

Administrative Tasks Monitoring



Copyright

© Postgres Professional, 2017–2026

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Aleksei Beresnev

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Cover photo by Oleg Bartunov (Nepal peak, 7143 m, Lhonak)

Use of course materials

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

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

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

Topics



OS Tools

Server Statistics

Server Message Log

External Monitoring Systems

Processes

ps, pgrep...

update_process_title parameter for updating the status of processes

cluster_name parameter for setting the cluster name

Resource usage

iostat, vmstat, sar, top...

Disk space

df, du, quota...

PostgreSQL runs on an operating system and to a certain extent depends on its configuration.

PostgreSQL process information is accessible via OS tools. The server parameter *update_process_title* (on by default) displays the state of each process next to its title. The *cluster_name* parameter specifies the instance name used to identify it among running processes.

Various tools are available to monitor the use of system resources (CPU, RAM, disks) in Unix: iostat, vmstat, sar, top, etc.

Disk space monitoring is also necessary. The space occupied by the database on disk can be viewed both from the database itself (see the Data Organization module) and from the OS (with the du command). The amount of disk space available is also displayed with the df command in the OS. If disk quotas are used, they must also be taken into account.

The tools and approaches to monitoring differ significantly between various OS and file systems, so we will not discuss them in detail.

<https://postgrespro.com/docs/postgresql/18/monitoring-ps>

<https://postgrespro.com/docs/postgresql/18/diskusage>



Current Activities

Monitoring of Command Execution

Cumulative Statistics

System Processes

Additional Statistics

There are two primary sources of information about the state of the system. The first one is statistical information collected by PostgreSQL and presented via `pg_stat_*` views.

Current Activities



Views

<code>pg_stat_activity</code>	current per-process activities, including waits and locks
<code>pg_locks</code>	locks details

Settings

`track_activities = on`

The current activities of all backends and background processes are displayed in the `pg_stat_activity` view. This view depends on the `track_activities` parameter (enabled by default).

Another important view that shows current locks is called `pg_locks`.

Current Activities

Let's imitate a scenario when one process blocks another, and then figure it out using system views.

Create a table with one row in a separate database:

```
=> CREATE DATABASE admin monitoring;
```

CREATE DATABASE

```
=> \c admin monitoring
```

You are now connected to database "admin monitoring" as user "student".

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

CREATE TABLE

```
=> INSERT INTO t VALUES(42);
```

INSERT 0 1

Start two sessions, one of which changes the table and does not complete the transaction:

```
student$ psql -d admin_monitoring
```

```
=> BEGIN;
```

BEGIN

```
=> UPDATE t SET n = n + 1;
```

UPDATE 1

And the other tries to change the same row and gets blocked:

```
student$ psql -d admin monitoring
```

```
=> UPDATE t SET n = n + 2;
```

View data about backend processes:

```
=> SELECT pid, query, state, wait_event, wait_event_type, pg_blocking_pids(pid)
FROM pg_stat_activity
WHERE backend type = 'client backend' \gx
```

```

-[ RECORD 1
]-----+-----
pid          | 12982
query        | SELECT pid, query, state, wait_event, wait_event_type,
pg_blocking_pids(pid)+
              | FROM pg_stat_activity
+
              | WHERE backend_type = 'client backend'
state        | active
wait_event   |
wait_event_type |
pg_blocking_pids | {}
-[ RECORD 2
]-----+-----
pid          | 13094
query        | UPDATE t SET n = n + 1;
state        | idle in transaction
wait_event   | ClientRead
wait_event_type | Client
pg_blocking_pids | {}
-[ RECORD 3
]-----+-----
pid          | 13191
query        | UPDATE t SET n = n + 2;
state        | active
wait_event   | transactionid
wait_event_type | Lock
pg_blocking_pids | {13094}

```

The state “idle in transaction” means that the session has started a transaction, but isn’t doing anything at the moment, and the transaction isn’t closed. This could become a problem if the situation comes up regularly (for example, because of poor application code or driver errors), because an open transaction holds a data snapshot and prevents vacuuming.

The administrator has a parameter `idle_in_transaction_session_timeout` at their disposal to force sessions to close after their transaction is idle for a certain period of time. The `idle_session_timeout` parameter forcibly terminates sessions that remain idle beyond the specified duration.

You can also terminate a blocking session manually. First, you need the blocked process ID. The function `pg_blocking_pids` can help you with that:

```
=> SELECT pid AS blocked_pid
FROM pg_stat_activity
WHERE backend_type = 'client backend'
AND cardinality(pg_blocking_pids(pid)) > 0;
```

```
blocked_pid
-----
13191
(1 row)
```

Instead, you can use a query on the locks table. It will return two rows in this case: one transaction has been granted the lock, while the other is waiting for it.

```
=> SELECT locktype, transactionid, pid, mode, granted
FROM pg_locks
WHERE transactionid IN (
  SELECT transactionid FROM pg_locks WHERE pid = 13191 AND NOT granted
);
```

locktype	transactionid	pid	mode	granted
transactionid	758	13191	ShareLock	f
transactionid	758	13094	ExclusiveLock	t

(2 rows)

Generally, you have to keep the lock type in mind.

Running query can be cancelled with the `pg_cancel_backend` function. The transaction is idle in our case, so we use the `pg_terminate_backend` to terminate the session:

```
=> SELECT pg_terminate_backend(b.pid)
FROM unnest(pg_blocking_pids(13191)) AS b(pid);
```

```
pg_terminate_backend
-----
t
(1 row)
```

The `unnest` function is necessary because `pg_blocking_pids` returns an array of process IDs that block the specified process. There is only one in our examples, but there can be multiple.

Locks are discussed in more detail in the DBA2 course.

Check the backends:

```
=> SELECT pid, query, state, wait_event, wait_event_type
FROM pg_stat_activity
WHERE backend_type = 'client backend' \gx
```

- [RECORD 1] -	
pid	12982
query	SELECT pid, query, state, wait_event, wait_event_type+ FROM pg_stat_activity WHERE backend_type = 'client backend' +
state	active
wait_event	
wait_event_type	

- [RECORD 2] -	
pid	13191
query	UPDATE t SET n = n + 2;
state	idle
wait_event	ClientRead
wait_event_type	Client

Only two remain, and the blocked one has completed its transaction successfully.

The `pg_stat_activity` view shows the information about backend and system background processes running on the instance:

```
=> SELECT pid, backend_type, backend_start, state
```

```
FROM pg_stat_activity
```

```
ORDER BY pid;
```

pid	backend_type	backend_start	state
12859	io worker	2026-10-19 23:00:49.565742+03	
12860	io worker	2026-10-19 23:00:49.566603+03	
12861	io worker	2026-10-19 23:00:49.567435+03	
12862	checkpointer	2026-10-19 23:00:49.569449+03	
12863	background writer	2026-10-19 23:00:49.571066+03	
12865	walwriter	2026-10-19 23:00:49.667769+03	
12866	autovacuum launcher	2026-10-19 23:00:49.668368+03	
12867	logical replication launcher	2026-10-19 23:00:49.671512+03	
12982	client backend	2026-10-19 23:00:57.452035+03	active
13191	client backend	2026-10-19 23:00:58.089816+03	idle

(10 rows)

Compare that to what the OS reports:

```
student$ sudo head -n 1 /var/lib/postgresql/18/main/postmaster.pid
```

```
12858
```

```
student$ ps -o pid,command --ppid 12858
```

```
PID COMMAND
12859 postgres: 18/main: io worker 0
12860 postgres: 18/main: io worker 1
12861 postgres: 18/main: io worker 2
12862 postgres: 18/main: checkpointer
12863 postgres: 18/main: background writer
12865 postgres: 18/main: walwriter
12866 postgres: 18/main: autovacuum launcher
12867 postgres: 18/main: logical replication launcher
12982 postgres: 18/main: student admin_monitoring [local] idle
13191 postgres: 18/main: student admin_monitoring [local] idle
```



Views

<code>pg_stat_progress_analyze</code>	ANALYZE
<code>pg_stat_progress_vacuum</code>	VACUUM, including autovacuum
<code>pg_stat_progress_cluster</code>	CLUSTER, VACUUM FULL
<code>pg_stat_progress_create_index</code>	CREATE INDEX, REINDEX
<code>pg_stat_progress_basebackup</code>	Create base backup
<code>pg_stat_progress_copy</code>	COPY

You can monitor the progress of some potentially long-running commands using the corresponding views.

The structures of the views are described in the documentation:

<https://postgrespro.com/docs/postgresql/18/progress-reporting>

Base backup and COPY will be discussed in Backup Overview lesson.

Command Execution

Create a table:

```
=> CREATE TABLE big AS  
    SELECT random() rnd FROM generate_series(1, 10_000_000);
```

```
SELECT 10000000
```

We will monitor the progress in creating an index on this table from another session:

```
=> CREATE INDEX ON big(rnd);
```

```
|| => SELECT * FROM pg_stat_progress_create_index \gx  
||  
|| -[ RECORD 1 ]-----+-----  
|| pid          | 12982  
|| datid        | 16386  
|| datname      | admin_monitoring  
|| relid        | 16390  
|| index_relid   | 0  
|| command      | CREATE INDEX  
|| phase        | building index: scanning table  
|| lockers_total | 0  
|| lockers_done  | 0  
|| current_locker_pid | 0  
|| blocks_total  | 44288  
|| blocks_done   | 626  
|| tuples_total  | 0  
|| tuples_done   | 0  
|| partitions_total | 0  
|| partitions_done | 0
```

```
CREATE INDEX
```

Cumulative Statistics



Views

<code>pg_stat_database</code>	database-wide statistics
<code>pg_stat_all_tables/indexes</code>	accesses to specific table or index
<code>pg_statio_all_tables/indexes</code>	I/O on a specific table or index
<code>pg_stat_io</code>	cluster-wide I/O statistics
<code>pg_stat_wal</code>	generation of WAL records
<code>pg_stat_bgwriter</code>	background writer
<code>pg_stat_checkpoint</code>	checkpoints

Settings

<code>track_counts = on</code>	collect table and index access statistics
<code>track_io_timing = off</code>	monitor block read and write timing statistics
<code>track_wal_io_timing = off</code>	monitor write timing of WAL operations

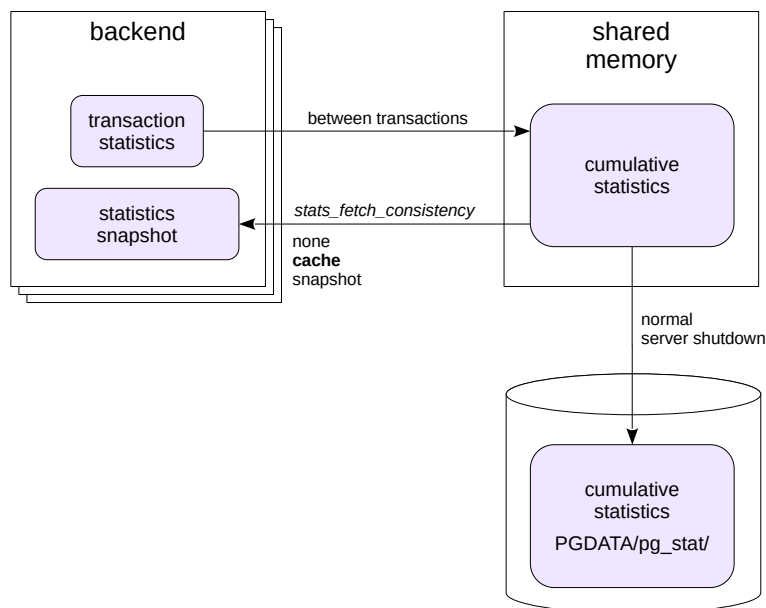
9

The cumulative statistics system in PostgreSQL collects and provides data on server operations. Cumulative statistics track access to tables and indexes at both the disk block and row levels. Additionally, they record details such as the number of rows, vacuum and analyze operations for each table.

PostgreSQL also collects statistics on system operations and background processes.

The amount of information collected is controlled by several server parameters, since the more information is collected, the greater the overhead.

<https://postgrespro.com/docs/postgresql/18/monitoring-stats>



Backends collect statistics in running transactions. The process stores this data in shared memory, updating it at most once per second (compile-time setting).

Cumulative statistics are flushed to PGDATA/pg_stat/ during a normal server shutdown and reloaded upon startup. In case of a crash shutdown, all counters are reset.

Backend may cache statistical data. The caching level is controlled by the *stats_fetch_consistency* parameter:

- None — no caching, statistics reside only in shared memory
- Cache — statistics for a single object are cached
- Snapshot — statistics for the entire database are cached

Cache is the default mode, which balances consistency and performance efficiency.

Cached statistics are not refreshed and are discarded at the end of a transaction, or when `pg_stat_clear_snapshot()` is called.

Due to latency and caching, the backend will not always have the latest statistics, but it is seldom necessary.

Cumulative Statistics

Enable collection of I/O statistics first:

```
=> ALTER SYSTEM SET track_io_timing=on;
```

```
ALTER SYSTEM
```

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

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

Collecting server statistics only makes sense if there is any activity. We can imitate load with pgbench, a standard benchmarking utility.

First, it creates a number of tables and fills them with data.

```
student$ pgbench -i admin_monitoring
```

```
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) of pgbench_accounts done (elapsed 0.15 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.49 s (drop tables 0.00 s, create tables 0.06 s, client-side generate 0.20 s,
vacuum 0.11 s, primary keys 0.12 s).
```

Let's reset the previously accumulated database statistics.

```
=> SELECT pg_stat_reset();
```

```
pg_stat_reset
-----
(1 row)
```

Including instance I/O statistics:

```
=> SELECT pg_stat_reset_shared('io');
```

```
pg_stat_reset_shared
-----
(1 row)
```

Start the TPC-B test and let it run for a few seconds:

```
student$ pgbench -T 10 admin_monitoring
```

```
pgbench (18.6 (Ubuntu 18.6-1.pgdg24.04+2))
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
maximum number of tries: 1
duration: 10 s
number of transactions actually processed: 1159
number of failed transactions: 0 (0.000%)
latency average = 8.625 ms
initial connection time = 8.649 ms
tps = 115.944708 (without initial connection time)
```

Now, let's check the statistics on table access in terms of rows:

```
=> SELECT *
```

```
FROM pg_stat_all_tables
WHERE relid = 'pgbench_accounts'::regclass \gx
```

```
-[ RECORD 1 ]-----+-----
relid          | 16401
schemaname     | public
relname        | pgbench_accounts
seq_scan       | 0
last_seq_scan  |
seq_tup_read   | 0
idx_scan       | 2318
last_idx_scan  | 2026-10-19 23:01:55.845191+03
idx_tup_fetch  | 2318
n_tup_ins      | 0
n_tup_upd      | 1159
n_tup_del      | 0
n_tup_hot_upd  | 326
n_tup_newpage_upd | 833
n_live_tup     | 0
n_dead_tup     | 1087
n_mod_since_analyze | 1159
n_ins_since_vacuum | 0
last_vacuum    |
last_autovacuum |
last_analyze   |
last_autoanalyze |
vacuum_count   | 0
autovacuum_count | 0
analyze_count  | 0
autoanalyze_count | 0
total_vacuum_time | 0
total_autovacuum_time | 0
total_analyze_time | 0
total_autoanalyze_time | 0
```

And in terms of tables:

```
=> SELECT *
FROM pg_statio_all_tables
WHERE relid = 'pgbench_accounts'::regclass \gx
```

```
-[ RECORD 1 ]---+-----
relid          | 16401
schemaname     | public
relname        | pgbench_accounts
heap_blks_read | 0
heap_blks_hit  | 8003
idx_blks_read  | 273
idx_blks_hit   | 6037
toast_blks_read |
toast_blks_hit |
tidx_blks_read |
tidx_blks_hit  |
```

There are similar views for indexes:

```
=> SELECT *
FROM pg_stat_all_indexes
WHERE relid = 'pgbench_accounts'::regclass \gx
```

```
-[ RECORD 1 ]-+-----
relid          | 16401
indexrelid     | 16417
schemaname     | public
relname        | pgbench_accounts
indexrelname   | pgbench_accounts_pkey
idx_scan       | 2318
last_idx_scan  | 2026-10-19 23:01:55.845191+03
idx_tup_read   | 3158
idx_tup_fetch  | 2318
```

```
=> SELECT *
FROM pg_statio_all_indexes
WHERE relid = 'pgbench_accounts'::regclass \gx
```

```

-[ RECORD 1 ]+-----
reloid       | 16401
indexrelid   | 16417
schemaname   | public
relname      | pgbench_accounts
indexrelname | pgbench_accounts_pkey
idx_blks_read | 273
idx_blks_hit  | 6037

```

These views can be used to pinpoint unused indexes. Such indexes not only occupy useful space on the disk, but also waste resources on updates every time data in the table changes.

There are also views for user-defined and system objects (all, user, sys), current transaction statistics (pg_stat_xact*), and more.

You can view global statistics across the whole database:

```

=> SELECT *
FROM pg_stat_database
WHERE datname = 'admin_monitoring' \gx

```

```

-[ RECORD 1 ]+-----
datid          | 16386
datname        | admin_monitoring
numbackends    | 2
xact_commit    | 1177
xact_rollback  | 0
blks_read      | 324
blks_hit       | 24029
tup_returned   | 18522
tup_fetched    | 3385
tup_inserted   | 1159
tup_updated    | 3478
tup_deleted    | 0
conflicts      | 0
temp_files     | 0
temp_bytes     | 0
deadlocks      | 0
checksum_failures | 0
checksum_last_failure | 
blk_read_time  | 36.377
blk_write_time | 1.303
session_time   | 22612.953
active_time    | 33685.113
idle_in_transaction_time | 483.858
sessions       | 2
sessions_abandoned | 0
sessions_fatal | 0
sessions_killed | 0
parallel_workers_to_launch | 0
parallel_workers_launched | 0
stats_reset    | 2026-10-19 23:01:45.500851+03

```

It provides a lot of data on the number of deadlocks occurred, committed and cancelled transactions, utilization of temporary files, and checksum errors. It also maintains total session count and statistics on sessions terminated for various reasons.

The numbackends column specifically indicates the current number of active backend processes connected to this database.

For monitoring I/O operations at the server level, administrators can query the pg_stat_io view. Let's first execute a checkpoint operation, then examine the resulting page read and write counts categorized by process type:

```

=> CHECKPOINT;

```

```

CHECKPOINT

```

```

=> SELECT backend_type, sum(hits) hits, sum(reads) reads, sum(writes) writes
FROM pg_stat_io
GROUP BY backend_type;

```

backend_type	hits	reads	writes
background worker	0	0	0
io worker	0	0	0
walsummarizer		0	0
slotsync worker	0	0	0
client backend	24225	329	1161
walwriter			2
walsender	0	0	0
standalone backend	0	0	0
autovacuum worker	1085	64	0
autovacuum launcher	0	0	0
background writer			1855
startup	0	0	0
walreceiver			0
checkpointer		0	13241
(14 rows)			

Additional Statistics



Supplied extensions

<code>pg_stat_statements</code>	query statistics
<code>pgstattuple</code>	row versions statistics
<code>pg_buffercache</code>	buffer cache status
...	

Third-party extensions

<code>pg_wait_sampling</code>	statistics for waits
<code>pg_stat_kcache</code>	CPU and I/O statistics
...	

There are extensions, both additional supplied and third-party, that enable the collection of additional statistics.

For example, the `pg_stat_statements` extension collects information about queries executed by the system, `pg_buffercache` provides tools for monitoring the buffer cache, etc.

Many key extensions are discussed in more detail in the DBA2 and DEV2 courses.



What to Log?

Where to Log?

Logging Collector

Log Analysis

The other primary source of information about the state of the server is the message log.

What to Log?

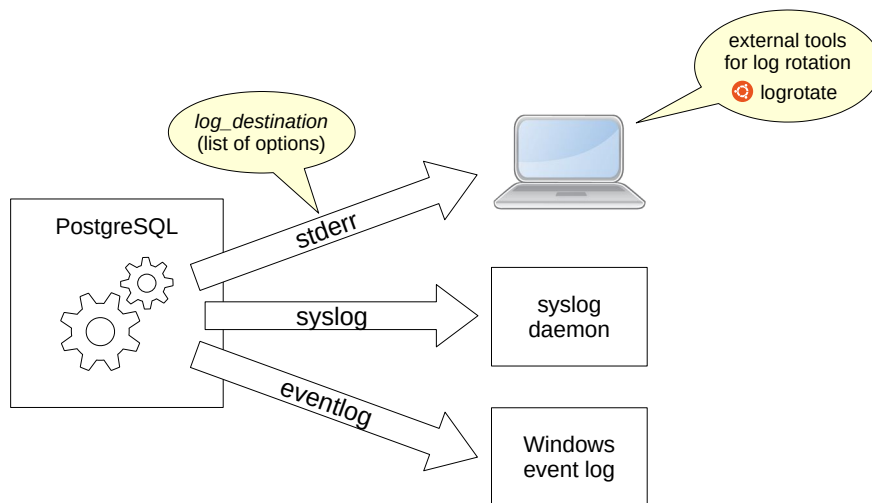


Settings

<i>log_min_messages</i>	level of messages
<i>log_min_duration_statement</i>	long command execution time
<i>log_duration</i>	command execution time
<i>application_name</i>	application name
<i>log_checkpoints</i>	checkpoints
<i>log_(dis)connections</i>	connections and disconnections
<i>log_lock_waits</i>	long lock waits
<i>log_statement</i>	command executed
<i>log_temp_files</i>	temporary file usage
...	

A lot of useful information can be output to the server message log. By default, almost all output is disabled so as not to turn logging into the bottleneck for the I/O subsystem. The administrator should decide what information is important, provide the necessary disk space to store it, and evaluate the impact of the log output on the overall system performance.

Where to Log?



15

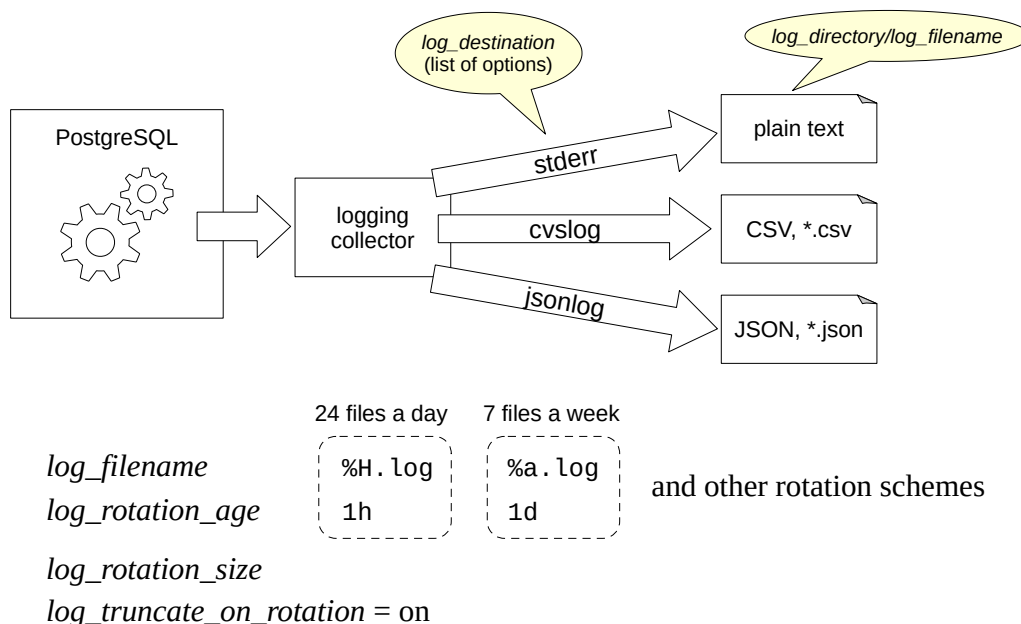
The server log can be forwarded to various destinations, which is determined primarily by the *log_destination* parameter (you can list multiple receivers separated by a comma).

The default *stderr* value sends messages to the standard error stream as plain text, which can be redirected to a file. If all the log output goes into a single file, sooner or later it will grow to an unmanageable size, making administration and analysis highly inconvenient. Therefore, a log rotation scheme is usually employed. For example, Ubuntu package uses *logrotate* system utility (it is configured by the `/etc/logrotate.d/postgresql-common` file).

<https://postgrespro.com/docs/postgresql/18/logfile-maintenance>

The *syslog* value forwards messages to the *syslog* daemon (for Unix systems), and the *eventlog* value does the same for the Windows event log.

Logging Collector



16

The logging collector is an auxiliary process that collects messages sent to the standard error stream and redirects them to log files in various formats. It is controlled by the `logging_collector` parameter and is on by default. The collector is designed to never lose messages (unlike syslog), therefore it can become the bottleneck in high-load environments.

For `stderr`, the messages are written into the file defined by the `log_filename` parameter, which is located in the directory defined by the `log_directory` parameter. When `cvslog` is selected as a receiver, the messages will be output to `log_filename.csv` file in CSV format. With the `jsonlog` output enabled, log files are written in JSON format and use the `.json` extension.

The logging collector has its own rotation capability.

The `log_filename` parameter can specify not just a name, but a file name pattern using designated date and time characters.

The `log_rotation_age` parameter determines how long a file is used before the output switches to a new one (and `log_rotation_size` is the file size at which to switch to the next one).

The `log_truncate_on_rotation` flag determines if PostgreSQL should overwrite existing files or append messages to them.

Different rotation schemes can be defined by using various file name patterns and switch time combinations.

<https://postgrespro.com/docs/postgresql/18/runtime-config-logging#RUNTIME-CONFIG-LOGGING-WHERE>



OS tools

grep, awk...

Special analysis tools

pgBadger — requires a certain log configuration

There are different ways to analyze logs.

You can search for certain information using OS tools or specially designed scripts.

The de facto standard for log analysis is the PgBadger application <https://github.com/darold/pgbadger>, but it imposes certain restrictions on the contents of the log. In particular, only messages in English are allowed.

Log Analysis

Let's consider a simple case. For example, display all messages of the FATAL level:

```
student$ sudo grep FATAL /var/log/postgresql/postgresql-18-main.log | tail -n 10

2026-10-19 23:00:18.517 MSK [11211] student@data_logical FATAL: terminating connection
due to administrator command
2026-10-19 23:00:59.896 MSK [13094] student@admin_monitoring FATAL: terminating
connection due to administrator command
```

The "terminating connection" message is caused by us terminating the blocking process.

Logs are usually used to analyse the queries that execute the longest. We can make the log display all executed commands and their execution times:

```
=> ALTER SYSTEM SET log_min_duration_statement=0;
```

```
ALTER SYSTEM
```

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

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

Now, run a command:

```
=> SELECT sum(random()) FROM generate_series(1,1_000_000);
```

```
sum
-----
499861.5310412343
(1 row)
```

Check the log:

```
student$ sudo tail -n 1 /var/log/postgresql/postgresql-18-main.log
```

```
2026-10-19 23:02:00.694 MSK [12982] student@admin_monitoring LOG: duration: 322.590 ms
statement: SELECT sum(random()) FROM generate_series(1,1_000_000);
```



Universal monitoring systems

Zabbix, Munin, Cacti...
cloud-based: Okmeter, NewRelic, Datadog...

PostgreSQL monitoring systems

pg_profile, pgpro_pwr
PGObserver
PostgreSQL Workload Analyzer (PoWA)
Open PostgreSQL Monitoring (OPM)
...

In practice, you need a full-fledged monitoring system that collects various metrics from both PostgreSQL and the operating system, stores the history of these metrics, displays them as readable graphs, notifies when certain metrics exceed certain limits, etc.

PostgreSQL does not come with such a system by itself, it only provides the means by which such information can be acquired. We have gone over them already. Therefore, for full-scale monitoring, an external system is required. There are quite a few such systems on the market. Some are universal and come with PostgreSQL plugins or agents. These include Zabbix, Munin, Cacti, cloud services such as Okmeter, NewRelic, Datadog, and others.

There are also systems specifically designed for PostgreSQL: PGObserver, PoWA, OPM, etc. The pg_profile extension allows you to build snapshots of static data and compare them, identifying resource-intensive operations and their dynamics. pgpro_pwr is its extended, commercially available version.

<https://postgrespro.com/docs/enterprise/18/pgpro-pwr>

An incomplete but representative list of monitoring systems can be viewed here: <https://wiki.postgresql.org/wiki/Monitoring>

Takeaways



Monitoring collects data on server operations both from the operating system and from the database server points of view
PostgreSQL provides statistics views and message log
Full-scale monitoring in production environment requires an external system

Practice



1. In a new database, create a table with autovacuum turned off, insert several rows, and then delete all rows.
Look at the table access statistics and reference the values (n_tup_ins, n_tup_del, n_live_tup, n_dead_tup) against your activity.
Perform a vacuum, check the statistics again and compare with the previous figures.
2. Create a deadlock with two transactions.
See what information is recorded in the server message log.

21

1. Use WITH (autovacuum_enabled = off) clause.
2. Deadlock is a situation when two (or more) transactions are waiting for each other to complete first. Unlike a normal lock, transactions have no way to get out of deadlock, and the DBMS has to resolve it by forcibly interrupting one of the transactions.

The easiest way to reproduce a deadlock is on a table with two rows. The first transaction changes (and locks) the first row, and the second one locks the second row. Then the first transaction tries to change the second row, discovers that it is locked, and starts waiting. And then the second transaction tries to change the first row, and also waits for the lock to be released.

1. Table Access Statistics

Create a database and a table:

```
=> CREATE DATABASE admin_monitoring;
```

CREATE DATABASE

```
=> \c admin_monitoring
```

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

```
=> CREATE TABLE t(n numeric) WITH (autovacuum_enabled = off);
```

CREATE TABLE

```
=> INSERT INTO t SELECT 1 FROM generate_series(1,1000);
```

INSERT 0 1000

```
=> DELETE FROM t;
```

DELETE 1000

Check access statistics.

```
=> SELECT * FROM pg_stat_all_tables WHERE relid = 't'::regclass \gx
```

```
-[ RECORD 1 ]-----+-----
relid          | 16387
schemaname     | public
relname        | t
seq_scan       | 1
last_seq_scan  | 2026-10-19 23:07:41.772295+03
seq_tup_read   | 1000
idx_scan       | 
last_idx_scan  | 
idx_tup_fetch  | 
n_tup_ins      | 1000
n_tup_upd      | 0
n_tup_del      | 1000
n_tup_hot_upd  | 0
n_tup_newpage_upd | 0
n_live_tup     | 0
n_dead_tup     | 1000
n_mod_since_analyze | 2000
n_ins_since_vacuum | 1000
last_vacuum    | 
last_autovacuum | 
last_analyze   | 
last_autoanalyze | 
vacuum_count   | 0
autovacuum_count | 0
analyze_count  | 0
autoanalyze_count | 0
total_vacuum_time | 0
total_autovacuum_time | 0
total_analyze_time | 0
total_autoanalyze_time | 0
```

We inserted 1000 rows (n_tup_ins = 1000), then removed 1000 rows (n_tup_del = 1000).

No live row versions remain (n_live_tup = 0), all 1000 rows are dead (n_dead_tup = 1000).

Run vacuuming.

```
=> VACUUM;
```

VACUUM

```
=> SELECT * FROM pg_stat_all_tables WHERE relid = 't'::regclass \gx
```

```

-[ RECORD 1 ]-----+-----
reloid          | 16387
schemaname      | public
relname         | t
seq_scan        | 1
last_seq_scan   | 2026-10-19 23:07:41.772295+03
seq_tup_read    | 1000
idx_scan        |
last_idx_scan   |
idx_tup_fetch   |
n_tup_ins       | 1000
n_tup_upd       | 0
n_tup_del       | 1000
n_tup_hot_upd   | 0
n_tup_newpage_upd | 0
n_live_tup      | 0
n_dead_tup      | 0
n_mod_since_analyze | 2000
n_ins_since_vacuum | 0
last_vacuum     | 2026-10-19 23:07:43.458769+03
last_autovacuum |
last_analyze    |
last_autoanalyze |
vacuum_count    | 1
autovacuum_count | 0
analyze_count   | 0
autoanalyze_count | 0
total_vacuum_time | 19
total_autovacuum_time | 0
total_analyze_time | 0
total_autoanalyze_time | 0

```

Dead row versions vacuumed (n_dead_tup = 0), vacuuming performed one time (vacuum_count = 1).

2. Deadlocks

=> **INSERT INTO t VALUES (1),(2);**

INSERT 0 2

One transaction locks the first row of the table...

student\$ **psql**

```

| => \c admin_monitoring
|
| You are now connected to database "admin_monitoring" as user "student".
|
| => BEGIN;
|
| BEGIN
|
| => UPDATE t SET n = 10 WHERE n = 1;
|
| UPDATE 1

```

The other locks the second row...

student\$ **psql**

```

|| => \c admin_monitoring
||
|| You are now connected to database "admin_monitoring" as user "student".
||
|| => BEGIN;
||
|| BEGIN
||
|| => UPDATE t SET n = 200 WHERE n = 2;
||
|| UPDATE 1

```

Now, the first transaction tries to change the second row and waits for it to release...

```

| => UPDATE t SET n = 20 WHERE n = 2;

```

While the second transaction waits for the first row to release...

```

|| => UPDATE t SET n = 100 WHERE n = 1;

```

...and so a deadlock occurs. The server terminates one of the transactions:

```
ERROR: deadlock detected
```

```
DETAIL: Process 26889 waits for ShareLock on transaction 761; blocked by process 26745.
```

```
Process 26745 waits for ShareLock on transaction 762; blocked by process 26889.
```

```
HINT: See server log for query details.
```

```
CONTEXT: while updating tuple (0,1) in relation "t"
```

The other transaction gets unblocked:

```
UPDATE 1
```

Check the message log:

```
student$ sudo tail -n 8 /var/log/postgresql/postgresql-18-main.log
```

```
2026-10-19 23:07:47.875 MSK [26889] student@admin_monitoring ERROR: deadlock detected
```

```
2026-10-19 23:07:47.875 MSK [26889] student@admin_monitoring DETAIL: Process 26889 waits  
for ShareLock on transaction 761; blocked by process 26745.
```

```
Process 26745 waits for ShareLock on transaction 762; blocked by process 26889.
```

```
Process 26889: UPDATE t SET n = 100 WHERE n = 1;
```

```
Process 26745: UPDATE t SET n = 20 WHERE n = 2;
```

```
2026-10-19 23:07:47.875 MSK [26889] student@admin_monitoring HINT: See server log for  
query details.
```

```
2026-10-19 23:07:47.875 MSK [26889] student@admin_monitoring CONTEXT: while updating  
tuple (0,1) in relation "t"
```

```
2026-10-19 23:07:47.875 MSK [26889] student@admin_monitoring STATEMENT: UPDATE t SET n =  
100 WHERE n = 1;
```