an `hstore` column as an array:

```
SELECT avals (attributes) FROM products;
```

The result:

```
                avals
-------------------------------------
 {sci-fi,202,"Herbert Wells",yes}
 {novel,256,"F. Scott Fitzgerald",yes}
 {AD 12 2022,38,"Time Inc",yes}
 {sci-fi,440,"Dan Simmons",no}
```

Utilize the `svals` function to get all keys as a set of records:

```
SELECT svals (attributes) FROM products;
```

The result:

```
   svals
-------------
 sci-fi
 202
 Herbert Wells
 yes
 novel
 256
 F. Scott Fitzgerald
 yes
 AD 12 2022
 38
 Time Inc
 yes
 sci-fi
 440
 Dan Simmons
 no
```

Convert hstore data to a set of records

Use the `each` function to extract `hstore` data as a set of records:

```
SELECT
  name,
  (each(attributes)).*
FROM products;
```

The result:

```
      name      |    key    |      value
----------------+-----------+-----------------
 The Time Machine | genre     | sci-fi
 The Time Machine | pages     | 202
 The Time Machine | author    | Herbert Wells
 The Time Machine | freeshipping | yes
 The Great Gatsby | genre     | novel
 The Great Gatsby | pages     | 256
 The Great Gatsby | author    | F. Scott Fitzgerald
 The Great Gatsby | freeshipping | yes
 People         | date      | AD 12 2022
 People         | number    | 38
 People         | publisher | Time Inc
 People         | freeshipping | yes
 Hyperion       | genre     | sci-fi
 Hyperion       | pages     | 440
 Hyperion       | author    | Dan Simmons
 Hyperion       | freeshipping | no
```

Convert hstore data to JSON/JSONB

PostgreSQL provides the `hstore_to_json` and `hstore_to_jsonb` functions to convert `hstore` data to the JSON or JSONB format. The following statement converts the `attributes` column to `JSONB`:

```
SELECT hstore_to_jsonb (attributes) jsonb
FROM products;
```

The result:

```
                                                jsonb
------------------------------------------------------------------------------------------
 {"genre": "sci-fi", "pages": "202", "author": "Herbert Wells", "freeshipping": "yes"}
 {"genre": "novel", "pages": "256", "author": "F. Scott Fitzgerald", "freeshipping": "yes"}
 {"date": "AD 12 2022", "number": "38", "publisher": "Time Inc", "freeshipping": "yes"}
 {"genre": "sci-fi", "pages": "440", "author": "Dan Simmons", "freeshipping": "no"}
```

Create indexes on the hstore field

You can use indexes to reduce the time PostgreSQL takes to look up particular keys and values. The `hstore` type supports GIN and GIST indexing with `@>`, `?`, `?&`, and `?|` operators.

The following statements create the `GIN` and `GIST` indexes on the `attributes` column:

```
CREATE INDEX idx_gin_attributes ON products USING GIN(attributes);

CREATE INDEX idx_gist_attributes ON products USING GIST(attributes);
```

You can also use the `hash` and `btree` indexes on the `attributes` column:

```
CREATE INDEX idx_btree_attributes ON products USING BTREE(attributes);

CREATE INDEX idx_hash_attributes ON products USING HASH(attributes);
```

Bytea data type

Eugenia Kuzina

Contents

To Table of Contents

The bytea hex format

The bytea escape format

Binary string functions and operators

- SQL functions and operators
- Text/binary string conversion functions

The `bytea` data type allows you to store variable-length binary strings. A binary string is a sequence of octets (bytes). Binary strings differ from character strings in the following:

- Binary strings allow storing octets of zero values and other non-printable octets. Character strings do not allow zero octets and any other octet values that are invalid according to the database encoding.
- Operations on binary strings process bytes, the character string processing depends on locale settings.

Binary strings are appropriate for "raw bytes", the character strings are suitable for text.

The `bytea` data type adds 1 or 4 bytes to the binary string size.

`Bytea` supports two formats for input and output: the `hex` format and PostgreSQL historical `escape` format. The input data can be one of these formats. The bytea_output configuration parameter determines the output format, its default value is `hex`.

The bytea hex format

The `hex` format encodes binary data as two hexadecimal digits per byte. It has the following format:

```
'\x<binary string>'
```

Example:

```
SELECT '\xDEADBEEF';
```

The hexadecimal digits can be upper or lower case, whitespaces are permitted between digit pairs and prohibited within a digit pair and at the beginning of the `\x` sequence. The `hex` format is compatible with a wide range of applications and protocols, and it tends to be faster to convert than the `escape` format.

The bytea escape format

The `escape` format is the traditional PostgreSQL format for the `bytea` type. It represents a binary string as a sequence of ASCII characters and converts bytes that cannot be represented as an ASCII character into escape sequences.

To escape an octet, convert it into its three-digit octal value and precede it with a backslash. To add the backslash character to a binary string, use the double backslash (`\\`). While any octet value can be escaped, the values in the table below must be escaped.

Decimal octet value	Description	Escaped input representation	Example	Hex representation
0	zero octet	'\000'	'\000'::bytea	\x00
39	single quote	'' or '\047'	''::bytea	\x27
92	backslash	'\\' or '\134'	'\\'::bytea	\x5c
0 to 31 and 127 to 255	"non-printable" octets	'\xxx' (octal value)	'\001'::bytea	\x01

The requirement to escape "non-printable" octets depends on locale settings.

The `bytea` octets are output in the `hex` format by default.

If you change `bytea_output` to `escape`, PostgreSQL converts "non-printable" octets (decimal octet values from 0 to 31 and from 127 to 255) to equivalent three-digit octal values. Printable octets are output as their standard representation in the client character set. The backslash octet is doubled in the output.

Example:

```
SET bytea_output = 'escape';

SELECT 'abc \144\145\146 \052 \251 \134'::bytea;
```

The result:

```
      bytea
------------------
 abc def * \251 \\
```

TIP

The `escape` format blurs the differences between binary strings and character strings and has unwieldy escape mechanism. It is preferable to use the `hex` format instead of `escape`.

Binary string functions and operators

PostgreSQL provides a lot of functions and operators for binary string manipulations. Many of them are equivalent to the text string functions. This section gives multiple examples. For more information on binary string operators and functions, refer to the PostgreSQL documentation: Binary string functions and operators.

SQL functions and operators

Some of the SQL operators and functions for working with binary strings are described below:

- Use the `||` operator to concatenate two binary strings:

```
SELECT '\x12345e'::bytea || '\x789a00bcde'::bytea;
```

  The result: `\x12345e789a00bcde`.

- The `position` function returns the first starting index of the specified substring in bytes, or zero if the substring is not found:

```
SELECT position('\x5678'::bytea in '\x1234567890'::bytea);
```

  The result: `3`.

- The `substring` function extracts a substring that starts at the specified byte and has the specified length in bytes. The function has the following syntax:

```
substring ( bytes bytea [ FROM start integer ] [ FOR count integer ] ) → bytea
```

  Example:

```
SELECT substring('\x1234567890'::bytea from 3 for 2);
```

  The result: `\x5678`.

Text/binary string conversion functions

PostgreSQL allows you to convert strings between different encodings and transform binary data to text form. The available conversions are listed in the following article: Available character set conversions. You can use the following functions:

- `convert`. The function converts a binary string that contains text in the `src_encoding` encoding to a binary string in the `dest_encoding` encoding.

```
convert ( <bytes> bytea, <src_encoding>, <dest_encoding> ) → bytea
```

  Example:

```
SELECT convert('text_in_utf8', 'UTF8', 'LATIN1');
```

  The result: `\x746578745f696e5f75746638`.

- `convert_from`. The function converts a binary string that contains text in the `src_encoding` encoding to text in the database encoding.

```
convert_from ( <bytes> bytea, <src_encoding> ) → text
```

  Example:

```
SELECT convert_from('\x746578745f696e5f75746638', 'UTF8');
```

  The result: `text_in_utf8`.

- `convert_to`. The function converts a text string in the database encoding to a binary string in the `dest_encoding` encoding.

```
convert_to ( <string> text, <dest_encoding> ) → bytea
```

  Example:

```
SELECT convert_to('some_text', 'UTF8');
```

  The result: `\x736f6d655f74657874`.

- `encode`. The function encodes binary data to a text representation. Supported formats are: base64, `escape`, `hex`.

```
encode ( <bytes> bytea, <format> text ) → text
```

  Example:

```
SELECT encode('\x1445047', 'base64');
```

  The result: `QURQRw==`.

- `decode`. The function decodes binary data from a text string. Supported formats are: base64, `escape`, `hex`.

```
decode ( <string> text, <format> text ) → bytea
```

  Example:

```
SELECT decode('QURQRw==', 'base64');
```

  The result: `\x1445047`.

Contents

To Table of Contents

The tsvector and tsquery formats
- tsvector
- tsquery

The match operator @@
Increase search speed
Rank search results

PostgreSQL provides the functionality to implement the full text search. It allows you to identify documents that match a query and sort them by relevance.

The pattern text search is a part of the SQL standard since the beginning. PostgreSQL provides `LIKE`, `ILIKE` and `~`/`~*` operators to support it, but they lack many essential properties required by modern information systems. The pattern search does not take into account word forms. For example, if you search `TRY`, `TRIED` is not included in the result. It also does not rank search results. When thousands of matching documents are found, more relevant documents may be at the end of the list. Pattern search operators are usually not supported by database support indexes and must process all documents for each search request.

The full text search solves these problems. PostgreSQL has the `tsvector` function to convert a document to the [domain](#) data type a `tsvector` value is a sorted list of distinct [lexemes](#) which are words that have been normalized to make different variations of the same word. For example, also includes `TRY`, `TRIED` that are a user query and create the `tsquery` value from the query text. The `tsquery` type contains normalized lexemes combined with the logical and phrase search operators. PostgreSQL uses [dictionaries](#) to converts text to `tsvector`. The dictionaries may try to determine stop words that should be ignored, and convert different word forms in a single lexeme. The technique allows PostgreSQL to overcome the pattern search limitations.

The tsvector and tsquery formats

This section describes inner formats of the `tsvector` and `tsquery` types.

tsvector

Execute the query below to see in what format PostgreSQL stores data into the `tsvector` type. Use english language dictionaries to combine different word forms in a single lexeme. For this, specify the `english` configuration parameter in the `to_tsvector` function. See [configuration Examples](#) for configuration details:

```
SELECT to_tsvector('english', 'a row satisfies the condition if it returns true.');
```

The result:

```
                to_tsvector
-------------------------------------------
 'condit':5 'return':8 'row':2 'satisfi':3
```

The `to_tsvector` function calls a parser which breaks text into lexems and assigns a type to each token. PostgreSQL uses a list of [dictionaries](#) for each token, the list can vary depending on the token type. The first dictionary that recognizes the token emits one or more normalized lexemes to represent the token. In our example, `satisfies` and `satisfied` are recognized, returns `satisfi`, and `condition` and `conditional` returns `condit`.

The resulting `tsvector` value does not contain the words `a`, `the`, `if` and `it`. These words are recognized as [stop words](#) and are excluded from the result since they occur too frequently to be useful in search.

The numbers in the result determines the word position in the initial string (for `condit`, 5 for `search`, 2 for `row`, and etc.).

Lexemes in the `tsvector` type can have a weight label, which can be `A`, `B`, `C`, or `D`. The default and is not shown in the output.

Example

```
SELECT 'condit:5B return:8C row:2A satisfi:3C true:9D'::tsvector;
```

The result:

```
                   tsvector
-------------------------------------------
 'condit':5B 'return':8C 'row':2A 'satisfi':3C 'true':9
```

Weight labels are typically used to rank entries coming from different parts of a document (title, body, and others).

PostgreSQL also contains the `setweight` function to set weights of `tsvector` entries. The example below uses `setweight` to label the source of each lexeme, and merges the labeled tsvector values using the concatenation operator `||`. For more information on operators, see [Manipulating Documents](#).

```
UPDATE docs SET tsvector_document =
    setweight(to_tsvector(coalesce(title, '')), 'A') ||
    setweight(to_tsvector(coalesce(abstract, '')), 'B') ||
    setweight(to_tsvector(coalesce(content, '')), 'C') ||
    setweight(to_tsvector(coalesce(body, '')), 'D');
```

After you specify lexeme labels, you can use search rank functions described below to assign different priority values for different weight labels.

tsquery

The `tsquery` value type contains lexemes combined with the phrase search operator `<N>` (FOLLOWED BY) and logical operators `&` (AND), `|` (OR), `!` (NOT). Rule the following query to see the `tsquery` value:

```
SELECT to_tsquery('english', 'row & satisfy');
```

The result:

```
 to_tsquery
------------
 'row' & 'satisfi'
```

The result contains lexemes and the logical operator `&`.

The `tsquery` format can also include weight labels.

```
SELECT to_tsquery('english', 'row & satisfy:AB');
```

The result:

```
   to_tsquery
-----------------
 'row' & 'satisfi':AB
```

The `tsquery` lexemes with weight labels match only `tsvector` lexemes with the same weight.

A `tsquery` lexeme can contain `*` to specify prefix matching:

```
SELECT to_tsquery('super:*');
```

This lexeme matches any word in a `tsvector` value that begins with the `super` string.

`to_tsquery` can process the search single-quoted phrases. This is useful, if the configuration includes a thesaurus dictionary that triggers on these phrases.

In addition to `to_tsquery`, PostgreSQL provides the `plainto_tsquery`, `phraseto_tsquery` and `websearch_to_tsquery` functions for converting a query to the `tsquery` data type.

`plainto_tsquery` transforms unformatted text to a `tsquery` value. The function parses and normalizes text and inserts the `&` (AND) operator between words.

`plainto_tsquery` does not recognize operators, weight labels, or prefix-match labels in its input.

Example

```
SELECT plainto_tsquery('english', 'a joined table');
```

The result:

```
 plainto_tsquery
-----------------
 'join' & 'tabl' & 'a'
```

`phraseto_tsquery` parses and normalizes text and inserts the `<N>` (FOLLOWED BY) operator between words. The FOLLOWED BY operator can be represented as `<N>`, where `N` is an integer value that specifies the distance between two lexemes. `<1>` is equivalent to `<2>`. If a phrase contains stop words, they are discarded, but the `<N>` operator shows a non-zero distance. The `phraseto_tsquery` function is useful to search for exserne sequences (phrases), since the FOLLOWED BY operators check lexeme order.

`phraseto_tsquery` does not recognize operators, weight labels, or prefix-match labels in its input.

Example

```
SELECT phraseto_tsquery('english', 'a row satisfies the condition if it returns true.');
```

The result:

```
          phraseto_tsquery
-----------------------------------
 'row' <-> 'satisfi' <2> 'condit' <3> 'return' <-> 'true'
```

`websearch_to_tsquery` uses an alternative syntax to create a `tsquery` value from query text. It supports the following syntax:

- Unquoted text is converted to lexemes separated by the `&` operator.
- Quoted text is converted to lexemes separated by the `<N>` operator.
- `OR` is converted to the `|` operator.
- `-` is converted to the `!` operator.

Example

```
SELECT websearch_to_tsquery('english', '"a row satisfies the condition" if it returns true OR "false" -"index"');
```

The result:

```
                 websearch_to_tsquery
-------------------------------------------------------
 'row' <-> 'satisfi' <2> 'condit' & 'return' <-> 'true' | 'fals' & !'index'
```

The `websearch_to_tsquery` function does not raise syntax errors. You can use it for a raw user-supplied input. `websearch_to_tsquery` does not recognize operators, weight labels, or prefix-match labels in its input.

> **tip**
>
> You can configure a parser, dictionaries, and types of lexems that should be indexed for the `to_tsvector` and `to_tsquery` functions. See [configuration Examples](#).

The match operator @@

PostgreSQL provides the match operator `@@` for the full text search. Create a table to test it:

```
CREATE TABLE documents(
    id SERIAL,
    document_text TEXT
);

INSERT INTO documents (document_text) VALUES
('If the condition is not satisfied, rows are not returned.'),
('A joined table is a table derived from two other tables according to the rules of the particular join type.'),
('Indexes can be added to and removed from tables at any time.'),
('An index scanned on a column that is part of a join condition can also significantly speed up queries with joins.'),
('A row satisfies the condition if it returns true.'),
('Use LIMIT to specify the maximum number of results to return.'),
('If clauses do not specify a value for a certain column must satisfy a boolean expression.');
```

The match operator `@@` returns `true` if `tsvector` value (document) matches a `tsquery` value (query). To use english language dictionaries, specify the `english` configuration parameter in the `to_tsvector` and `to_tsquery` functions.

```
SELECT * FROM documents
    WHERE to_tsvector('english', document_text) @@ to_tsquery('english', 'satisfy');
```

The result includes documents that contain `satisfy` and its modified forms, `satisfies` and `satisfied`:

```
 document_id |                    document_text
-------------+-----------------------------------------------------
           1 | A row satisfies the condition if it returns true.
           7 | If clauses do to specify that the value in a certain column
             must satisfy a boolean expression.
           5 | If the condition is not satisfied, rows are not returned.
```

Another approach is to store the `to_tsvector` output to a separate `tsvector` column and create indexes on it as follows:

```
ALTER TABLE documents
    ADD COLUMN tsvect tsvector
        GENERATED ALWAYS AS (to_tsvector('english',document_text))
        STORED;

CREATE INDEX idx_gin_documents_tsvect
    ON documents USING GIN (tsvect);
```

After the table is updated, use the `@@` field for search:

```
SELECT document_id, document_text FROM documents
    WHERE tsvect @@ to_tsquery('english', 'satisfy');
```

The same rows are returned:

```
 document_id |                    document_text
-------------+-----------------------------------------------------
           7 | If clauses do to specify that the value in a certain column
             must satisfy a boolean expression.
           1 | If the condition is not satisfied, rows are not returned.
           5 | A row satisfies the condition if it returns true.
```

In the excessive column approach, it is not necessary to explicitly specify the text search configuration in queries. Also, searches are faster, since PostgreSQL does not repeat the `to_tsvector` calls to verify index matches.

The expression-style approach is simpler to set up and it requires less disk space since `tsvector` values are not stored explicitly.

Rank search results

When searching in a large database and a lot of matching documents, not all results are equally useful. Usually, more relevant documents may be at the end of the list. To avoid this problem, PostgreSQL provides two predefined functions to rank the results: `ts_rank` and `ts_rank_cd`.

`ts_rank` counts frequency of query lexemes in the document. `ts_rank_cd` computes the cover density ranking for the given document and query. Refer to the following link for details: [Ranking Search Results](#).

These functions have similar signatures:

```
ts_rank([ weights= float4[], ] vector= tsvector, query= tsquery [, normalization= integer ]) returns float4

ts_rank_cd([ weights= float4[], ] vector= tsvector, query= tsquery [, normalization= integer ]) returns float4
```

Where

- `vector=` - the document `tsvector` value.
- `query=` - the query `tsquery` value.
- `weights=` - contains weight values for different weight labels (optional).
- `normalization=` - specifies how a document's length impacts its rank (optional).

Execute the query with the `ts_rank` function without optional parameters:

```
SELECT document_id, ts_rank(tsv, query) AS rank, document_text
FROM documents, to_tsquery('english', 'table') query
WHERE tsv @@ query
ORDER BY rank DESC;
```

The result:

```
 document_id |  rank   |                    document_text
-------------+---------+-----------------------------------------------------
           2 | 0.082743 | A joined table is a table derived from two other tables
             |          |                according to the rules of the particular join type.
           3 | 0.0607927 | Indexes can be added to and removed from tables at any time.
           6 | 0.0607927 | Use LIMIT to specify the maximum number of results to return.
```

The document that contains `table` twice has a higher rank.

The `weights` parameter offers the ability to change the weight of lexemes in the search based on their weight labels. A weight label is a part of the `tsvector` and `tsquery` values and can be `A`, `B`, `C`, or `D`. The following example uses the default values for weights in the following order: `{0.1, 0.2, 0.4, 1.0}`. `{0.1, 0.2, 0.4, 1.0}` are the default values for the parameter. rank functions use the default value `{0.1, 0.2, 0.4, 1.0}`.

Update a `tsv` text value for the third row with the label weight:

```
UPDATE documents SET tsv = setweight(tsv, 'c')
WHERE document_id = 3
RETURNING tsv;
```

The result:

```
                                tsv
-----------------------------------------------------------------
 'ad':3C 'index':1C 'remov':7C 'tabl':9C 'time':12C
```

Execute the query with the specified weight array:

```
SELECT document_id, ts_rank('{0.05, 0.2, 0.4, 1.0}', tsv, query) AS rank, document_text
  FROM documents, to_tsquery('english', 'table') query
  WHERE tsv @@ query
  ORDER BY rank DESC;
```

The result is different. The row, in which the `table` lexeme has the `A` weight label, obtains a higher rank:

```
 document_id |  rank   |                    document_text
-------------+---------+-----------------------------------------------------
           2 | 0.405775 | A joined table is a table derived from two other tables
             |          |                according to the rules of the particular join type.
           3 | 0.0405775 | Indexes can be added to and removed from tables at any time.
           6 | 0.0405775 | Use LIMIT to specify the maximum number of results to return.
```

A longer document has a greater chance to contain a query term. It is reasonable to take into account the document size. A hundred-word document with five instances of a searched word is more relevant than a thousand-word document with five instances. Both rank functions have the `normalization` option that specifies how a document's length impacts its rank. The table below contains the `normalization` values:

Value	Description
0	Ignores the document length. This is the default value
1	Divides the rank by 1 + the logarithm of the document length
2	Divides the rank by the document length
4	Divides the rank by the mean harmonic distance between extents (only for `ts_rank_cd`)
8	Divides the rank by the number of unique words in the document
16	Divides the rank by 1 + the logarithm of the number of unique words in the document
32	Divides the rank by itself + 1

You can use `|` to specify several values, for example, `32|4`. The transformations are applied in the order listed.

Add a normalization option to the query above:

```
SELECT document_id, ts_rank('{0.05, 0.2, 0.4, 1.0}', tsv, query, 4) AS rank, document_text
  FROM documents, to_tsquery('english', 'table') query
  WHERE tsv @@ query
  ORDER BY rank DESC;
```

The result:

```
 document_id |   rank   |                    document_text
-------------+----------+-----------------------------------------------------
           2 | 0.812356 | A joined table is a table derived from two other tables according
             |          |                to the rules of the particular join type.
           3 | 0.0811548 | Indexes can be added to and removed from tables at any time.
           6 | 0.0811548 | Use LIMIT to specify the maximum number of results to return.
```

> **NOTE**
>
> Ranking search results can be an expensive operation since it requires reading the `tsvector` value of each matching document. It can take significant time if PostgreSQL stores `tsvector` values from disk. It is almost impossible to avoid, since search queries often result in large numbers of matches.

Analyze queries

Eugenia Kuzmina

Contents

To Table of Contents

Cache usage

Index usage

PostgreSQL creates an execution plan for each query. Since the resulting plan is critical to performance, PostgreSQL includes a complex planner that chooses the best plan based on an estimated cost of each alternative option. You can use the `EXPLAIN` command to see what plan the planner produces for a query.

Create a table for test examples:

```
CREATE TABLE test_table (field1 integer, field2 text);
```

Add data to the table and execute the `ANALYZE` command to create table statistics after data update:

```
INSERT INTO test_table
  SELECT i, md5(random()::text)
  FROM generate_series(1, 1000000) AS i;
ANALYZE test_table;
```

Call the `EXPLAIN` command to see the query plan.

```
EXPLAIN SELECT * FROM test_table;
```

The result:

```
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on test_table (cost=0.00..18334.00 rows=1000000 width=37)
```

PostgreSQL can read data from a table in several ways. `Seq Scan` reports that PostgreSQL reads data sequentially, block by block.

The `(cost` option contains two values: estimated start-up and total costs of the query execution. The start-up cost has the `0.00` value. This is the cost value before the output phase begins; for example, the cost value required to sort a sort node. The estimated total cost (`18334.00`) shows the cost that is necessary for retrieving all available rows. The `(cost` option does not take into account resources taken to send result rows to the client, because the planner cannot change this value by altering the plan. By default, the `cost` is measured in units of disk page fetches (`seq_page_cost`). The `cost` value most often indicates the consumption of resources, and not the required time, although these values correlate with each other. See Planner Cost Constants for details.

The `rows` option is the estimated number of rows that `Seq Scan` returns.

The `width` option is the estimated average size of an output row in bytes.

When you call `EXPLAIN` PostgreSQL does not execute the query, instead it creates an estimated execution plan based on available statistics. The actual plan can differ. Use the `ANALYZE` parameter to execute the query and get actual statistics.

```
EXPLAIN (ANALYZE) SELECT * FROM test_table;
```

The result:

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Seq Scan on test_table
    (cost=0.00..18334.00 rows=1000000 width=37)
    (actual time=0.005..42.035 rows=1000000 loops=1)
Planning Time: 0.056 ms
Execution Time: 96.757 ms
```

The `actual time` option is the time expended to get the first row and all rows, in milliseconds.

The `rows` option is the actual number of rows that `Seq Scan` returns.

The `loops` option shows how many times PostgreSQL scans the entire table, i.e. executes `Seq Scan` on all table data.

`Execution Time` is the total time to complete the request.

The `EXPLAIN ANALYZE` command executes queries. If you need to get statistics on the modifying `INSERT`, `DELETE`, or `UPDATE` statements and do not change data, use the `BEGIN` and `ROLLBACK` commands to execute `EXPLAIN ANALYZE` inside a transaction block with a rollback.

```
BEGIN;

EXPLAIN (ANALYZE) INSERT INTO test_table
  SELECT i, md5(random()::text)
  FROM generate_series(1, 100) AS i;

ROLLBACK;
```

Cache usage

The `EXPLAIN` command allows you to get information on buffer usage. Utilize the `BUFFERS` parameter for this purpose.

To see correct results, restart ADPG to clear the buffer cache before testing. `BUFFERS`

Execute the `EXPLAIN` command with the `ANALYZE` and `BUFFERS` parameters:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM test_table;
```

The result:

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Seq Scan on test_table
    (cost=0.00..18334.00 rows=1000000 width=37)
    (actual time=1.488..424.291 rows=1000000 loops=1)
  Buffers: shared read=8334
 Planning:
   Buffers: shared hit=47 read=14
 Planning Time: 39.559 ms
 Execution Time: 463.332 ms
```

PostgreSQL reads the table in parts called blocks. The `Buffers: shared read` option is the number of blocks read from the disk. The cache is empty and the table is completely read from the disk, 8334 blocks are read.

Run the same query:

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM test_table;
```

The result:

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Seq Scan on test_table
    (cost=0.00..18334.00 rows=1000000 width=37)
    (actual time=0.037..72.519 rows=1000000 loops=1)
  Buffers: shared hit=32 read=8302
Planning Time: 0.033 ms
Execution Time: 108.673 ms
```

PostgreSQL adds the `Buffers: shared hit` parameter to the result. `Buffers: shared hit` is the number of blocks read from the PostgreSQL cache. If you repeat this query ten times, you will see that PostgreSQL takes more and more data from the cache each time. PostgreSQL adds data to the cache on each request. It reads from the cache faster than from the disk. PostgreSQL increases `Buffers: shared hit` and reduces `Execution Time` on each query execution.

The `shared_buffers` configuration parameter determines the cache size. You can change it via ADCM, see Configuration parameters.

Index usage

Add the `WHERE` clause to the query:

```
EXPLAIN SELECT * FROM test_table WHERE field1 > 750;
```

The result:

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Seq Scan on test_table (cost=0.00..20834.00 rows=999208 width=37)
  Filter: (field1 > 750)
```

The table has no indexes, and PostgreSQL reads each record sequentially (`Seq Scan`).

`Filter: (field1 > 750)` means that each entry is compared with the condition `field1 > 750`. If a `field1` value meets the condition, PostgreSQL inserts the corresponding row into the result.

The `(cost` values is increased. The expected number of result rows (`rows`) is reduced because of the WHERE clause.

Create an index on `field1` and execute the same query:

```
CREATE INDEX ON test_table(field1);
EXPLAIN (ANALYZE) SELECT * FROM test_table WHERE field1 > 750;
```

The result:

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Seq Scan on test_table
    (cost=0.00..20834.00 rows=999279 width=37)
    (actual time=0.152..89.711 rows=999250 loops=1)
  Filter: (field1 > 750)
  Rows Removed by Filter: 750
Planning Time: 0.063 ms
Execution Time: 121.256 ms
```

Only 750 rows were filtered out, query returns more than 99.9% of the table. Despite an index, the planner chose `Seq Scan`.

Utilize the enable_seqscan setting to disable `Seq Scan` and force the query planner to use the index:

```
SET enable_seqscan TO off;
EXPLAIN (ANALYZE) SELECT * FROM test_table WHERE field1 > 750;
```

The result:

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Index Scan using test_table_field1_idx on test_table
    (cost=0.42..36794.81 rows=999279 width=37)
    (actual time=0.064..102.518 rows=999250 loops=1)
  Index Cond: (field1 > 750)
Planning Time: 0.082 ms
Execution Time: 139.922 ms
```

`Index Scan` means that PostgreSQL scans an index. `Index Cond` contains the condition used to find rows.

PostgreSQL uses the index, but it increases the execution time. The planner chose the best plan without index in the previous example.

Enable the `enable_seqscan` setting and modify the query as follows:

```
SET enable_seqscan TO on;
EXPLAIN (ANALYZE) SELECT * FROM test_table WHERE field1 < 750;
```

The result

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Index Scan using test_table_field1_idx on test_table
    (cost=0.42..31.02 rows=720 width=37)
    (actual time=0.041..0.144 rows=749 loops=1)
  Index Cond: (field1 < 750)
Planning Time: 0.074 ms
Execution Time: 0.182 ms
```

The planner uses the `test_table_field1_idx` index to get the best result.

Add an index on `field2`. Since our database stores text fields in UTF8, create the index with the text_pattern_ops operator class.

```
CREATE INDEX ON test_table(field2 text_pattern_ops);
```

Run the following query with the condition for the text field:

```
EXPLAIN (ANALYZE) SELECT * FROM test_table WHERE field2 LIKE 'ah';
```

The result

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Bitmap Heap Scan on test_table
        (cost=2711.24..11912.67 rows=79714 width=37)
        (actual time=7.473..29.434 rows=62382 loops=1)
  Filter: ((field2 ~~ 'ah'::text)
  Heap Blocks: exact=6321
  -> Bitmap Index Scan on test_table_field2_idx1
        (cost=0.00..2691.37 rows=84714 width=0)
        (actual time=6.343..6.344 rows=62382 loops=1)
        Index Cond: ((field2 >=> 'ah'::text) AND (field2 <<= 'ai'::text))
Planning Time: 0.097 ms
Execution Time: 28.520 ms
```

In the `EXPLAIN` command output, the `||` sign and indentation reflect the hierarchical plan tree structure.

`Bitmap Index Scan` collects a bitmap with the positions of records that match the condition. After that, PostgreSQL sequentially scans the target table for these positions within `Bitmap Heap Scan`.

`Bitmap Heap Scan` works effectively with a large number of records that correspond to a condition.

PostgreSQL also supports `Index Only Scan` operation that is used if a query only references indexed columns. Execute a query that returns a single indexed field:

```
EXPLAIN SELECT field1 FROM test_table WHERE field1 < 750;
```

The result

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Index Only Scan using test_table_field1_idx on test_table
  (cost=0.42..29.69 rows=750 width=4)
Index Cond: (field1 < 750)
```

`Index Only Scan` is faster than `Index Scan`, because `Index Only Scan` does not need to read the entire row. `width=4`, except checking the visibility of an index row. Indexes improve performance of different operations, including sorting. Consider the `ORDER BY` clause.

Drop the index for `field1`:

```
DROP INDEX test_table_field1_idx;
```

Sort the table by `field1`:

```
EXPLAIN (ANALYZE) SELECT * FROM test_table ORDER BY field1;
```

The result

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Gather Merge  (cost=143799.84..161833.70 rows=153418 width=37)
              (actual time=9.311..277.651 rows=1000000 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Sort  (cost=142799.82..143034.79 rows=93768 width=37)
           (actual time=9181.127..215 rows=333567 loops=3)
        Sort Key: field1
        Sort Method: external merge  Disk: 18560kB
        Worker 0:  Sort Method: external merge  Disk: 12944kB
        Worker 1:  Sort Method: external merge  Disk: 20504kB
        -> Parallel Seq Scan on test_table
              (cost=0.00..12502.68 rows=93768 width=37)
              (actual time=0.008..29.574 rows=333567 loops=3)
Planning Time: 0.099 ms
Execution Time: 316.971 ms
```

When the planner determines that a parallel query is the fastest execution strategy for a particular query, it creates a plan that includes a `Gather` or `Gather Merge` node (in our case, the `Gather Merge` node). This node includes the execution plan subtree, which is processed in parallel.

The first operation is `Parallel Seq Scan` on `test_table`. The table blocks are divided among the cooperating processes (`Worker 0` and `Worker 1`). `Worker 0` and `Worker 1` scan different consecutive blocks of the table in parallel.

Then, PostgreSQL sorts the result. `Sort Key` contains the sort condition (`field1`). `Sort Method` + `external merge Disk` the system creates temporary files on disk.

The query execution plan above contains several operations and locks complete.

Add an index to the `field1` and execute the same query:

```
CREATE INDEX ON test_table(field1);
EXPLAIN (ANALYZE) SELECT * FROM test_table ORDER BY field1;
```

The result

```
                                                  QUERY PLAN
--------------------------------------------------------------------------------------------------------------
Index Scan using test_table_field1_idx on test_table
    (cost=0.42..34319.95 rows=1000160 width=37)
    (actual time=0.018..186.251 rows=1000160 loops=1)
Planning Time: 0.140 ms
Execution Time: 148.442 ms
```

With an index, the plan contains only the `Index Scan` operation, and as a consequence – the execution time is reduced.

Work with extensions

Eugenia Kuzina

Contents

To Table of Contents

View available extensions

Manage extensions

- Create an extension
- Drop an extension
- Alter an extension

PostgreSQL provides a way to extend the functionality of a database by bundling SQL objects into a package and using them as a unit. This functionality is implemented as PostgreSQL extensions.

Extensions are modules that provide additional functionality: functions, operators, types, etc.

View available extensions

You can use psql to view the installed extensions:

- The `\dx` meta-command lists installed extensions.
- The `\dx+` meta-command displays installed extensions and their associated objects.

The pg_available_extensions view contains the extensions available for installation. Use the following code to list them:

```
SELECT * FROM pg_available_extensions;
```

You can also find the list of available extensions in the Supported extensions article.

The pg_extension catalog stores information about the installed extensions. In ADPG, the `plpgsql` and `pg_stat_statements` extensions are preinstalled in the `postgres` database. When creating a new database, the extensions that are present in the `template1` database are used (by default, only `plpgsql`). See Template Databases.

Utilize the following code to check installed extensions:

```
SELECT * FROM pg_extension;
```

Result:

```
  oid  |      extname       | extowner | extnamespace | extrelocatable | extversion | extconfig | extcondition
-------+--------------------+----------+--------------+----------------+------------+-----------+--------------
 14472 | plpgsql            |       10 |           11 | f              | 1.0        |           |
 16388 | pg_stat_statements |       10 |         2200 | t              | 1.9        |           |
```

Manage extensions

You can create, alter, and drop extensions using corresponding commands.

> **CAUTION**
>
> If you use PL/Perl, PL/PerlU, PL/Python3U, PL/Tcl, or PL/TclU procedural languages that require additional packages, make sure that all packages for corresponding extensions are installed on all ADPG nodes — the leader and all replicas. Otherwise, cluster can be damaged during minor/major upgrade.

Create an extension

`CREATE EXTENSION` loads a new extension into the current database. An extension with the same name cannot be loaded.

PostgreSQL runs the extension script file. The script creates new SQL objects, for example, functions, data types, operators, and index support methods. `CREATE EXTENSION` also records the identifiers of all the created objects, so that they can be dropped if `DROP EXTENSION` is run.

The user who runs `CREATE EXTENSION` becomes the extension owner and the owner of any objects created by the extension script.

If a matching extension is not found, `CREATE EXTENSION` reports that the extension script file does not exist.

If an extension cannot be found, install an appropriate package that contains this extension.

The following code creates the `pg_stat_statements` module, which is included in the *contrib* extension:

```
CREATE EXTENSION pg_stat_statements;
```

For more information on this command, refer to CREATE EXTENSION.

Drop an extension

`DROP EXTENSION` removes extensions from the database. Dropping an extension causes its component objects to be also dropped.

The following code uses the `CASCADE` option to drop an extension, all dependent objects, and objects that depend on the dependent objects:

```
DROP EXTENSION plperlu CASCADE;
```

For additional information, see DROP EXTENSION.

Alter an extension

`ALTER EXTENSION` modifies an extension.

The following code changes the schema of the `hstore` extension to `schema1`:

```
ALTER EXTENSION hstore SET SCHEMA schema1;
```

For more information, refer to ALTER EXTENSION.

Use timescaledb

by Ahmed Sharif

Contents

Overview
Hypertables
- Create an empty hypertable
- Convert a table with data to a hypertable

Compression
Use timescaledb hyperfunctions to analyze data
- first and last
- time_bucket
- histogram

Overview

The timescaledb extension offers additional capabilities for processing time series data. It allows you to perform scalable inserts and complex queries. It also increases the performance of time series data queries.

The package required for the `timescaledb` installation is shipped with ADPG. To use `timescaledb` just run the `CREATE EXTENSION` command:

```
CREATE EXTENSION timescaledb;
```

> **NOTE**
> If the `timescaledb` extension is created in the `template1` database used as the default template, all subsequently created databases will have this extension installed.

ADPG uses the `timescaledb` version 2.15.0. To check it, execute the following query:

```
SELECT extname FROM pg_extension
  WHERE extname = 'timescaledb';
```

Result:

```
 extname
-------------
 timescaledb

(1 row)
```

The `timescaledb` extension adds a number of functions, views, and tables to a database. For more information on `timescaledb` API, see TimescaleDB API reference.

You can use the `\dv+ psql` meta-command to display the various schemas that `timescaledb` has created:

Result:

```
                List of schemas
        Name         |  Owner
---------------------+----------
 _timescaledb_cache  | postgres
 _timescaledb_catalog| postgres
 _timescaledb_config | postgres
 _timescaledb_debug  | postgres
 _timescaledb_functions | postgres
 _timescaledb_internal  | postgres
 public              | postgres
 timescaledb_experimental | postgres
 timescaledb_information | postgres
```

Hypertables

The `timescaledb` extension provides *hypertables* that are a high-level abstraction, extending traditional PostgreSQL tables to operate with temporal data more effectively. Any operation that you can do with regular PostgreSQL tables, you can perform with hypertables. A hypertable simplifies the organization and retrieval of time-series information by providing built-in partitioning based on time intervals. `timescaledb` splits hypertables into **chunks**.

Create an empty hypertable

To generate a hypertable, you need to create an ordinary table and pass it as a parameter to the `create_hypertable` function:

```
CREATE TABLE base_table(
  id uuid,
  time_column timestamptz not null default clock_timestamp(),
  data_column1 int default random()*100,
  data_column2 int default random()*10
);

SELECT create_hypertable('base_table', by_range('time_column'));
```

Result:

```
create_hypertable
-------------------
 (7,t)
```

To truncate the output (in this case `7`) it will be used later in the names of chunks belonging to this hypertable.

The `create_hypertable` function has two required parameters:

- `relation` - an identifier of the table to convert to a hypertable. The `relation` must be of the `regclass` type (`'base_table'` in the example above).
- `dimension` - must be of the *dimension_info* type. To create a `dimension_info` instance, you need to call the `by_range` or `by_hash` function. This parameter specifies conditions according to which a table will be divided into chunks - `by_range('time_column')` in the example above).

Populate the hypertable with data:

```
INSERT INTO base_table(id, time_column) SELECT gen_random_uuid(), generate_series(
now() - INTERVAL '6 months',
now(),
INTERVAL '1 minute'
);
```

Use the `\dv+ psql` meta-command to display information on the `base_table` hypertable:

```
\d+ base_table
```

Result:

```
                                                  Table "public.base_table"
    Column     |           Type           | Collation | Nullable |                  Default                  | Storage | Compression | Stats target | Description
---------------+--------------------------+-----------+----------+-------------------------------------------+---------+-------------+--------------+-------------
 id            | uuid                     |           |          |                                           | plain   |             |              |
 time_column   | timestamp with time zone |           | not null | clock_timestamp()                         | plain   |             |              |
 data_column1  | integer                  |           |          | (random() * '100000'::numeric)::double precision) | plain   |             |              |
 data_column2  | integer                  |           |          | (random() * '100000'::numeric)::double precision) | plain   |             |              |
Indexes:
    "base_table_time_column_idx" btree (time_column DESC)
Triggers:
    ts_insert_blocker BEFORE INSERT ON base_table FOR EACH ROW EXECUTE FUNCTION _timescaledb_functions.insert_blocker()
Child tables: _timescaledb_internal._hyper_7_30_chunk,
              _timescaledb_internal._hyper_7_31_chunk,
              _timescaledb_internal._hyper_7_59_chunk,
              _timescaledb_internal._hyper_7_60_chunk,
              _timescaledb_internal._hyper_7_61_chunk,
              _timescaledb_internal._hyper_7_62_chunk,
              _timescaledb_internal._hyper_7_63_chunk,
              _timescaledb_internal._hyper_7_64_chunk,
              _timescaledb_internal._hyper_7_65_chunk,
              _timescaledb_internal._hyper_7_66_chunk,
              _timescaledb_internal._hyper_7_67_chunk,
              _timescaledb_internal._hyper_7_68_chunk,
              _timescaledb_internal._hyper_7_69_chunk,
              _timescaledb_internal._hyper_7_70_chunk,
              _timescaledb_internal._hyper_7_71_chunk,
              _timescaledb_internal._hyper_7_72_chunk,
              _timescaledb_internal._hyper_7_73_chunk,
              _timescaledb_internal._hyper_7_74_chunk,
              _timescaledb_internal._hyper_7_75_chunk,
              _timescaledb_internal._hyper_7_76_chunk,
              _timescaledb_internal._hyper_7_77_chunk,
              _timescaledb_internal._hyper_7_78_chunk,
              _timescaledb_internal._hyper_7_79_chunk,
              _timescaledb_internal._hyper_7_80_chunk,
              _timescaledb_internal._hyper_7_81_chunk,
              _timescaledb_internal._hyper_7_82_chunk,
              _timescaledb_internal._hyper_7_83_chunk,
              _timescaledb_internal._hyper_7_84_chunk
Access method: heap
```

In the output, you can see that `timescaledb` has created 28 chunks for `base_table`.

You can also specify an interval for chunks. To do this, set an interval in the `by_range` function. It is possible to set it in milliseconds:

```
CREATE TABLE table1 (LIKE base_table);
SELECT create_hypertable('table1', by_range('time_column', 86400000000));
```

Alternatively, use the `INTERVAL` keyword:

```
CREATE TABLE table2 (LIKE base_table);
SELECT create_hypertable('table2', by_range('time_column', INTERVAL '1 day'));
```

When a hypertable is created, `timescaledb` generates indexes on partitioning columns. The `create_default_indexes` parameter controls this behavior. It is an optional parameter, and its default value is `true`. You can cancel index creation by setting its value to `false`:

```
CREATE TABLE table3 (LIKE base_table);
SELECT create_hypertable('table3', by_range('time_column'), create_default_indexes=> false);
```

If you try to call the `create_hypertable` function for a table that is already a hypertable, the exception is raised. `ERROR: table "base_table" is already a hypertable`. You can set an optional parameter `if_not_exists` to `true` to print a warning instead of raising the exception:

```
SELECT create_hypertable('base_table', by_range('time_column'), if_not_exists=> true);
```

Result:

```
create_hypertable
-------------------
 (7,t)
(1 row)


NOTICE:  table "base_table" is already a hypertable, skipping
 create_hypertable
-------------------
 (7,f)
(1 row)
```

Convert a table with data to a hypertable

To create a hypertable from a table with data, specify an optional parameter `migrate_data`. The code below creates a new table and populates it with data:

```
CREATE FUNCTION return_location() RETURNS text AS $$
BEGIN
  RETURN 'location ' || floor(random() * 5+1 )::  text;
END;
$$ LANGUAGE plpgsql;

CREATE TABLE "conditions"(
  time        TIMESTAMP WITH TIME ZONE NOT NULL,
  location    TEXT DEFAULT return_location(),
  temperature  NUMERIC,
  humidity    NUMERIC
);

INSERT INTO conditions(time, temperature, humidity) SELECT generate_series(
  now() - INTERVAL '6 months',
  now(),
  INTERVAL '1 minute'
)as time,
  round(random()*100)  as temperature,
  round(random()*100) as humidity;
```

Convert `conditions` to a hypertable passing the `migrate_data` parameter:

```
SELECT create_hypertable('conditions', 'time', migrate_data => true);
```

Result:

```
NOTICE:  migrating data to chunks
DETAIL:  Migration might take a while depending on the amount of data.
 create_hypertable
-------------------
(14,public,conditions,t)
```

If a converted table contains data, but the `migrate_data` parameter is not passed or set to `false`, an error occurs:

```
ERROR:  table "conditions" is not empty
HINT:  You can migrate data by specifying 'migrate_data => true' when calling this function.
```

Compression

The `timescaledb` extension allows you to compress time series data and reduce chunk size significantly. When you enable compression, the data in the hypertable is compressed chunk by chunk. When the chunk is compressed, multiple records are combined into a single row. This reduces the amount of disk space needed, as a single row occupies less space than many rows. You can use compression features, such as About compression.

To enable compression, follow the steps below:

1. Use the `ALTER TABLE` statement to turn on compression and set compression options:

```
ALTER TABLE conditions SET (
  timescaledb.compress,
  timescaledb.compress_segmentby = 'location'
);
```

2. Call the `add_compression_policy` function to add a compression policy. The following code compresses chunks that are older than 1 hour:

```
SELECT add_compression_policy('conditions', INTERVAL '1 hour');
```

It is possible to decompress a certain chunk using the `decompress_chunk` function, for example, to maximize performance if various OLAL operations need to be performed. You can use the `timescaledb_information.chunks` view to determine a chunk's name and pass it as a parameter to `decompress_chunk`:

```
SELECT decompress_chunk('_timescaledb_internal._hyper_1s_376_chunk');
```

Use the `remove_compression_policy` function to remove compression:

```
SELECT remove_compression_policy('conditions');
```

Use timescaledb hyperfunctions to analyze data

The `timescaledb` extension provides *hyperfunctions* to analyze data. This section describes some of them. The examples in this section utilize the `conditions` table, the creation of which is described above.

first and last

The `first` and `last` functions allow you to get the first or last value respectively of one column as ordered by another. The following example returns the earliest temperature value based on time within each location:

```
SELECT location, first(temperature, time) FROM conditions
  GROUP BY location ORDER BY location;
```

Result:

```
 location | first
----------+-------
 location 1 |    29
 location 2 |    30
 location 3 |    63
 location 4 |    87
 location 5 |    91
```

The code below returns the latest humidity value based on time within a location:

```
SELECT location, last(humidity, time) FROM conditions
  GROUP BY location ORDER BY location;
```

Result:

```
 location | last
----------+------
 location 1 |   15
 location 2 |   49
 location 3 |   47
 location 4 |   98
 location 5 |   49
```

time_bucket

The `time_bucket` function allows you to aggregate data into buckets of time.

The following query calculates the average temperature per day for a specific location:

```
SELECT time_bucket('1 day', time) AS bucket_time, AVG(temperature) AS avg_temperature
  FROM conditions WHERE location = 'location 3'
  GROUP BY bucket_time ORDER BY bucket_time
  LIMIT 5;
```

Result:

```
     bucket_time      |  avg_temperature
----------------------+--------------------
 2024-09-02 00:00:00+03 | 49.85417595628415
 2024-09-03 00:00:00+03 | 49.42329274125436
 2024-09-04 00:00:00+03 | 47.97741107741107
 2024-09-05 00:00:00+03 | 47.797385620915037
 2024-09-06 00:00:00+03 | 51.758241758241752
```

histogram

The `histogram` function calculates how many times a value falls into a specified range (bucket). For example, the query below breaks up the humidity range (`0`-`100`) into 10 buckets, and returns how many times the humidity value falls in each bucket for each location.

```
SELECT location, histogram(humidity, 0, 100, 10)
  FROM conditions GROUP BY location ORDER BY location;
```

The return value is an array containing the specified number of buckets plus two buckets. The first bucket at the head of the array is for values under the lower min bound, and the last bucket is for values greater than or equal to the max bound.

```
  location  |                                  histogram
------------+------------------------------------------------------------------------------
 location 1 | {0,4903,5314,5324,5326,5208,5219,5304,5173,5296,5172,2782}
 location 2 | {0,4838,5358,5414,5158,5232,5199,5354,5353,5338,5375,2677}
 location 3 | {0,4903,5314,5324,5326,5208,5219,5304,5173,5296,5172,2782}
 location 4 | {0,4922,5075,5312,5324,5329,5055,5347,5354,5150,5362,2777}
 location 5 | {0,4892,5264,5289,5294,5058,5291,5150,5338,5319,5358,2660}
```

Values equal to the min bucket value are included in the bucket starting with this value, but values equal to the max value are included in the next bucket. The test data of the example contains 1200 humidity values that equal `100`. In the result, the last (market bucket) contains this value distributed by locations (location 1 - 276, location 2 - 347, location 3 - 266, location 4 - 277, location 5 - 260).

Use PL/pgSQL

Contents

- Overview
- PL/pgSQL structure
- Declarations
  - Variable parameters
  - Aliases
  - Row types
  - Record types
- Execute SQL commands
- Control structures
  - IF/THEN
  - Loops
  - Error conditions
  - Cursors

Overview

PL/pgSQL is a loadable procedural language for the PostgreSQL database system.

The design goal of PL/pgSQL was to create a loadable procedural language that can be used to create functions and trigger procedures, add control structures to the SQL language, and perform complex computations on those functions.

- Functions can be created in terms of any built-in types, or in terms of types defined by the user.
- Functions can be created in terms of any built-in operators or other functions.
- Any set of valid SQL statements can be used.
- Variables of a user's SQL type can be declared.

The variables declared in PL/pgSQL can be of any SQL type, such as `CHAR`, `VARCHAR`, or `NUMERIC`. However, they can also be of `%TYPE` and `%ROWTYPE`, which allow the type to be derived from a table's definition.

Control structures in PL/pgSQL include `IF`, `CASE`, `LOOP`, `WHILE`, `FOR`, and `FOREACH`. Cursors can be opened and traversed, and even returned to the caller.

Functions in PL/pgSQL can be overloaded, so that the same function name can be used for several different functions that accept different numbers or types of arguments.

PL/pgSQL structure

PL/pgSQL statements are structured as blocks of code:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon (`;`). A block that appears within another block must have a semicolon after `END`, as shown above. However, the final `END` that concludes a function does not require a semicolon.

Tip: A common mistake is to write a semicolon immediately after `BEGIN`. This is incorrect and results in a syntax error.

A label is required only if you want to identify the block for use in an `EXIT` statement, or to qualify the names of the variables declared in the block. If you put a label after `END`, it must match the label at the block's beginning.

All key words are case-insensitive. Identifiers are implicitly converted to lowercase unless double-quoted, just as they are in ordinary SQL commands.

Comments work the same way in PL/pgSQL code as in ordinary SQL. A double dash (`--`) starts a comment that extends to the end of the line. A `/*` starts a block comment that extends to the matching occurrence of `*/`. Block comments cannot be nested, but double dashes can be enclosed in a block comment, and a block comment can be used to temporarily comment out text containing double dashes.

Every function must have at least a `BEGIN` block, even if it has no statements.

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50
    RETURN quantity;
END;
$$ LANGUAGE plpgsql;

SELECT somefunc();
```

Result:

```
NOTICE:  Quantity here is 30
NOTICE:  Quantity here is 80
NOTICE:  Outer quantity here is 50
NOTICE:  Quantity here is 50
 somefunc
----------
       50
(1 row)
```

Declarations

All variables used in a block must be declared in the declarations section of the block. (The only exceptions are that the loop variable of a `FOR` loop iterating over a range of integer values is automatically declared as an integer variable, and likewise the loop variable of a `FOR` loop iterating over the results of a cursor is automatically declared as a record variable.)

PL/pgSQL variables can have any SQL data type, such as `INTEGER`, `VARCHAR`, and `CHAR`.

Here are some examples of variable declarations:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

The general syntax of a variable declaration is:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := | = } expression ];
```

`name`	Variable name
`CONSTANT`	If present, the value of the variable cannot be changed after initialization, or else a run-time error occurs
`type`	Variable type
`collation_name`	Name of a collation to use for this variable (see below)
`NOT NULL`	If specified, a run-time error occurs if the variable is assigned a null value
`DEFAULT expression`	The variable's initial value if none is specified in the declaration. If `CONSTANT` is true, an initial value must be specified.

The value of a variable is evaluated each time the variable is initialized.

Declaration function parameters

Parameters passed to functions are named with the identifiers `$1`, `$2`, etc. Optionally, aliases can be declared for `$n` parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
DECLARE
    amount ALIAS FOR $1;
BEGIN
    RETURN amount * 0.06;
END;
$$ LANGUAGE plpgsql;
```

A parameter's function name can be qualified with the function's name, just like any other variable. This can be useful when a parameter has the same name as a table column.

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN sales_tax.subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Aliases

The syntax for `ALIAS` is:

```
name ALIAS FOR target;
```

The following example shows the use of `ALIAS` with `DECLARE`:

```
DECLARE
    prior_id ALIAS FOR id;
```

Copy types

`%TYPE` provides the data type of a variable or a table column.

You can declare variables and function parameters to be of the same type as another variable or table column. You can also use `%TYPE` to declare a variable to be of the same type as a column in a table.

```
DECLARE
    user_id users.id%TYPE;
```

Row types

A variable of a composite type is called a row variable. Such a variable can hold a whole row of a `SELECT` or `FOR` result, so long as that result has a column set that matches the variable's declared type.

The syntax is:

```
name table_name%ROWTYPE;
name composite_type_name;
```

```
CREATE TABLE users (
    id integer,
    name varchar,
    email varchar
);

CREATE FUNCTION get_email_by_id(integer) RETURNS varchar AS $$
DECLARE
    user_id ALIAS FOR $1;
    user_row users%ROWTYPE;
BEGIN
    SELECT * INTO user_row FROM users WHERE id = user_id;
    RETURN user_row.email;
END;
$$ LANGUAGE plpgsql;
```

Record types

Record variables are similar to row-type variables, but they have no predefined structure. They use the structure of the actual row assigned to them by a `SELECT` or `FOR` command. The substructure of a record variable can change each time it is assigned a value. Until a record variable is first assigned to, it has no substructure, and any attempt to access one of its fields will result in a run-time error.

Note that `RECORD` is not a true data type, only a placeholder.

```
CREATE FUNCTION get_email_by_id(integer) RETURNS varchar AS $$
DECLARE
    user_id ALIAS FOR $1;
    user_row RECORD;
BEGIN
    SELECT * INTO user_row FROM users WHERE id = user_id;
    RETURN user_row.email;
END;
$$ LANGUAGE plpgsql;
```

Execute SQL commands

To execute an SQL command that does not return any data, just write the command:

```
INSERT INTO mytable VALUES (1, 'one');
UPDATE mytable SET name = 'two' WHERE id = 2;
DELETE FROM mytable WHERE id = 3;
```

Any PL/pgSQL variable name appearing in the command text is treated as a parameter, and then the current value of the variable is provided as the parameter value at run time. This is called a *prepared statement*. Variable substitution only works in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands.

To process the result of a query, write the query as the data source for a `FOR` loop:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing materialized views...';

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Now "mviews" has one record from cs_materialized_views

        RAISE NOTICE 'Refreshing materialized view %s ...', quote_ident(mviews.mv_name);
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mviews.mv_name) || ' ' || mviews.mv_query;
    END LOOP;

    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Source: Example 43-10 in the docs

Normally the `FOR` loop's variable is a record variable, which is suitable for holding a row of results from most kinds of queries. If the query is guaranteed to return a single column, you can use a simple variable instead, and the value of that column will be assigned to the variable on each iteration.

```
CREATE FUNCTION get_all_user_ids() RETURNS SETOF integer AS $$
DECLARE
    user_id integer;
BEGIN
    FOR user_id IN SELECT id FROM users LOOP
        RETURN NEXT user_id;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT get_all_user_ids();
```

Result:

```
 get_all_user_ids
------------------
                1
                2
                3
(3 rows)
```

Control structures

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

IF/THEN

`IF/THEN` declarations let you perform simple conditional execution. This is the ability to skip a statement or group of statements based on a condition. The `IF` statement has three forms:

- `IF/THEN/END IF`
- `IF/THEN/ELSE/END IF`
- `IF/THEN/ELSIF/THEN/ELSE/END IF`

```
IF boolean-expression THEN
    statements
END IF;
```

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

Example:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

Loops

With the `LOOP`, `EXIT`, `CONTINUE`, `WHILE`, `FOR`, and `FOREACH` statements, you can arrange for your PL/pgSQL program to repeat a series of commands.

`LOOP` defines an unconditional loop that is repeated indefinitely until terminated by an `EXIT` or `RETURN` statement. The optional `label` can be used by `EXIT` and `CONTINUE` statements within nested loops to specify which loop those statements refer to.

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

`EXIT` causes a `LOOP` to end when execution reaches it.

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;
```

`EXIT` can be followed by `WHEN` and a boolean expression.

```
LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;
```

`CONTINUE` causes the current iteration of a loop to end and control to pass to the next iteration.

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

`WHILE` defines a loop that is repeated as long as its boolean expression evaluates to true. The expression is checked just before each entry to the loop body.

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT boolean-expression LOOP
    statements
END LOOP;

WHILE true LOOP
    -- some computations
    EXIT WHEN boolean-expression;
END LOOP;
```

`FOR` defines a loop that iterates over a range of integer values. The variable `name` is automatically created as type `integer` and exists only inside the loop. The expressions for the lower and upper bound of the range are evaluated when the loop is first entered.

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP [ label ];
```

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;
```

`FOREACH` defines a loop that iterates over a set of values. The variable `name` is created as type `integer` and exists only inside the loop.

```
[ <<label>> ]
FOREACH name IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

```
FOREACH x IN ARRAY ARRAY[1,2,3,4,5,6,7,8,9,10] LOOP
    -- x will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;
```

Error conditions

`RAISE` reports messages and raises errors.

```
RAISE [ level ] 'format' [, expression [, ... ]];
```

`level` can be `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, or `EXCEPTION`. The default is `EXCEPTION`, which raises an error (which normally aborts the current transaction); the other levels only generate messages of different priority levels. Whether messages of particular priority are reported to the client, written to the server log, or both is controlled by the `log_min_messages` and `client_min_messages` configuration variables.

`format` is a format string specifying the message text to be reported. The format string can be followed by optional `expression` arguments to be inserted into the message. In the format string, `%` is replaced by the next optional argument's value. Write `%%` to emit a literal `%`.

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

`ASSERT` is a shorthand for checking that a boolean expression is true.

```
ASSERT boolean-expression [, 'message' ];
```

Cursors

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overruns when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since `FOR` loops automatically use a cursor internally to avoid memory problems.) A more interesting reason is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

```
CREATE FUNCTION myfunc(refcursor) RETURNS refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM users;
    RETURN $1;
END;
$$ LANGUAGE plpgsql;

BEGIN;
SELECT myfunc('mycursor');
FETCH ALL IN mycursor;
COMMIT;
```

Result:

```
 myfunc
--------
 mycursor
(1 row)

 id | name | email
----+------+-------
(0 rows)
```

Source: Example 43-11 in the docs

Vue: PL/Python

Jérôme Petazzoni

Contents

- Contents
- PL/Python functions
- Data types
- Set returning functions
- Anonymous code blocks
- Sharing data
- Database access
- Debugging
- Error handling
- Subtransactions
- Trigger functions
- Event triggers

Overview

PL/Python lets you execute Python code in PostgreSQL. Code is executed when you call a function or a procedure, or when a trigger fires.

Python 2 is supported since PostgreSQL 8.0, Python 3 since PostgreSQL 9.1. Python 2 support is deprecated and will be removed in PostgreSQL 14.

Each function gets its own execution environment in the Python interpreter. By default, these environments are isolated from each other, and from anything else running in the interpreter. For instance, modules imported by PL/Python code are only visible to code executed in the same session.

```
$ sudo apt install postgresql-plpython3
```

Use `CREATE EXTENSION` command to activate PL/Python. Add `plpython3u` to `shared_preload_libraries` in `postgresql.conf` to make it available to all databases.

```
CREATE EXTENSION plpython3u;
```

PL/Python functions

Use `CREATE FUNCTION` to define a function using a PL/Python implementation.

```
CREATE FUNCTION my_func(a integer, b integer)
RETURNS integer
AS $$
  return a + b
$$ LANGUAGE plpython3u;
```

The last statement of a function must be a `return` statement (or `yield` for set returning functions). Control flow statements (`if`, `for`, `while`) are supported, as well as exceptions (`try`/`except`/`finally`). Functions can `import` modules from the standard library, or from any package installed in the Python environment.

```
CREATE FUNCTION my_func(a integer, b integer)
RETURNS integer
AS $$
  if a > b:
    return a
  else:
    return b
$$ LANGUAGE plpython3u;
```

Functions can be `STRICT` (automatically return `NULL` if any argument is `NULL`) or `CALLED ON NULL INPUT` (default behavior).

```
CREATE FUNCTION my_func(a integer, b integer)
RETURNS integer
STRICT
AS $$
  return a + b
$$ LANGUAGE plpython3u;
```

Functions can be `VOLATILE` (default), `STABLE` or `IMMUTABLE`.

```
CREATE FUNCTION my_func(a integer, b integer)
RETURNS integer
IMMUTABLE
AS $$
  return a + b
$$ LANGUAGE plpython3u;
```

Data types

Many of the standard scalar types are supported out of the box. Integers, floats, and strings are converted to their Python equivalent. Other types are converted to strings.

Mapping of PostgreSQL types to Python types

PostgreSQL type	Python type
Boolean	`bool`
Smallint, bigint, int	`int`
Real, double	`float`
Numeric	`decimal`
Bytea	`bytes` (Python 3) or `str` (Python 2)
Text	`str` (Python 3) or `unicode` (Python 2)
Array	`list`
Composite types	`dict` (default), or `object` if `composite_type_as_obj` is true
Composite types, inside nested objects	`tuple`

Return type conversions

Return type	Conversion rule
Boolean	Value is converted to a boolean using the rules of Python. Notably, `NULL` will become `False`. Use `RETURNS boolean STRICT` to avoid this.
String	Value is converted to a string using `str` (Python 3) or `unicode` (Python 2).
Sequence (list, tuple, set, generator)	Value must be iterable. Each element will become an item of the array.
Mapping (dict)	Value will become a composite type. Keys must be strings, and will be matched with the composite type's field names.
Object	Value will become a composite type. Attributes will be read to get the composite type's fields.

```
CREATE FUNCTION get_me_an_array()
RETURNS integer[]
AS $$
  return [1, 2, 3]
$$ LANGUAGE plpython3u;

SELECT get_me_an_array();
```

```
 get_me_an_array
-----------------
 {1,2,3}
(1 row)
```

The following one returns `NULL`:

```
CREATE FUNCTION get_me_an_array()
RETURNS integer[]
AS $$
  return [1, 2, 3]
$$ LANGUAGE plpython3u;

SELECT get_me_an_array();
```

```
 get_me_an_array
-----------------

(1 row)
```

Arrays → Lists

Arrays are converted to lists. Arrays are indexed as in Python, but the index does not start from 0 (it starts from the lower bound of the array).

```
CREATE FUNCTION get_an_array()
RETURNS integer[]
AS $$
  return [1, 2, 3]
$$ LANGUAGE plpython3u;

SELECT get_an_array();
```

```
 get_an_array
--------------
 {1,2,3}
(1 row)
```

```
CREATE FUNCTION get_an_array()
RETURNS integer[]
AS $$
  return [1, 2, 3]
$$ LANGUAGE plpython3u;

SELECT get_an_array()[0];
```

```
 get_an_array
--------------

(1 row)
```

```
CREATE FUNCTION get_an_array()
RETURNS integer[]
AS $$
  return [1, 2, 3]
$$ LANGUAGE plpython3u;

SELECT get_an_array()[1];
```

```
 get_an_array
--------------
            1
(1 row)
```

Composite types

Composite types are converted to dictionaries. Keys are the names of the fields in the composite type. Values are converted to their Python equivalent.

```
CREATE TABLE mytable (
    name text UNIQUE NOT NULL,
    value1 integer,
    value2 integer
);
INSERT INTO mytable VALUES ('one', 1, 11);
INSERT INTO mytable VALUES ('two', 2, 22);
INSERT INTO mytable VALUES ('three', 3, 33);

CREATE FUNCTION get_mytable_as_dicts()
RETURNS SETOF mytable
AS $$
  return mytable.fetch(10)
$$ LANGUAGE plpython3u;

SELECT * FROM get_mytable_as_dicts();
```

```
  name   | value1 | value2
---------+--------+--------
 one     |      1 |     11
 two     |      2 |     22
 three   |      3 |     33
(3 rows)
```

To convert rows to dictionaries, use `fetch()` on the result of `execute()`. Then, call `next()` on the result to get the first row.

```
CREATE FUNCTION get_mytable_as_dicts()
RETURNS SETOF mytable
AS $$
  return plpy.execute("SELECT * FROM mytable", 10)
$$ LANGUAGE plpython3u;

SELECT * FROM get_mytable_as_dicts();
```

```
  name   | value1 | value2
---------+--------+--------
 one     |      1 |     11
 two     |      2 |     22
 three   |      3 |     33
(3 rows)
```

Passing dictionaries

The same conversion rules apply when a dictionary has to be passed to a function call.

```
CREATE FUNCTION check_mytable(mytable mytable)
RETURNS boolean
AS $$
  return mytable['value1'] == mytable['value2']
$$ LANGUAGE plpython3u;

SELECT check_mytable(mytable) FROM mytable;
```

```
 check_mytable
---------------
 f
 f
 f
(3 rows)
```

Passing dictionaries with missing fields (default)

Missing fields are passed as `None`.

```
CREATE FUNCTION check_mytable(mytable mytable)
RETURNS boolean
AS $$
  if mytable['value1'] is None:
    mytable['value1'] = 0
  if mytable['value2'] is None:
    mytable['value2'] = 0
  return mytable['value1'] == mytable['value2']
$$ LANGUAGE plpython3u;

SELECT check_mytable(mytable) FROM mytable;
```

```
 check_mytable
---------------
 f
 f
 f
(3 rows)
```

Passing dictionaries with missing fields (error)

Accessing a missing field will raise a `KeyError`.

```
CREATE FUNCTION check_mytable(mytable mytable)
RETURNS boolean
AS $$
  return mytable['value1'] == mytable['value2']
$$ LANGUAGE plpython3u;

SELECT check_mytable(mytable) FROM mytable;
```

```
ERROR:  KeyError: 'value1'
CONTEXT:  Traceback (most recent call last):
  PL/Python function "check_mytable", line 2, in <module>
    return mytable['value1'] == mytable['value2']
PL/Python function "check_mytable"
```

Passing dictionaries with missing fields (strict)

Use `STRICT` to automatically return `NULL` if any argument is `NULL`.

```
CREATE FUNCTION check_mytable(mytable mytable)
RETURNS boolean
STRICT
AS $$
  return mytable['value1'] == mytable['value2']
$$ LANGUAGE plpython3u;

SELECT check_mytable(mytable) FROM mytable;
```

```
 check_mytable
---------------

(1 row)
```

Passing dictionaries with missing fields (get)

Use `get()` to provide a default value.

```
CREATE FUNCTION check_mytable(mytable mytable)
RETURNS boolean
AS $$
  return mytable.get('value1', 0) == mytable.get('value2', 0)
$$ LANGUAGE plpython3u;

SELECT check_mytable(mytable) FROM mytable;
```

```
 check_mytable
---------------
 f
 f
 f
(3 rows)
```

Passing dictionaries with missing fields (in operator)

Use `in` to check if a field is present.

```
CREATE FUNCTION check_mytable(mytable mytable)
RETURNS boolean
AS $$
  if 'value1' not in mytable:
    mytable['value1'] = 0
  if 'value2' not in mytable:
    mytable['value2'] = 0
  return mytable['value1'] == mytable['value2']
$$ LANGUAGE plpython3u;

SELECT check_mytable(mytable) FROM mytable;
```

```
 check_mytable
---------------
 f
 f
 f
(3 rows)
```

Passing dictionaries with missing fields (try/except)

Use `try`/`except` to catch `KeyError`.

```
CREATE FUNCTION check_mytable(mytable mytable)
RETURNS boolean
AS $$
  try:
    return mytable['value1'] == mytable['value2']
  except KeyError:
    return False
$$ LANGUAGE plpython3u;

SELECT check_mytable(mytable) FROM mytable;
```

```
 check_mytable
---------------
 f
 f
 f
(3 rows)
```

Set returning functions

Functions can return multiple rows. Use `RETURNS SETOF` to specify the type of the rows. Use `yield` to return each row.

```
CREATE FUNCTION count_to(n integer)
RETURNS SETOF integer
AS $$
  for i in range(1, n+1):
    yield i
$$ LANGUAGE plpython3u;

SELECT count_to(5);
```

```
 count_to
----------
        1
        2
        3
        4
        5
(5 rows)
```

Returning composite types with OUT parameters

Use `OUT` parameters to define the composite type returned by the function.

```
CREATE FUNCTION count_to_squared(n integer, OUT i integer, OUT square integer)
RETURNS SETOF record
AS $$
  for i in range(1, n+1):
    yield (i, i*i)
$$ LANGUAGE plpython3u;

SELECT * FROM count_to_squared(5);
```

```
 i | square
---+--------
 1 |      1
 2 |      4
 3 |      9
 4 |     16
 5 |     25
(5 rows)
```

Returning composite types with OUT parameters (named)

Use `RETURNS TABLE` to name the composite type returned by the function.

```
CREATE FUNCTION count_to_squared(n integer)
RETURNS TABLE(i integer, square integer)
AS $$
  for i in range(1, n+1):
    yield (i, i*i)
$$ LANGUAGE plpython3u;

SELECT * FROM count_to_squared(5);
```

```
 i | square
---+--------
 1 |      1
 2 |      4
 3 |      9
 4 |     16
 5 |     25
(5 rows)
```

Anonymous code blocks

PL/Python can be executed as a function, with `CREATE FUNCTION`. Anonymous code blocks are supported, using `DO` instead of `CREATE FUNCTION`. `DO` blocks don't take arguments, and don't return anything.

```
DO $$
  plpy.notice("Hello, world!")
$$ LANGUAGE plpython3u;
```

```
NOTICE:  Hello, world!
DO
```

Sharing data

The `GD` dictionary persists between function calls. It is private to the current session.

```
CREATE FUNCTION set_variable(name text, value text)
RETURNS void
AS $$
  GD[name] = value
$$ LANGUAGE plpython3u;

CREATE FUNCTION get_variable(name text)
RETURNS text
AS $$
  return GD.get(name)
$$ LANGUAGE plpython3u;

SELECT set_variable('foo', 'bar');
SELECT get_variable('foo');
```

```
 set_variable
--------------

(1 row)

 get_variable
--------------
 bar
(1 row)
```

The `SD` dictionary is similar to `GD`, but it is private to each function.

```
CREATE FUNCTION set_variable(name text, value text)
RETURNS void
AS $$
  SD[name] = value
$$ LANGUAGE plpython3u;

CREATE FUNCTION get_variable(name text)
RETURNS text
AS $$
  return SD.get(name)
$$ LANGUAGE plpython3u;

SELECT set_variable('foo', 'bar');
SELECT get_variable('foo');
```

```
 set_variable
--------------

(1 row)

 get_variable
--------------

(1 row)
```

Database access

Use `plpy.execute()` to execute a query. Results are returned as a list of dictionaries.

```
CREATE TABLE mytable (
    name text UNIQUE NOT NULL,
    value1 integer,
    value2 integer
);
INSERT INTO mytable VALUES ('one', 1, 11);
INSERT INTO mytable VALUES ('two', 2, 22);
INSERT INTO mytable VALUES ('three', 3, 33);

CREATE FUNCTION get_mytable_as_dicts()
RETURNS SETOF mytable
AS $$
  return plpy.execute("SELECT * FROM mytable", 10)
$$ LANGUAGE plpython3u;

SELECT * FROM get_mytable_as_dicts();
```

```
  name   | value1 | value2
---------+--------+--------
 one     |      1 |     11
 two     |      2 |     22
 three   |      3 |     33
(3 rows)
```

Prepared statements

Use `plpy.prepare()` to prepare a statement. Use `$1`, `$2`, etc. to refer to parameters. Use `plpy.execute()` to execute the prepared statement.

```
CREATE FUNCTION get_mytable_as_dicts(limit integer)
RETURNS SETOF mytable
AS $$
  plan = plpy.prepare("SELECT * FROM mytable LIMIT $1", ["integer"])
  return plpy.execute(plan, [limit])
$$ LANGUAGE plpython3u;

SELECT * FROM get_mytable_as_dicts(2);
```

```
 name | value1 | value2
------+--------+--------
 one  |      1 |     11
 two  |      2 |     22
(2 rows)
```

Subtransactions

You can execute queries in a subtransaction with `plpy.subtransaction()`.

```
CREATE FUNCTION insert_mytable(name text, value1 integer, value2 integer)
RETURNS void
AS $$
  plpy.subtransaction()
  plan = plpy.prepare("INSERT INTO mytable VALUES ($1, $2, $3)")
  plpy.execute(plan, [name, value1, value2])
$$ LANGUAGE plpython3u;

SELECT insert_mytable('four', 4, 44);
```

```
 insert_mytable
----------------

(1 row)
```

Subtransactions are useful when you want to catch errors.

```
CREATE FUNCTION insert_mytable(name text, value1 integer, value2 integer)
RETURNS void
AS $$
  try:
    plpy.subtransaction()
    plan = plpy.prepare("INSERT INTO mytable VALUES ($1, $2, $3)")
    plpy.execute(plan, [name, value1, value2])
  except plpy.SPIError:
    plpy.notice("Caught SPIError")
$$ LANGUAGE plpython3u;

SELECT insert_mytable('four', 4, 44);
SELECT insert_mytable('four', 4, 44);
```

```
 insert_mytable
----------------

(1 row)

NOTICE:  Caught SPIError
 insert_mytable
----------------

(1 row)
```

Set returning functions

Functions can return multiple rows. Use `RETURNS SETOF` to specify the type of the rows. Use `yield` to return each row.

```
CREATE FUNCTION count_to(n integer)
RETURNS SETOF integer
AS $$
  for i in range(1, n+1):
    yield i
$$ LANGUAGE plpython3u;

SELECT count_to(5);
```

```
 count_to
----------
        1
        2
        3
        4
        5
(5 rows)
```

Returning composite types with OUT parameters

Use `OUT` parameters to define the composite type returned by the function.

```
CREATE FUNCTION count_to_squared(n integer, OUT i integer, OUT square integer)
RETURNS SETOF record
AS $$
  for i in range(1, n+1):
    yield (i, i*i)
$$ LANGUAGE plpython3u;

SELECT * FROM count_to_squared(5);
```

```
 i | square
---+--------
 1 |      1
 2 |      4
 3 |      9
 4 |     16
 5 |     25
(5 rows)
```

Returning composite types with OUT parameters (named)

Use `RETURNS TABLE` to name the composite type returned by the function.

```
CREATE FUNCTION count_to_squared(n integer)
RETURNS TABLE(i integer, square integer)
AS $$
  for i in range(1, n+1):
    yield (i, i*i)
$$ LANGUAGE plpython3u;

SELECT * FROM count_to_squared(5);
```

```
 i | square
---+--------
 1 |      1
 2 |      4
 3 |      9
 4 |     16
 5 |     25
(5 rows)
```

Anonymous code blocks

PL/Python can be executed as a function, with `CREATE FUNCTION`. Anonymous code blocks are supported, using `DO` instead of `CREATE FUNCTION`. `DO` blocks don't take arguments, and don't return anything.

```
DO $$
  plpy.notice("Hello, world!")
$$ LANGUAGE plpython3u;
```

```
NOTICE:  Hello, world!
DO
```

Sharing data

The `GD` dictionary persists between function calls. It is private to the current session.

```
CREATE FUNCTION set_variable(name text, value text)
RETURNS void
AS $$
  GD[name] = value
$$ LANGUAGE plpython3u;

CREATE FUNCTION get_variable(name text)
RETURNS text
AS $$
  return GD.get(name)
$$ LANGUAGE plpython3u;

SELECT set_variable('foo', 'bar');
SELECT get_variable('foo');
```

```
 set_variable
--------------

(1 row)

 get_variable
--------------
 bar
(1 row)
```

The `SD` dictionary is similar to `GD`, but it is private to each function.

```
CREATE FUNCTION set_variable(name text, value text)
RETURNS void
AS $$
  SD[name] = value
$$ LANGUAGE plpython3u;

CREATE FUNCTION get_variable(name text)
RETURNS text
AS $$
  return SD.get(name)
$$ LANGUAGE plpython3u;

SELECT set_variable('foo', 'bar');
SELECT get_variable('foo');
```

```
 set_variable
--------------

(1 row)

 get_variable
--------------

(1 row)
```

Database access

Use `plpy.execute()` to execute a query. Results are returned as a list of dictionaries.

```
CREATE TABLE mytable (
    name text UNIQUE NOT NULL,
    value1 integer,
    value2 integer
);
INSERT INTO mytable VALUES ('one', 1, 11);
INSERT INTO mytable VALUES ('two', 2, 22);
INSERT INTO mytable VALUES ('three', 3, 33);

CREATE FUNCTION get_mytable_as_dicts()
RETURNS SETOF mytable
AS $$
  return plpy.execute("SELECT * FROM mytable", 10)
$$ LANGUAGE plpython3u;

SELECT * FROM get_mytable_as_dicts();
```

```
  name   | value1 | value2
---------+--------+--------
 one     |      1 |     11
 two     |      2 |     22
 three   |      3 |     33
(3 rows)
```

Prepared statements

Use `plpy.prepare()` to prepare a statement. Use `$1`, `$2`, etc. to refer to parameters. Use `plpy.execute()` to execute the prepared statement.

```
CREATE FUNCTION get_mytable_as_dicts(limit integer)
RETURNS SETOF mytable
AS $$
  plan = plpy.prepare("SELECT * FROM mytable LIMIT $1", ["integer"])
  return plpy.execute(plan, [limit])
$$ LANGUAGE plpython3u;

SELECT * FROM get_mytable_as_dicts(2);
```

```
 name | value1 | value2
------+--------+--------
 one  |      1 |     11
 two  |      2 |     22
(2 rows)
```

Subtransactions

You can execute queries in a subtransaction with `plpy.subtransaction()`.

```
CREATE FUNCTION insert_mytable(name text, value1 integer, value2 integer)
RETURNS void
AS $$
  plpy.subtransaction()
  plan = plpy.prepare("INSERT INTO mytable VALUES ($1, $2, $3)")
  plpy.execute(plan, [name, value1, value2])
$$ LANGUAGE plpython3u;

SELECT insert_mytable('four', 4, 44);
```

```
 insert_mytable
----------------

(1 row)
```

Subtransactions are useful when you want to catch errors.

```
CREATE FUNCTION insert_mytable(name text, value1 integer, value2 integer)
RETURNS void
AS $$
  try:
    plpy.subtransaction()
    plan = plpy.prepare("INSERT INTO mytable VALUES ($1, $2, $3)")
    plpy.execute(plan, [name, value1, value2])
  except plpy.SPIError:
    plpy.notice("Caught SPIError")
$$ LANGUAGE plpython3u;

SELECT insert_mytable('four', 4, 44);
SELECT insert_mytable('four', 4, 44);
```

```
 insert_mytable
----------------

(1 row)

NOTICE:  Caught SPIError
 insert_mytable
----------------

(1 row)
```

Debugging

Use `plpy.debug()` to send debug messages to the client.

```
CREATE FUNCTION debug_me()
RETURNS void
AS $$
  plpy.debug("Debug message")
$$ LANGUAGE plpython3u;

SELECT debug_me();
```

```
DEBUG:  Debug message
 debug_me
----------

(1 row)
```

Use `client_min_messages` to control which messages are sent to the client.

```
SET client_min_messages TO DEBUG;
SELECT debug_me();
```

```
DEBUG:  Debug message
 debug_me
----------

(1 row)
```

```
SET client_min_messages TO NOTICE;
SELECT debug_me();
```

```
 debug_me
----------

(1 row)
```

Error handling

Use `plpy.error()` to raise an error.

```
CREATE FUNCTION raise_error()
RETURNS void
AS $$
  plpy.error("Error message")
$$ LANGUAGE plpython3u;

SELECT raise_error();
```

```
ERROR:  Error message
CONTEXT:  PL/Python function "raise_error"
```

Use `plpy.fatal()` to raise a fatal error. This will terminate the session.

```
CREATE FUNCTION raise_fatal()
RETURNS void
AS $$
  plpy.fatal("Fatal message")
$$ LANGUAGE plpython3u;

SELECT raise_fatal();
```

```
FATAL:  Fatal message
CONTEXT:  PL/Python function "raise_fatal"
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
```

Use `plpy.spiexceptions` to catch specific exceptions.

```
CREATE FUNCTION insert_mytable(name text, value1 integer, value2 integer)
RETURNS void
AS $$
  try:
    plpy.subtransaction()
    plan = plpy.prepare("INSERT INTO mytable VALUES ($1, $2, $3)")
    plpy.execute(plan, [name, value1, value2])
  except plpy.spiexceptions.UniqueViolation:
    plpy.notice("Caught UniqueViolation")
$$ LANGUAGE plpython3u;

SELECT insert_mytable('four', 4, 44);
SELECT insert_mytable('four', 4, 44);
```

```
 insert_mytable
----------------

(1 row)

NOTICE:  Caught UniqueViolation
 insert_mytable
----------------

(1 row)
```

Trigger functions

You can write trigger functions in PL/Python. Use `RETURNS trigger` to define the function. Trigger data is passed in `TD`.

```
CREATE FUNCTION mytable_trigger()
RETURNS trigger
AS $$
  plpy.notice("TD[event] = " + TD['event'])
  plpy.notice("TD[when] = " + TD['when'])
  plpy.notice("TD[level] = " + TD['level'])
  plpy.notice("TD[new] = " + str(TD['new']))
  plpy.notice("TD[old] = " + str(TD['old']))
$$ LANGUAGE plpython3u;

CREATE TRIGGER mytable_trigger
AFTER INSERT OR UPDATE OR DELETE ON mytable
FOR EACH ROW EXECUTE PROCEDURE mytable_trigger();

INSERT INTO mytable VALUES ('five', 5, 55);
UPDATE mytable SET value1 = 6 WHERE name = 'five';
DELETE FROM mytable WHERE name = 'five';
```

```
NOTICE:  TD[event] = INSERT
NOTICE:  TD[when] = AFTER
NOTICE:  TD[level] = ROW
NOTICE:  TD[new] = {'name': 'five', 'value1': 5, 'value2': 55}
NOTICE:  TD[old] = None
INSERT 0 1
NOTICE:  TD[event] = UPDATE
NOTICE:  TD[when] = AFTER
NOTICE:  TD[level] = ROW
NOTICE:  TD[new] = {'name': 'five', 'value1': 6, 'value2': 55}
NOTICE:  TD[old] = {'name': 'five', 'value1': 5, 'value2': 55}
UPDATE 1
NOTICE:  TD[event] = DELETE
NOTICE:  TD[when] = AFTER
NOTICE:  TD[level] = ROW
NOTICE:  TD[new] = None
NOTICE:  TD[old] = {'name': 'five', 'value1': 6, 'value2': 55}
DELETE 1
```

Event triggers

Event triggers are similar to trigger functions, but they are triggered by DDL events instead of DML events.

```
CREATE FUNCTION myevent_trigger()
RETURNS event_trigger
AS $$
  plpy.notice("TD[event] = " + TD['event'])
  plpy.notice("TD[tag] = " + TD['tag'])
$$ LANGUAGE plpython3u;

CREATE EVENT TRIGGER myevent_trigger
ON ddl_command_start
EXECUTE PROCEDURE myevent_trigger();

CREATE TABLE foo(id SERIAL);
```

```
NOTICE:  TD[event] = ddl_command_start
NOTICE:  TD[tag] = CREATE TABLE
CREATE TABLE
```

Use C-language functions

Contents

- Dynamic loading
- Create a sample function
- Base types in C-language functions
- Version 1 calling conventions
- Basic rules for writing code
- Composite-type arguments
- Return sets
- Polymorphic arguments and return types
- Shared memory and LWLocks
- Writing code
- Compiling and linking dynamic-load modules
- Extension building infrastructure
- Binary compatibility issues
- Compiling code examples
- Server API

Try "table of contents"

A procedural language's call handler for a compiled language (such as C) is supported in the same way as C-language functions are supported: the call handler must be a version-1 function implemented in the language and declared as taking no arguments and returning type `language_handler`. This special pseudo-type identifies the function as a call handler and prevents it from being called directly in SQL commands. For more details about C language call handlers, see `PLTEMPLATE_PLSAMPLE` in `src/pl/plsample`.

Dynamic loading

PostgreSQL provides a general facility for dynamic loading of shared libraries. The C language functions that have been defined in the table `pg_proc` are loaded dynamically as they are called by the server. The way that the server finds the correct shared library file to load for a given function call is specified by the `probin` field of the function's `pg_proc` entry. This field must contain the absolute path to the shared library file. Relative paths are not allowed, nor are searches for the library to be done automatically.

If after the server has started you modify the contents of an existing shared library file (or replace it by installing a new version), the server will load the new contents the next time it is told to open that shared library file.

Tip

It might be useful to exploit this dynamic-loading behavior during development. However, you should not rely on it in production, since there is no mechanism that will guarantee that all backends will stop using a shared library file that has been replaced on disk. For that, you must restart the server.

On macOS, Windows, and ELF-based Unix systems, the directory containing a shared library file is automatically searched if the file is referenced in the `probin` without an explicit directory part. This means that if you put your shared library files in a standard location, you will not need to specify a directory part in the `AS` clause of `CREATE FUNCTION`. You can use the `dynamic_library_path` configuration variable to specify additional directories to search.

On macOS, Windows, and ELF-based Unix systems, the suffix `.dylib` for macOS, `.dll` for Windows, and `.so` for most other systems can be omitted in the `AS` clause of `CREATE FUNCTION`. It is best to follow this practice so that your function definitions will not need to change if you move them to a new system.

You can use the `LOAD` command to explicitly load a shared library file. For example, `LOAD 'mylibname'` will force the server to load the shared library file `mylibname.so` (with the appropriate suffix for your system) if it has not already done so. Unlike loading that happens implicitly when a function is called, `LOAD` will not look in the directories listed in `dynamic_library_path`. You must specify the file name with a directory part, or it will be rejected. `LOAD` is mainly useful for libraries that have no functions to call, but instead change the behavior of the server through "hooking" functions.

On macOS, Windows, and ELF-based Unix systems, the directory containing a shared library file is automatically searched if the file is referenced in the `LOAD` command without an explicit directory part. This behavior is the same as what happens for `CREATE FUNCTION`. For portability, it is best to specify `LOAD` for libraries that you plan to share between multiple systems, and to specify an explicit directory part in each case.

Create a sample function

Let's create a simple C-language function. This function will count the number of times it has been called during the current session.

```
CREATE FUNCTION add_one_and_one()
    RETURNS INTEGER
    AS 'MODULE_PATHNAME'
    LANGUAGE C
    STRICT;
```

The `AS` clause identifies the shared-library file to be loaded as `MODULE_PATHNAME`. The `CREATE FUNCTION` command will substitute the actual path to the shared library file (specified implicitly by `CREATE EXTENSION`) at this point. For further information about the `CREATE FUNCTION` command, see `CREATE FUNCTION`.

The `LANGUAGE` line tells the server that the function is to be executed using the C language handler that is named in the `AS` clause. The `STRICT` keyword tells the server that the function should not be called if any of its inputs are null; in that case, it will simply be assumed to return null. Omitting `STRICT` means that the function must be prepared to be called with null inputs.

Tip

Depending on the implementation details of your function, it might also be safe to omit `STRICT` and instead test for null inputs within the function itself. It is generally faster to declare a function `STRICT` than to write a test for null inputs, because the system can sometimes avoid calling the function at all in null cases.

Now we need a function to call `add_one_and_one`. The easiest choice is a query that just invokes this function:

```
SELECT add_one_and_one();
```

Create a sample function

Let's create a simple C-language function. This function will count the number of times it has been called during the current session.

```
CREATE FUNCTION add_one_and_one()
    RETURNS INTEGER
    AS 'MODULE_PATHNAME'
    LANGUAGE C
    STRICT;
```

The `AS` clause identifies the shared-library file to be loaded as `MODULE_PATHNAME`. The `CREATE FUNCTION` command will substitute the actual path to the shared library file (specified implicitly by `CREATE EXTENSION`) at this point. For further information about the `CREATE FUNCTION` command, see `CREATE FUNCTION`.

The `LANGUAGE` line tells the server that the function is to be executed using the C language handler that is named in the `AS` clause. The `STRICT` keyword tells the server that the function should not be called if any of its inputs are null; in that case, it will simply be assumed to return null. Omitting `STRICT` means that the function must be prepared to be called with null inputs.

Tip

Depending on the implementation details of your function, it might also be safe to omit `STRICT` and instead test for null inputs within the function itself. It is generally faster to declare a function `STRICT` than to write a test for null inputs, because the system can sometimes avoid calling the function at all in null cases.

Now we need a function to call `add_one_and_one`. The easiest choice is a query that just invokes this function:

```
SELECT add_one_and_one();
```

API

PostgreSQL's server API is defined in the following header files:

```
$ sudo apt install pgxs
$ sudo apt install libpq-dev
```

The `PG_FUNCTION_INFO_V1` macro declares the function as a version-1 function. This is required for version-1 functions. It is harmless if you omit it from functions that are not version-1 functions, but it is best to get in the habit of using it.

```
$ sudo apt install pgxs
$ sudo apt install libpq-dev
```

The `PG_MODULE_MAGIC` macro declares that the compiled module is compatible with the server. This is required for all modules

Use PL/Perl

Contents

Jump to:

- Overview
- PL/Perl functions, procedures, and arguments
  - Scalar data transformation
  - Arrays
  - Composite types
- Use SPI::OK* with functions and procedures
- Database access from the PL/Perl code
  - Return a scalar set
  - Return a large row set
  - Use prepared query plans
- Message interactions
- Utility functions

Overview

In this tutorial, we'll look at how to develop stored procedures and user-defined functions in PostgreSQL using Perl. The main advantage of PL/Perl is that it's a mature, well-known, and powerful language. Perl has been around for a long time and has a huge ecosystem of libraries (CPAN) that you can use to extend your code.

Perl is also a good choice for text manipulation. Its regular expressions are powerful and well-integrated into the language. Perl's data structures are flexible and easy to work with. Perl is a dynamic language, so you don't have to worry about compiling your code before you run it.

To install PL/Perl in a database on a local Postgres, run the `CREATE EXTENSION` command as follows:

```
CREATE EXTENSIONplperl;
```

To install PL/Perl in a database on a managed service, such as Amazon RDS, run the `CREATE EXTENSION` command as follows:

```
CREATE EXTENSIONplperl;
```


Note that `plperl` is `plperl` extension is created with `TRUSTED`. Untrusted version is the short-name of autoloaded by default whenever a file.dll is present.

```
plperl.dll
plperlu.dll
```

Trusted PL/Perl code is restricted in what it can do. The `PL/Perl` extension is created by default. Untrusted PL/Perl code is not restricted in what it can do. The `PL/PerlU` extension is created by default.

Trusted	Untrusted
`plperl.dll`	`plperlu.dll`
`plperl.so`	`plperlu.so`
`plperl.dylib`	`plperlu.dylib`
`plperl--1.0.sql`	`plperlu--1.0.sql`
`plperl.control`	`plperlu.control`

Trusted PL/Perl code is restricted in what it can do. Untrusted PL/Perl code is not restricted in what it can do. The `PL/PerlU` extension is created by default. Untrusted PL/Perl code is not restricted in what it can do. The `PL/PerlU` extension is created by default.

For more information, see the Perl on Amazon RDS section.

PL/Perl functions, procedures, and arguments

To create a function in the PL/Perl language, use the `CREATE FUNCTION` command:

```
CREATE FUNCTIONfunction_name(argument_list)
RETURNS return_type
AS $$
  # PL/Perl function body
$$ LANGUAGE plperl;
```

The function body contains a Perl code. Note that the value returned from a function is the value of the last expression evaluated. If you want to return a value earlier, you can use the `return` statement:

```
CREATE FUNCTIONfunction_name(argument_list)
RETURNS return_type
AS $$
  # PL/Perl function body
  return value;
$$ LANGUAGE plperl;
```

To call a function, use the `SELECT` command:

```
SELECTfunction_name(argument_list);
```


To create a procedure, use the `CREATE PROCEDURE` command:

```
CREATE PROCEDUREprocedure_name(argument_list)
AS $$
  # PL/Perl procedure body
$$ LANGUAGE plperl;
```

To call a procedure, use the `CALL` command:

```
CALLprocedure_name(argument_list);
```


Note that `CALL` is used to call procedures, while `SELECT` is used to call functions.

Use PL/Tcl

A Tcyborg Series

Contents

Overview
PL/Tcl functions and arguments
Global data
Database access from PL/Tcl
spi_exec
spi_prepare
spi_execp
quote
subtransaction
elog

Overview

PL/Tcl is a loadable procedural language for the AzPGThreeSQL database system that allows you to use the Tcl language to write functions and procedures.

PL/Tcl offers most of the capabilities that the C language provides with few limitations and you also have the powerful string processing features that are available for Tcl.

One useful extension that is available for PL/Tcl is the module called pltcl_modules which is loaded automatically, if it exists. It contains only a few commands are available to access the database using SPI and to issue messages via elog(). PL/Tcl does not provide the ability to interact with the internal representation of the database server or access the OS with the permissions of the server process. The client can be compiled so that the Tcl commands are not available at all, which makes it reasonably safe for server processes to have it enabled.

Another major limitation is that Tcl functions cannot be used to create input/output functions for new data types.

Sometimes it is desirable to write Tcl functions that are hard to crash or hijack. That is what is called running in a safe interpreter. Tcl 8.0 and all Tcl 8.x versions are designed to run safe interpreters. However, to ensure that safety is not circumvented in some subtle way, the PL/Tcl module must be used with extreme caution: all knowledge of how to quote data and avoid privileged escapes in the Tcl language must be applied if you are converting an untrusted procedure into a trusted one. Also, the Tcl commands that read from or write to files must not be made available, since this would make it impossible to prevent the procedure from accessing arbitrary files on the server machine. Consult the section about configuring PL/Tcl for more information.

To install PL/Tcl and/or PL/TclU in a particular database, use the CREATE EXTENSION command.

```
CREATE EXTENSION pltcl;
CREATE EXTENSION pltclu;
```

Note

If the pltcl or pltclu extension is created in the template1 database cluster-ed all the default templates all subsequently created databases will have this extension installed.

PL/Tcl functions and arguments

To create a function in the PL/Tcl language, use the standard CREATE FUNCTION command:

```
CREATE FUNCTION <funcname> (argument-types) RETURNS return-type AS $$
    # PL/Tcl function body
$$ LANGUAGE pltcl; [ATHELL]
```

The body of the function is a Tcl script. When the function is called, the argument values are passed to the Tcl script as variables named $1 $2 etc. The result is returned from the Tcl code with the [RETURN] statement.

The PL/Tcl function has the same restriction that the language must be specified as pltcl or pltclu.

For example, the following function returns the greater of two integer values:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Note the STRICT keyword. If NULL input values is passed, the function is not called and a NULL result is returned automatically. To cope with NULL inputs in such a function, you can omit STRICT and test the argument variables for NULL.

Another function uses the spi_exec command to execute a database query. Query results are returned in a Tcl array named after the second argument, with the column names used as array subscripts. For example, with the EMP table from the PostgreSQL tutorial:

```
CREATE FUNCTION show_dept_of_emp(integer, integer) RETURNS integer AS $$
    if [catch {
        spi_exec "SELECT * FROM emp WHERE empno = $1"
    } err] {
        return -code error $err
    }
    if {$2 == 0} {
        return [expr {$DEPTNO % 100}]
    } elseif {$2 == 1} {
        return [expr {$DEPTNO / 100}]
    } else {
        return $DEPTNO
    }
$$ LANGUAGE pltcl;
```

Input values to functions are passed to the function as [global] variables. The second and following ones are the attributes names of the composite type. If an attributes is a row has the [NULL] value, it does not appears in the array. The following example uses an array:

```
CREATE TABLE test (
    i int4, b bool, t text, j json, ia _int4,
    ta _text, c char(10), d date, ts timestamp
);
INSERT INTO test(i,b,t,j,ia,ta,c,d,ts) VALUES
('Mrs. Scrimmy','t',
'Mrs. Scrimmy is cute.',
'{"Mrs.Scrimmy is cute."}',
'{1,2,3}','{"Mrs.Scrimmy is cute."}',
'Mrs. Scrimmy','2020-12-25',
'2020-12-25 00:00:00');

CREATE FUNCTION pltclu_record(test) RETURNS boolean AS $$
    if [array exists 1] {
        parray 1
    }
    return 't'
$$ LANGUAGE pltclu;

SELECT pltclu_record(test) FROM test;
```

Result:

```
1(i) = '1'
1(b) = 't'
1(t) = 'Mrs. Scrimmy is cute.'
1(j) = '{"Mrs.Scrimmy is cute."}'
1(ia) = '{1,2,3}'
1(ta) = '{"Mrs.Scrimmy is cute."}'
1(c) = 'Mrs. Scrimmy'
1(d) = '2020-12-25'
1(ts) = '2020-12-25 00:00:00'

 pltclu_record
---------------
 t
(1 row)
```

PL/Tcl functions can also return composite-type results. To do this, the Tcl code should call [RETURN_NEXT] once per row to return an appropriate value for a list of columns. The command [RETURN_NEXT] takes a single argument, which is a list in the format expected by [array set], and can be used directly with the results of [spi_exec] or [spi_execp].

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Here is an example of a function that returns a row from a query:

```
CREATE FUNCTION getfoo(int) RETURNS record AS $$
    spi_exec -array C "SELECT * FROM foo WHERE fooid = $1"
    return_next [array get C]
$$ LANGUAGE pltcl;
```

Result:

```
 fooid | fooname | f1
-------+---------+----
     1 | Mrs. Scrimmy |
     2 | Mrs. Scrimmy |
     3 | Mrs. Scrimmy |
     4 | The Great Gazoo |
     5 | Mrs. Scrimmy |
(5 rows)
```

PL/Tcl functions can also return composite-type results. To do this, the Tcl code should call [RETURN_NEXT] once per row, in return an appropriate value for a scalar type, or a list of columns for a composite type. The command [RETURN_NEXT] takes a single argument, which is a list in the format expected by [array set], and can be used directly with the results of [spi_exec] or [spi_execp].

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Here is an example of a function that returns a row from a query:

```
CREATE FUNCTION getfoo(int) RETURNS record AS $$
    spi_exec -array C "SELECT * FROM foo WHERE fooid = $1"
    return_next [array get C]
$$ LANGUAGE pltcl;
```

Result:

```
 fooid | fooname | f1
-------+---------+----
     1 | Mrs. Scrimmy |
     2 | Mrs. Scrimmy |
     3 | Mrs. Scrimmy |
     4 | The Great Gazoo |
     5 | Mrs. Scrimmy |
(5 rows)
```

PL/Tcl functions can also return sets of records. To do this, the Tcl code should call [RETURN_NEXT] once per row to

Encryption using pgcrypto

Pavel Stehule

Contents

Table Of Contents

- Security Definitions
- Install pgcrypto
- Supported algorithms
- Pgcrypto functions
  - General hashing functions
    - Password hashing functions
  - PGP encryption functions
  - Generate PGP keys with GnuPG
  - Raw encryption functions
  - Random data

Security Definitions

All pgcrypto functions run on the database server. The data and passwords must become pgcrypto and client application's duty over it. You must connect locally or use SSL connections. Alternatively, you can encrypt data inside the client application.

Also, you must trust the administrator to be trustworthy and competent.

According to the SSL version of pgcrypto functions should trust. If any of the algorithms are flaws. They issue security warnings to users' mailing lists.

Install pgcrypto

The pgcrypto is available in the `contrib` subdirectory of PostgreSQL. For more information, see the `Build and Setup` procedure.

The pgcrypto module is included, users with CREATEDB privilege or the current database can install the module.

The following command installs pgcrypto:

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

Supported algorithms

The following algorithms are supported:

- 3DES.
- AES.
- Blowfish (`bf`).
- Cast5 (`cast5`).

Other digest algorithms supported by OpenSSL, except ciphers, which should be explicitly supported:

- Blowfish.
- AES.
- DES/3DES/CAST5.
- Non-OpenSSL.
- PGP symmetric-encryption.
- PGP Public Key encryption.

Pgcrypto functions

The pgcrypto module includes cryptographic functions. You can utilize them to hash data, hash passwords, encrypt data with symmetric and public keys (`PGP`) and then generate non-cryptographic, but good random data.

General hashing functions

digest()

```
digest(data bytea, type text) returns bytea
digest(data bytea, type text) returns bytea
```

Computes a binary hash from the given data. The `type` parameter defines the algorithm to use. Standard algorithms are `md5`, `sha1`, `sha224`, `sha256`, `sha384` and `sha512`. You can also use any supported algorithm. The second form sends `digest()` to generate a binary hash of the `hex` code.

```
CREATE TABLE testdigest (data bytea, hash bytea);

INSERT INTO testdigest VALUES ('\x0123456789abcdef'::bytea);

UPDATE testdigest
    SET hash = digest(data::bytea, 'sha256');
SELECT *
FROM testdigest;
\x0123456789abcdef \x15e2b0d3c33891ebb0f1ef609ec419420c20e320ce94c65fbc8c3312448eb225
```

hmac()

```
hmac(data bytea, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

This function calculates salted hmac. Internally it calculates and calls `digest()` with the specified `key`. The `type` parameter defines the algorithm to use.

`digest()` is similar to `digest()` but it calculates and returns a hash of the concatenated `key` and `data`. It prevents the sources where someone modifies the data and also changes the hash.

```
SELECT hmac('Pchemaster', 'My_key', 'sha256');
```

If the key is longer than the hash block, it is hashed first and the result is used as key.

Password hashing functions

The `crypt()` and `gen_salt()` functions are designed for hashing passwords. The `crypt()` function performs the hashing, and `gen_salt()` prepares the algorithm parameters. The algorithms in `crypt()` differ from the standard hashing algorithms:

- They are slow. So no brute force attack is possible and no rainbow tables can be precalculated.
- They use a random value called the `salt` with the same key passwords have different encrypted passwords.
- They include the algorithm type in the encrypted password, so different algorithms can coexist.
- Some of them are adaptive. When computers get faster, you can fix the algorithm to be slower and keep it compatible with existing passwords.

crypt()

```
crypt(password text, salt text) returns text
```

Calculates the password hash from the given `password` and `salt`. If the stored password is not known, the `gen_salt()` generates a new `salt` value when store a new password.

The following example shows how to set up a new password by `crypt()`:

```
UPDATE table SET password_hash = crypt('new_password', gen_salt('md5'));
```

To authenticate a password, you can compare the stored password hash to the one returned when trying the password:

```
SELECT (password_hash = crypt('entered_password', password_hash)) AS password_match FROM table;
```

gen_salt()

```
gen_salt(type text [, iter_count integer ]) returns text
```

This function generates a new random salt string for `crypt()`. The salt string also defines the algorithm used in `crypt()`. The `type` parameter specifies the hashing algorithm. The optional `iter_count` parameter specifies how many iterations the algorithm should perform.

The `iter_count` parameter specifies the iteration count for the algorithm. The longer it takes to hash the password and the longer to break the password hash. Too long can be annoying because it takes too much CPU time to check a password. If the `iter_count` parameter is omitted, the default iteration count is used. The valid iteration count values are given below; values outside this range cause errors. The default iteration count is chosen for reasonably good security.

Algorithm	Default	Min	Max
`des`	725	1	16777215
`bf`	6	4	31

PGP encryption functions

PGP encryption functions implement the encryption part of the OpenPGP (RFC 4880) standard. They support the symmetric-key and public-key encryption.

An encrypted PGP message consists of 3 parts or **packets**:

- Packet that contains the session key, either symmetric-key or public-key encrypted.
- A packet that contains data encrypted with the session key.
- Packet that contains integrity protection signature for the data packet. Currently only SHA-1 is used for signing. Also, the integrity protection packet can be used without a signature to detect modifications in encrypted data. This is currently implemented always.

Internally, there is a separate packet framework that supports only those packet types that are needed by the current implementation.

The following limitations should be observed:

- Support only one subkey. Although it is common practice to have a master key and subkey for encryption, there is no reason to complicate things.
- It does not support signing. That also means that key validation is not supported either.
- It does not support encrypted packet length. This is a non-issue when encrypting normal files.
- It does not support compression.
- It does not support several consecutive passwords for symmetric-key encryption.

The following encryption algorithms are supported:

- Symmetric-key: AES, 3DES and Blowfish.
- Public-key: AES-128, AES-256 and 3DES.

The data encryption includes the following steps:

- Generate a random session key. Unless requested, the session key is 128 bits long, which is good enough for AES.
- Encrypt the session key with the given public key or password.
- Create a packet from the encrypted session key.
- Prefix the data with random bytes. This is also called `IV` (Initialization Vector).
- Calculate the hash of the prefixed data and add it to the packet.
- Encrypt the data and add it to the packet.

pgp_sym_encrypt()

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

Encrypts the data in `data` with the symmetric-key `psw`. The `options` parameter can be used to change the encryption algorithms. For example, `cipher-algo=aes256`. For more information about supported options, see the section "Options for PGP functions".

The `pgp_sym_encrypt()` function encrypts a text string, and `pgp_sym_encrypt_bytea()` encrypts a binary string. The `options` parameter contains optional options separated from each other with commas. The table below lists the options:

Name	Description	Default value	Possible values
`cipher-algo`	Specifies which algorithm to use.	`aes128`	`bf`, `aes128`, `aes192`, `aes256`, `3des`, `cast5`
`compress-algo`	Specifies whether to compress the data before encryption.	`0`	`0` — no compression `1` — ZIP compression `2` — ZLIB compression
`compress-level`	Specifies the level of compression to use. Only relevant if `compress-algo` is set to `1` or `2`.	`6`	`0` — store only `1` — fastest compression `2` — ... `9` — best compression
`disable-mdc`	Specifies whether or not to omit the SHA-1 integrity protection. SHA-1 protection is on by default; it should only be turned off for backward compatibility. It is safer to upgrade the software than to turn off this protection.	`0`	`0` — integrity protection `1` — no integrity protection
`sess-key`	Specifies whether to use a separate key for the session.	`0`	`0` — don't use a separate key `1` — use a separate key
`s2k-mode`	Specifies the S2K mode to use.	`3`	`0` — without salt `1` — with salt `3` — with salt and many iterations
`s2k-count`	The number of iterations for the S2K algorithm to use. Must be between 1024 and 65011712 inclusive.	A random value between 65536 and 253952.	A value between 1024 and 65011712
`s2k-digest-algo`	Specifies the digest algorithm to use in the S2K calculation.	`sha1`	`md5`, `sha1`
`s2k-cipher-algo`	Specifies the cipher algorithm to use to encrypt the session key.	The `cipher-algo` value is used.	`bf`, `aes`, `aes128`, `aes192`, `aes256`
`unicode-mode`	Specifies whether to convert the text from client_encoding to UTF8 before encryption, and back after decryption. It is useful if the client uses another encoding that is not UTF8. However, it is not necessary if the database is already in UTF8.	`0`	`0` — don't convert `1` — convert

Example:

```
INSERT INTO username, pgp_sym_encrypt(password_hash, 'sec_key') AS password
FROM users WHERE username LIKE 'Admin%';
```

pgp_sym_decrypt()

```
pgp_sym_decrypt(data bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

The function decrypts a public-key encrypted message. The `pgp_sym_decrypt_bytea` function is identical to `pgp_sym_decrypt` except that it returns a binary string. The `options` parameter contains options as in `pgp_sym_encrypt`.

Name	Description	Default value	Possible values
`convert-crlf`	Specifies whether to convert `0x0d0a` (CRLF) to `0x0a` (LF) when decrypting data. Default is `0`. Applies only to `pgp_sym_decrypt`. It should convert `TEXT` files from Windows or Mac to Unix format before encrypting, if this option is not used.	`0`	`0` — don't convert `1` — convert

Example:

```
SELECT username, pgp_sym_decrypt(password_hash, 'sec_key') AS password
FROM users WHERE username LIKE 'Admin%';
```

pgp_pub_encrypt()

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

Encrypts the data in `data` with the public key `key`. The `options` parameter can be used to change the encryption algorithms. For example, `cipher-algo=aes256`. For more information about supported options, see the section "Options for PGP functions".

The `pgp_pub_encrypt()` function encrypts a text string, and `pgp_pub_encrypt_bytea()` encrypts a binary string. The `options` parameter contains optional options separated from each other with commas. The table below lists the options:

Name	Description	Default value	Possible values
`cipher-algo`	Specifies which algorithm to use.	`aes128`	`bf`, `aes128`, `aes192`, `aes256`, `3des`, `cast5`
`compress-algo`	Specifies whether to compress the data before encryption. Compression is not supported with public-key encryption.	`0`	`0` — no compression `1` — ZIP compression `2` — ZLIB compression
`compress-level`	Specifies the level of compression to use. Only relevant if `compress-algo` is set to `1` or `2`.	`6`	`0` — store only `1` — fastest compression `2` — ... `9` — best compression
`convert-crlf`	Specifies whether to convert `0x0a` (LF) to `0x0d0a` (CRLF) when encrypting data. Default is `0`. Applies only to `pgp_pub_encrypt`. It should convert `TEXT` files to Windows or Mac format before encrypting, if this option is not used.	`0`	`0` — don't convert `1` — convert
`unicode-mode`	Specifies whether text data should be converted from the server encoding to UTF8 before encryption and back after decryption. It is useful if the client uses another encoding that is not UTF8. However, it is not necessary if the database is already in UTF8.	`0`	`0` — don't convert `1` — convert

Example:

```
INSERT INTO users (username, password_hash) VALUES ('Admin', pgp_pub_encrypt('password', dearmor('-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.11 (GNU/Linux)

mQGiBEr4HxQRBADvmlr1ZtfJ8voKECfUCRgLCu2tsHgv4+2Ycnfx8ntjUBE4Lpdf
...
=Nmcg
-----END PGP PUBLIC KEY BLOCK-----')));
```

In this example the `dearmor()` function strips away the PGP armor headers.

pgp_pub_decrypt()

```
pgp_pub_decrypt(bytea msg, bytea key [, text psw [, options text ]]) returns text
pgp_pub_decrypt_bytea(bytea msg, bytea key [, text psw [, options text ]]) returns bytea
```

This function decrypts a public-key encrypted message. Use the `pgp_pub_decrypt_bytea` function to decrypt binary data. You can also utilize `pgp_pub_decrypt_bytea` to decrypt text data.

The `psw` parameter is a password for decryption. If you specify a password, you should also specify the corresponding key. If the password is not required, but the key is encrypted, an empty password must be specified. Otherwise, an error is reported. The `options` parameter contains optional options separated from each other with commas. The table below lists the options:

Name	Description	Default value	Possible values
`convert-crlf`	Specifies whether to convert `0x0d0a` (CRLF) to `0x0a` (LF) when decrypting data. Default is `0`. Applies only to `pgp_pub_decrypt`. It should convert `TEXT` files from Windows or Mac format before encrypting, if this option is not used.	`0`	`0` — don't convert `1` — convert

Example:

```
SELECT pgp_pub_decrypt(password_hash, dearmor('-----BEGIN PGP PRIVATE KEY BLOCK-----
...
=Nmcg
-----END PGP PRIVATE KEY BLOCK-----'), 'psw') FROM users WHERE username LIKE 'Admin%';
```

In this example the `dearmor()` function strips away data from the PGP ASCII armor format.

pgp_key_id()

```
pgp_key_id(bytea key) returns text
```

This function returns the key ID of a PGP public or secret key. It also extracts the key ID that is used for encrypting the data, if an encrypted message is passed as a parameter. It will return one of two special key IDs:

- `SYMKEY` — the message is a public-key encrypted, but the key ID is not known.
- `ANYKEY` — the message is a public-key encrypted, but the key ID is removed.

Note, different keys may have the same ID. This is rare, but normal event. The client application should try to decrypt the message with each key to determine which one is appropriate.

armor(), dearmor()

```
armor(data bytea [, keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

These functions add or remove ASCII armor. The armor format is defined in RFC 4880. The armored data consists of a base64 encoding of the binary data, with optional header and footer lines and optional checksum.

The `armor` can optionally be given `keys` and corresponding `values` to be inserted in the header. If the `keys` or `values` parameter is `NULL`, it is ignored. If the `keys` or `values` parameter is not `NULL`, it must be the same length as the other one.

pgp_armor_headers()

```
pgp_armor_headers(data text) returns setof record (key text, value text)
```

This function extracts the armor headers from `data`. The return value is a set of rows with two columns, `key` and `value`. If keys or values contain any non-ASCII characters, they are not encoded.

Generate PGP keys with GnuPG

You can use the `GnuPG` tool to generate PGP keys. Execute the following command to generate a new key:

```
$ gpg --gen-key
```

The terminal asks you to enter a type of key:

```
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?
```

The preferred key type is `RSA (default)`. For RSA encryption, you should create **RSA (sign only)** and **RSA (encrypt only)** keys as master and then add an RSA encryption subkey with `gpg --edit-key command`.

Execute the following command to list keys:

```
$ gpg --list-secret-keys
```

To export a public key in the ASCII armor format:

```
$ gpg -a --export KEYID > public.key
```

To export a secret key in the ASCII armor format:

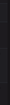
```
$ gpg -a --export-secret-keys KEYID > secret.key
```

Use the `dearmor()` function on these keys before pass them to the PGP functions. Alternatively, if you can handle binary data, omit the `-a` option from the `gpg` command.

Raw encryption functions

These functions run a cipher over the data but do not have any advanced features of the PGP encryption. They have the following main problems:

- Functions use a user key directly as cipher key.
- Functions do not provide any integrity checking, so it can be encrypted data is modified.
- Users manage all encryption parameters themselves, such IV.
- Functions do not have tests.



Caution

Usage of raw encryption functions is not recommended. Use the `PGP encryption functions` instead.

Raw encryption functions use the cipher method specified by `Type`, and encrypt data in cbc mode.

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

The syntax of the `type` string is:

```
algorithm [ -bits ] [ -mode ] [ /pad:padding ]
```

The table below contains possible parameter values:

Name	Values
algorithm	`bf` — Blowfish, `aes` — AES (Rijndael-128, -192 or -256)
mode	`cbc` — next block depends on previous (default), `ecb` — each block is encrypted separately (for testing only)
padding	`pkcs` — data can be of arbitrary length (default), `none` — data should be multiple of cipher block size

The following functions are equivalent:

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

In `encrypt_iv` and `decrypt_iv`, the `iv` parameter specifies the initial value for the CBC mode. This value is ignored for the ECB mode. The initial value is truncated or padded with zeros if it is too short or too long. This is a convenient way if the application does not have a need to care about it.

Random data functions

gen_random_bytes()

```
gen_random_bytes(count integer) returns bytea
```

This function returns a number of `count` bytes of cryptographically strong random bytes. A maximum of 1024 bytes can be received per call to avoid draining the random generator pool.

gen_random_uuid()

```
gen_random_uuid() returns uuid
```

This function returns a version 4 (random) UUID.

Configuration parameters

Contents

- API
  - Controller Server
  - Controller Client
  - Controller Common
  - Controller Server Core (API)
  - Controller Server Core (Internal)
  - Controller Server Core (Plugins)
  - Controller Server Core (Plugins) (API)
  - Controller Server Core (Plugins) (Internal)
  - Controller Server Core (Plugins) (Plugins)
- Core
- Controller
- Server
- Client
- Common

The Controller Server Core (Plugins) (API) section of the API is the API of the Controller Server Core (Plugins) (API) package.

Controller Server Core (Plugins) (API)

The Controller Server Core (Plugins) (API) section of the API is the API of the Controller Server Core (Plugins) (API) package.

Server

Key name	Description	Default value
`Port`	The port to listen on.	`8080`

Server parameters

Server parameters are configured in the `Server` section of the `Controller Server Core (Plugins) (API)` configuration file.

```
Server
    Port 8080
```

In this example, the server will listen on port `8080`.

Server Port Values

Value	Description
`1`	The lowest port number.
`65535`	The highest port number.

Server Port Example

```
Server
    Port 1
```

In this example, the server will listen on port `1`.

Server Port Example

```
Server
    Port 65535
```

In this example, the server will listen on port `65535`.

Server Port Example

```
Server
    Port 8080
```

In this example, the server will listen on port `8080

Supported extensions

Eugenia Kuzina

To Table of Contents

Extensions are special modules that provide additional database functionality by bundling several SQL objects into a package to use them as a single unit. Extensions loaded into a database can operate like features that are built-in. See Work with extensions.

The table below lists the extensions available for installation.

Extension name	Default version	Description
address_standardizer	3.4.1	A rule-based address standardizer that is used to parse an address into constituent elements
address_standardizer_data_us	3.4.1	Contains data for standardizing US addresses
adminpack	2.1	Administrative functions for ADPG/PostgreSQL
amcheck	1.3	Functions to verify table and index consistency
autoinc	1.0	Functions for autoincrementing fields
bloom	1.0	An index access method based on Bloom filters
btree_gin	1.3	Provides support for indexing common data types in GIN
btree_gist	1.7	Provides support for indexing common data types in GiST
citext	1.6	The case-insensitive character string type — `citext`
cube	1.5	The `cube` data type for representing multidimensional cubes
dblink	1.2	Executes queries in a remote database
dict_int	1.0	An example of an add-on dictionary template for full-text search. This example dictionary controls the indexing of integers, allowing such numbers to be indexed and preventing the number of unique words from growing too large
dict_xsyn	1.0	An example of a text search dictionary template for extended synonym processing. This dictionary type replaces words with groups of their synonyms, and makes it possible to search for a word using any of its synonyms
earthdistance	1.1	Calculates great-circle distances on the surface of the Earth
file_fdw	1.0	A foreign-data wrapper for flat file access
fuzzystrmatch	1.2	Determines similarities and distance between strings
hstore	1.8	A data type to store key/value pairs in a single value
insert_username	1.0	Functions for tracking the users who have changed a table
intagg	1.1	An integer aggregator and enumerator (obsolete)
intarray	1.5	Functions and operators for manipulating `NULL`-free arrays of integers
isn	1.2	Data types for international product numbering standards
lo	1.1	Provides support for managing Large Objects (also called LOs or BLOBs)
ltree	1.2	A data type for hierarchical tree-like structures
moddatetime	1.0	Functions for tracking last modification time
old_snapshot	1.0	Allows you to check the server state that is used to implement old_snapshot_threshold
pageinspect	1.12	Allows you to inspect the contents of database pages at a low level
pg_buffercache	1.4	Provides information about the state of the shared buffer cache
pg_freespacemap	1.2	Provides tools for exploring the free space map (FSM)
pg_orphaned	1.0	Handling orphaned files
pg_prewarm	1.2	Preloads relation data into buffer caches
pg_repack	1.5.2	Allows you to reorganize tables
pg_stat_statements	1.10	Provides a means for tracking planning and execution statistics of all SQL statements executed by a server
pg_surgery	1.0	An extension to perform surgery on a damaged relation
pg_trgm	1.6	Provides functions and operators for determining the similarity of alphanumeric text based on trigram matching and index operator classes that support fast searching for similar strings
pg_vector	0.8.0	Allows you to work with vector data
pg_visibility	1.2	Provides a means for examining the visibility map (VM) and page-level visibility information of a table
pgaudit	16.1	Provides detailed session and/or object audit logging via the standard PostgreSQL logging facility
pgauditlogtofile	1.6	An addon to `pgaudit` that redirects audit log lines to an independent file instead of using PostgreSQL server logger
pgcrypto	1.3	Cryptographic functions
pg_walinspect	1.1	Provides SQL functions that allow you to inspect the contents of write-ahead log of a running cluster at a low level
pgrowlocks	1.2	Provides a function to show row locking information for a specified table
pgstattuple	1.5	Provides functionality to show tuple-level statistics
plpgsql	1.0	The PL/pgSQL procedural language. See Use PL/pgSQL
plperl	1.0	The PL/Perl procedural language
plperlu	1.0	The PL/PerlU untrusted procedural language
plpython3u	1.0	The PL/Python3U untrusted procedural language. See Use PL/Python
pltcl	1.0	The PL/Tcl procedural language. See Use PL/Tcl
pltclu	1.0	The PL/TclU untrusted procedural language. See Use PL/Tcl
postgis	3.4.1	PostGIS geometry and geography spatial types and functions
postgis_raster	3.4.1	PostGIS raster types and functions
postgis_sfcgal	3.4.1	PostGIS SFCGAL functions
postgis_tiger_geocoder	3.4.1	PostGIS tiger geocoder and reverse geocoder
postgis_topology	3.4.1	PostGIS topology spatial types and functions
postgres_fdw	1.1	Foreign-data wrapper to access data stored in external PostgreSQL servers
refint	1.0	Functions for implementing referential integrity (obsolete)
seg	1.4	The `seg` data type for representing line segments or floating-point intervals
sslinfo	1.2	Provides information about SSL certificates
tablefunc	1.0	Functions that return tables including `crosstab`
tcn	1.0	Provides a trigger function that notifies listeners of changes to any table on which it is attached
timescaledb	2.15.0	Enables scalable inserts and complex queries for time-series data. See Use timescaledb
timescaledb_toolkit	1.19.0	Provides additional hyperfunctions
tsm_system_rows	1.0	Provides the table sampling method `SYSTEM_ROWS`, which can be used in the `TABLESAMPLE` clause of a `SELECT` command
tsm_system_time	1.0	Provides the table sampling method `SYSTEM_TIME`, which can be used in the `TABLESAMPLE` clause of a `SELECT` command
unaccent	1.1	A text search dictionary that removes accents (diacritic signs) from lexemes
uuid-ossp	1.1	Provides functions to generate universally unique identifiers (UUIDs) using one of several standard algorithms
xml2	1.1	Provides XPath querying and XSLT functionality

ADPG releases

2 September

Contents

To Table of Contents

- 16.3
- 16.2
- 16.1

16.3

16.3.3

Date: 01-1-3000

New Features

Added support for Add a User 1.2 "Voorwaal"

Added a new ADPG Control Agents service with new components: ADPG Control Agent, ADPG Control ADMA Agent

Added the `zypem001` extension

Added the `zypem001tmpfs16` extension

Improvements

Added the `datatron_global_session` parameter to the Datatron service

Bug Fixes

Fixed error when running `./upgrade.sh` on ADPG 16.3 was not ready for PromeNova settings

Fixed error when set certificate: on path: /security/ in the `RouterConfiguration` action to fail to

Fixed installation failure of the ADPG subservice if the `ad_DC_OTP_CE` build does not exist

Fixed reboot issue in mirror upgrade 02 with power `[WRONG]` regs

16.3.2

Date: 03-07-3005

New Features

Added support for SQL with protect Hangout 2.0 cluster action

Added support for Add a Partner REST API

Added DS. for client-server PostgreSQL connections

Added DS. to PgRaccoon

Added DS. to Hot Key

Added DS. to test

Added DS. to pgBackRest

Added HTTPS support for Enterprise Tools

Upgraded Postmaster.exe to 7.3.0.1

16.3.1

Date: 03-07-3005

New Features

Added support for SQL with protect Hangout 2.0 cluster action

Added DS. for client-server PostgreSQL connections

Added DS. to PgRaccoon

Added DS. to Hot Key

Added DS. to test

Added DS. to pgBackRest

Added HTTPS support for Enterprise Tools

Upgraded Postmaster.exe to 7.3.0.1

Improvements

Balancer was removed from the `zhyper` cluster page

Added protection that certificates exist on all nodes

Added protection that certificates are placed outside the ADPG data directory (PGDATA) and are specified with absolute paths

Added check for a general requirement that the ADPG service is a prerequisite of the Hangout page

Upgraded Ansible to 2.7.9

Added check and UI restriction that the `/api/clusterfsck` Pacman tag value is chosen from the list of Pacman members

Bug Fixes

Fixed when some config strings with special characters could be pre-processed by Ansible for safety reasons

Fixed when trust auth for a local connections in ADPG 16.2 was ignored

Fixed when Promethova Global Admin could not log in to ADPG 16

Fixed when a javascript exposer was not stopped after the Monitoring service `Stop` and cluster's `Stop` actions

Fixed when the Dice installation failed with uppercase hostnames

Fixed when Outlines remapping to a new host failed

16.2.3

Date: 13-04-2015

New Features

Added the check that the mount parameter `ppr0` is specified in the `postgresql.conf` field

The `global_cache_protection_full` check was refactored to display a list of all occurred problems

The `DIS_DISK_STRICTER` variable were pre-assigned to ensure that UTILITY.sh accounts created for ADPG will not match the reserved ones in the range 1-499

The `global_cache_protection_full` check was refactored to display a list of all occurred problems

ADPG cluster member names have been simplified to Patroni member names

Improvements

Added the `pg_stat_statements` extension

Added the `pg_repack` extension

Added the `pg_hashids` extension

Added the `tinyserial_feature` extension

Added the `pg_cron` extension

Added the check that the basic value of the `replicationflow` Patroni tag does not refer to the host itself for which the tag is specified

Bug Fixes

Fixed when `tinter_addresses` was incorrectly scanned in case of wrong DNS

Fixed when the **Check** cluster action failed because it expected PG in HotStby file and not FSDNs

Fixed when the **Abort upgrade** action failed when executed after **Upgrade leader** during the major upgrade

Fixed when the **Complete upgrade** action failed if the ADPG service was already stopped before the major upgrade

Fixed when the Patroni restart was used if reload was enough during the **Activate** or **Revert** action

Fixed when Balancer packages were not upgraded during the major upgrade (ADPG)

Fixed when the **Save** button was locked without reason of the cluster's **Configuration** page

Fixed when the **PG_PID** file was replaced with its default value after the **Reconfigure & Restart** action

Fixed when the **Restart** action failed with Patroni logic

Fixed when the **Info** tab of the ADPG service was not updated with new ADPG member names after the major upgrade

Fixed when the hypervisor's Active container the edge is data instead of edge if after the major upgrade to ADPG 16

16.2.2

Date: 15-01-2005

New Features

Added support for ISO ISO 13 Certified edition

Added Patroni member names to the ADPG service **Info** section

Added the `ATTACH_TO_TAPESTORE` property to ADPG major upgrade jobs

The `/etc/sysctl.conf` file is now automatically updated with the recommended PostgreSQL values that are ADCDB structs. ADCDB configuration groups, call terminators

Introduced Patroni tags (see Configuration parameters) in ADPG. Patroni ADPG tags and ability to propagate them to different ADPG nodes

Improvements

Changed the Postgres config file path to /var/log/pg/%y/pgstem

Changed the display name of the `use_inotrue_traverse` parameter to `use_inotrue_payment_log_dir`

Changed the default value of `DO_CHECKPG_ON_start` from `false` to `true`

Simplified the **ADPG Configuration** section in the cluster's **Configuration** page

Changed timeout to wait replica entirely during major upgrade and the **Expand** action

Changed the timeout to wait for the replica to start and rejoin the cluster during major upgrade to ADPG v.Plus

Changed the timeout to wait for the replica to start and rejoin the cluster during major upgrade to ADPG v.Plus

Changed the **Abort upgrade** job REST API check that at least one `ONDUTY` replica was still alive and existed during major upgrade to ADPG v16.x.x

16.2.1

Date: 17-10-2004

New Features

Upgraded the PostgreSQL version to 16.3

Added support for Ubuntu 22.04

Upgraded Patroni to 3.2.2

Upgraded etcd to 3.5.11

Upgraded HAProxy to 2.9.6

Upgraded vip-manager to 2.6.0

Upgraded Grafana to 10.2.8

Upgraded Pgbouncer to 1.20.1

Added the pg_wait_sampling extension

Added the pgaudit extension

- watcher_daemon.ini 0.4.1
- watcher_daemon.bin_dbs.ini 0.4.1
- three_pair 1.7
- hyuuumeed 1.3
- pogingpeed 1.3
- qq_ndfilecache 1.4
- qq_ovd_daemons 1.13
- emoji 0.4.1
- pongq_task 1.4.2
- emoji_sfupt 0.3.1
- pongq_type_genuate 1.4.1
- emoji_topology 0.4.1

Improvements

Added dynamic URLs to ADPG services

Removed the **Run SQL** action of the ADPG service

14.3

14.3.0

Date: 17-10-2004

New Features

Upgraded `pgpatroni` to 1.32.5

Upgraded `transactweb` to 1.14.2

Upgraded `zypfd` to 0.19.1

Improvements

Added GUI to the ADPG cluster

A Kubernetes ADPG environment is accessible in other Kubernetes products

Changed the **Enable maintenance mode** and **Disable maintenance mode** action names to Pause Patroni and Resume Patroni respectively

Improved DC upgrade with Patroni Pause/Resume

Added the `tutorialrun_ppl_ctl_0` to create a variable logon to Kingdom

Created the default value of the `NOT_CHECK_RIGHT` PostgreSQL parameter to 128 MB

Set `log_format = zypfd` in ADPG temporary template

Bug Fixes

Fixed when Grafana lost the connection to a database if the ADPG service port had been changed.

Fixed when store.io creation failed if the ADPG service was without port one, such

Fixed when stop-monitoring agent could not start in case of a non-standard ADPG cluster name

Fixed when the **Account status/action** cluster action did not update all nodes

Fixed when the **Reconfigure & Restart** action of the Balancer service failed after changing the Grafana port

Fixed when Grafana admin's password was not changed after the password for an external cluster

Fixed when pgBadger failed if ADPG has an extension with dash (e.g. `soft-total`)

14.2

14.2.1

Date: 13-04-2004

New Features

Added the ability to `upgrade.sh` using the functionality of the `/pgbin/bin/` utility. The `upgrade.sh` **Archive timeout(s)**, **Backup status**, **Backup status**, **Backup type**, **Backup type**, **Backup type

ADPG known issues

Eugenia Kuzina

To Table of Contents

Issue	From	Fixed	Workaround
After SSL certificates expire, the ***Stop*** and ***Start*** cluster actions fail	16.3.4	—	Update certificates and store them at the same paths. Run the ***Stop*** cluster action and then the ***Start*** cluster action
The ***Expand*** action of the ADPG service fails if SSL is enabled with the ***Manage SSL*** cluster action	16.3.4	—	Follows the steps below: 1. Run the ***Manage SSL*** action to turn off the ***Enable SSL*** toggle button. 2. Expand the ADPG service to a new host. 3. Put SSL keys and certificates to the new ADPG host according to the paths specified in the ***Manage SSL*** action window. 4. Run the ***Manage SSL*** action to switch on the ***Enable SSL*** toggle button.
The ***Add/remove components*** action of the Monitoring service that is executed to expand Grafana to a new host fails with the message `No package matching 'admprom-grafana-9.5.8' found available, installed or updated`	16.3.3	—	■ ADPG EE: first, expand the ADPG, Etcd, and Chrony services to a new host, then run the ***Add/remove components*** action to expand Grafana. ■ ADPG CE: first, expand the Chrony service to a new host, then run the ***Add/remove components*** action to expand Grafana.
After the ***Abort upgrade*** action in case of reverting the major upgrade 14 → 16, the Monitoring service is marked with the red flag, the `Service has an issue with config` error message is displayed, and the ***Save*** button is highlighted in red in the configuration section of the Metrics storage service	16.3.3	—	Resave the configuration without changing the values
The ***Check*** actions of the cluster and the Monitoring service will not fail if any Monitoring subjobs fail	16.3.3	—	On the ***Jobs*** page, expand all Monitoring sub-jobs to identify potential problems
After major upgrade from ADPG 14 to ADPG 16 cluster's ***statuschecker*** is marked with yellow, while all cluster components, hosts, and services are green. Reinstalling ***statuschecker*** does not help	16.3.3	—	Restart the ADCM container (and the Postgres container, if an external database is used)
If connecting with pgbouncer user to `psql`, it causes the warning `Some psql features might not work` due to the 1.23.0 pgbouncer version	16.3.1	—	You can safely disregard this warning
An error occurs if you remove (with 🗑) the Monitoring service mistakenly mapped to a host with an unsupported OS	14.3.6	—	Remap Monitoring components to another host with a supported OS (CE: CentOS, Ubuntu; EE: CentOS, Alt Linux, Astra Linux, Ubuntu, RED OS). After this operation, you can successfully remove the Monitoring service from a cluster
The ADPG cluster installation fails if one of the cluster host name matches the reserved service names: ■ `adpg` ■ `balancer` ■ `chrony` ■ `etcd` ■ `prometheus_monitoring`	14.3.6	—	Recreate the cluster with a hostname that is not the exact name of a service. For example, in the case of `adpg`, rename the host to `adpg1`, `adpg2`, or another non-reserved word
The port that PgBouncer listens on cannot be changed for a running cluster (the `listen_port` PgBouncer parameter of the ADPG service)	14.3.6	—	After changing the port, execute the Reconfigure & Restart action of the ADPG service
If you add nodes to expand the Etcd service, new nodes are not included in the ADPG cluster configuration	14.3.3	—	To solve the problem, execute the ***Reconfigure & Restart*** action of the ADPG and Balancer services
The Upgrade cluster action used for upgrade from ADPG 14.3.8 to ADPG 16 is not stable when moving replicas. Only ADPG clusters with a single-host ADPG service (without replicas) can be major-upgraded with this action	16.3.1	16.3.2	Use logical backup/restore with pg_dump and `pg_restore` utilities to move your data from ADPG 14 to ADPG 16
If you use a non-default port of the ADPG service (other than 5432), the stanza creation is failed during backup repo configuration	14.3.7	16.3.1	Switch to the default ADPG service port — 5432
During ADPG installation on the Astra "Orel" 1.7.x operating system the following error may occur: `invalid locale name "en_US.UTF-8"`	14.3.6	16.3.4	Add the `en_US.UTF-8` locale
Since the new built-in monitoring system was implemented in 14.3.4, ADPG no longer supports the integration with the old imported Monitoring cluster, and therefore each minor upgrade 14.x → 14.y resets the ***Enable monitoring*** setting to `false`	14.2.1	16.3.1	You have to configure the monitoring system and set ***Enable monitoring*** to `true` manually
Upgrade of an ADPG cluster from a previous version to version 14.3.2 does not preserve the settings specified in the ***postgresql.conf custom section*** configuration field of the ADPG service	14.2.1	14.3.2	If you need to upgrade of an ADPG cluster from the versions 14.2.1 - 14.3.1, you have to restore the ***postgresql.conf custom section*** contents manually