

Administrative tasks Monitoring



Copyright

© Postgres Professional, 2015–2022

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

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.

OS tools

Database statistics

Server message log

External monitoring systems

Processes

`ps (grep postgres)`

`update_process_title` parameter for updating the status of processes

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.

Unix provides multiple state and performance monitoring tools.

In particular, you can monitor the processes belonging to PostgreSQL.

The server parameter *update_process_title* (on by default) displays the state of each process next to its title, making it even more convenient.

Various tools are available to monitor the use of system resources (CPU, RAM, disks): `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/13/monitoring-ps>

<https://postgrespro.com/docs/postgresql/13/diskusage>

Stats collector

Ongoing system activities

Command execution monitoring

Extensions

There are two primary sources of information about the state of the system. The first one is statistical information collected by PostgreSQL and stored inside the database.

Stats collector process settings

statistics

table and index access
(access, touched rows)

page accesses

user function calls

parameter

track_counts

on by default
needed for vacuuming

track_io_timing

off by default

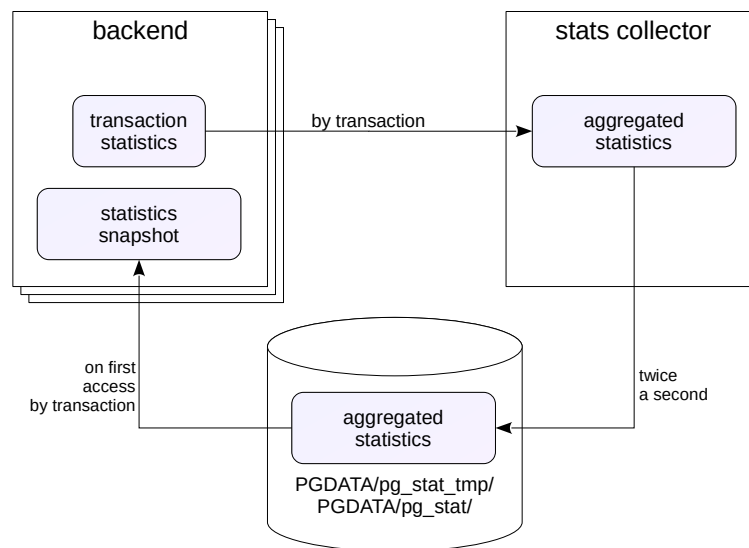
track_functions

off by default

In addition to tracking ongoing activities, PostgreSQL also collects some statistics.

These statistics are collected by the stats collector background process. 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/13/monitoring-stats>



Backends collect statistics from executed transactions. The stats collector process collects statistics from all backends and aggregates it. Once every half a second (can be changed during compilation), the collector dumps statistics to temporary files in the `PGDATA/pg_stat_tmp` directory. (Therefore, moving this directory to an in-memory file system can improve overall performance.)

When a worker process requests statistics data (via views or functions), it's served a *statistics snapshot*, the most recent version of statistics provided by the collector. Unless explicitly requested, the process will not read new snapshots until the end of the transaction to ensure consistency.

Due to latency, the worker process will not always have the latest statistics, but it is seldom necessary.

On server shutdown, the collector dumps statistics data into permanent files inside the `PGDATA/pg_stat` catalog. When the server starts up again, it can keep using the data. Statistics can be reset manually by the administrator, and always reset after a crash.

Database statistics

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

```
CREATE DATABASE
```

```
=> \c admin_monitoring
```

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

Enable collection of input-output statistics first:

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

```
ALTER SYSTEM
```

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

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

Monitoring server activity only makes sense if there is any activity to be monitored in the first place. We can imitate load with pgbench, a stock 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%) done (elapsed 0.11 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.30 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 0.14 s, vacuum 0.07 s, primary keys 0.08 s).
```

Reset any previously collected statistics:

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

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

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

```
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
```

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 10 s
number of transactions actually processed: 5717
latency average = 1.749 ms
tps = 571.687219 (including connections establishing)
tps = 571.841303 (excluding connections establishing)
```

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

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

```

-[ RECORD 1 ]-----+-----
reloid        | 16546
schemaname    | public
relname       | pgbench_accounts
seq_scan      | 0
seq_tup_read  | 0
idx_scan      | 11434
idx_tup_fetch | 11434
n_tup_ins     | 0
n_tup_upd     | 5717
n_tup_del     | 0
n_tup_hot_upd | 4119
n_live_tup    | 0
n_dead_tup    | 2989
n_mod_since_analyze | 5717
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

```

And in terms of tables:

```

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

```

```

-[ RECORD 1 ]-----+-----
reloid        | 16546
schemaname    | public
relname       | pgbench_accounts
heap_blks_read | 26
heap_blks_hit  | 23695
idx_blks_read  | 276
idx_blks_hit   | 25879
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 ]-----+-----
reloid        | 16546
indexrelid    | 16560
schemaname    | public
relname       | pgbench_accounts
indexrelname   | pgbench_accounts_pkey
idx_scan      | 11434
idx_tup_read  | 13122
idx_tup_fetch | 11434

```

```

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

```

```

-[ RECORD 1 ]-----+-----
reloid        | 16546
indexrelid    | 16560
schemaname    | public
relname       | pgbench_accounts
indexrelname   | pgbench_accounts_pkey
idx_blks_read | 276
idx_blks_hit  | 25879

```

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	16539
datname	admin_monitoring
numbackends	1
xact_commit	5732
xact_rollback	0
blks_read	395
blks_hit	79186
tup_returned	78116
tup_fetched	11956
tup_inserted	5717
tup_updated	17152
tup_deleted	0
conflicts	0
temp_files	0
temp_bytes	0
deadlocks	0
checksum_failures	
checksum_last_failure	
blk_read_time	7.186
blk_write_time	0
stats_reset	2024-03-07 13:50:39.859183+03

It provides a lot of data on the number of deadlocks occurred, committed and cancelled transactions, utilization of temporary files, and checksum errors.

PostgreSQL 14 also added statistics on user sessions.

There are separate statistics for background writer and checkpoint, valuable as they are for monitoring:

```
=> CHECKPOINT;
```

```
CHECKPOINT
```

```
=> SELECT * FROM pg_stat_bgwriter \gx
```

-[RECORD 1]-----+	
checkpoints_timed	0
checkpoints_req	1
checkpoint_write_time	44
checkpoint_sync_time	35
buffers_checkpoint	2043
buffers_clean	0
maxwritten_clean	0
buffers_backend	1742
buffers_backend_fsync	0
buffers_alloc	423
stats_reset	2024-03-07 13:50:39.907856+03

- buffers_clean — number of pages written by background writer;
- buffers_checkpoint — number of pages written with checkpoints;
- buffers_backend — number of pages written by backends.

Configuration

statistics

current activities
and backends' and background
processes' waits

parameter

track_activities
on by default

The current activities of all backends and background processes are displayed in the `pg_stat_activity` view. We will focus on it more in the demo. This view depends on the *track_activities* parameter (enabled by default).

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:

```
=> 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 nothing more:

```
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          | 136196
query        | UPDATE t SET n = n + 1;
state        | idle in transaction
wait_event   | ClientRead
wait_event_type | Client
pg_blocking_pids | {}
-[ RECORD 2 ]-----+-----
pid          | 135423
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 3 ]-----+-----
pid          | 136276
query        | UPDATE t SET n = n + 2;
state        | active
wait_event   | transactionid
wait_event_type | Lock
pg_blocking_pids | {136196}
```

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 session 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 they are idle for a certain period of time.

You can also terminate a 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
-----
      136276
(1 row)

```

You don't need `pg_blocking_pids` to find the blocking process. Instead, you can access the locks table directly. It will return two rows in this case: one for the transaction that has been granted the lock and another for the one that hasn't.

```

=> SELECT locktype, transactionid, pid, mode, granted
FROM pg_locks
WHERE transactionid IN (
  SELECT transactionid FROM pg_locks WHERE pid = 136276 AND NOT granted
);

```

locktype	transactionid	pid	mode	granted
transactionid	6301	136196	ExclusiveLock	t
transactionid	6301	136276	ShareLock	f

(2 rows)

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

A process can be cancelled with the `pg_cancel_backend` function. The transaction is idle in our case, so we can use the `pg_terminate_backend` command to terminate it:

```

=> SELECT pg_terminate_backend(b.pid)
FROM unnest(pg_blocking_pids(136276)) 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

```

pid	query	state	wait_event	wait_event_type
135423	SELECT pid, query, state, wait_event, wait_event_type+ FROM pg_stat_activity WHERE backend_type = 'client backend'	active		
136276	UPDATE t SET n = n + 2;	idle	ClientRead	Client

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

The `pg_stat_activity` view shows not only the information about backend processes, but also about the service processes running on the instance:

```

=> SELECT pid, backend_type, backend_start, state
FROM pg_stat_activity;

```

pid	backend_type	backend_start	state
129564	autovacuum launcher	2024-03-07 13:49:22.166377+03	
129566	logical replication launcher	2024-03-07 13:49:22.198149+03	
135423	client backend	2024-03-07 13:50:39.282317+03	active
136276	client backend	2024-03-07 13:50:52.1675+03	idle
129562	background writer	2024-03-07 13:49:22.204068+03	
129561	checkpointer	2024-03-07 13:49:22.206842+03	
129563	walwriter	2024-03-07 13:49:22.20064+03	

(7 rows)

Compare that to what the OS sees:

```
student$ sudo head -n 1 /var/lib/postgresql/13/main/postmaster.pid
```

129559

```
student$ sudo ps -o pid,command --ppid 129559
```

```

PID COMMAND
129561 postgres: 13/main: checkpointer
129562 postgres: 13/main: background writer
129563 postgres: 13/main: walwriter
129564 postgres: 13/main: autovacuum launcher
129565 postgres: 13/main: stats collector
129566 postgres: 13/main: logical replication launcher
135423 postgres: 13/main: student admin_monitoring [local] idle
136276 postgres: 13/main: student admin_monitoring [local] idle
```

pg_stat_activity does not include the stats collector process.

Views for monitoring command executions

<i>command</i>	<i>execution</i>
ANALYZE	pg_stat_progress_analyze
CREATE INDEX, REINDEX	pg_stat_progress_create_index
VACUUM including autovacuuming	pg_stat_progress_vacuum
CLUSTER, VACUUM FULL	pg_stat_progress_cluster
Create base backup	pg_stat_progress_basebackup

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/13/progress-reporting>

Backup is discussed in the Backup module.

In PostgreSQL 14, the `pg_stat_progress_copy` view was added to this list to track the COPY command.

Stock extensions

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

Other extensions

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

There are extensions, both stock 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.

Server message log

Log record configuration

Log file rotation

Log analysis

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

Server message log

Message receiver (*log_destination* = list)

stderr	error stream
csvlog	CSV format (if the collector is enabled)
syslog	the syslog daemon
eventlog	Windows event log

Message collector (*logging_collector* = on)

- can provide additional info
- never loses messages (unlike syslog)
- writes stderr and csvlog to the *log_directory/log_filename* file

The server log can be output in various formats and forwarded to various destinations. The format and the destination are determined primarily by the *log_destination* parameter (you can list multiple destinations separated by a comma).

The stderr flag (on by default) streams message log errors into the standard error log as plain text. The syslog flag tells the log to forward messages to the syslog daemon (for Unix systems), and the eventlog flag does the same for the Windows event log.

The message collector is an auxiliary process that collects additional information from all PostgreSQL processes to supplement the basic log messages. It is designed to keep track of every message, therefore it can become the bottleneck in high-load environments.

The message collector is switched on and off by the *logging_collector* flag. When stderr is on, the log writes into the file defined by the *log_filename* parameter, which is located in the directory defined by the *log_directory* parameter.

When the collector is on and csvlog is selected as a destination, the log will also write output into a CSV file *log_filename.csv*. And in PostgreSQL 15, jsonlog becomes a destination option.

What to log?

Settings

information

level of messages
long command execution time
command execution time
application name
checkpoints
connections and disconnections
long lock waits
command execution outputs
temporary files usage
etc.

parameter

log_min_messages
log_min_duration_statement
log_duration
application_name
log_checkpoints
log_(dis)connections
log_lock_waits
log_statement
log_temp_files

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 disk subsystem. The administrator must decide what information is important, provide the necessary disk space to store it, and evaluate the impact on the overall system performance.

Log file rotation


By the message collector

<i>statistics</i>	<i>parameter</i>
file name mask	<i>log_filename</i>
rotation time, minutes	<i>log_rotation_age</i>
rotation file size, KB	<i>log_rotation_size</i>
allow to rewrite files	<i>log_truncate_on_rotation</i> = on

different file name masks and rotation times allow for different combinations:

'postgresql-%H.log', '1h'	24 files a day
'postgresql-%a.log', '1d'	7 files a week

External tools

-  logrotate system utility

15

If all the log output goes into a single file, sooner or later the file will grow to an unmanageable size, making administration and analysis highly inconvenient. Therefore, a log rotation scheme is usually employed.

<https://postgrespro.com/docs/postgresql/13/logfile-maintenance>

The message collector has its own rotation tools. Some of the parameters that configure them are listed on the slide.

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

The *log_rotation_age* parameter determines how long a file is used before the log 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 the log should overwrite existing files or not.

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

<https://postgrespro.com/docs/postgresql/13/runtime-config-logging.html#RUNTIME-CONFIG-LOGGING-WHERE>

Alternatively, external rotation management tools can be used, such as logrotate from the Ubuntu package (it's configured through the `/etc/logrotate.d/postgresql-common` file).

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/dalibo/pgbadger>), but it imposes certain restrictions on the contents of the log.

In particular, only messages in English are allowed.

Log analysis

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

```
student$ sudo grep FATAL /var/log/postgresql/postgresql-13-main.log | tail -n 10
```

```
2024-03-07 13:48:13.303 MSK [122044] student@student FATAL: terminating connection due to administrator command
2024-03-07 13:49:12.002 MSK [128903] student@student FATAL: terminating connection due to administrator command
2024-03-07 13:50:53.569 MSK [136196] 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(1,1000000);
```

```
sum
-----
500408.0170670183
(1 row)
```

Check the log:

```
student$ sudo tail -n 1 /var/log/postgresql/postgresql-13-main.log
```

```
2024-03-07 13:50:54.323 MSK [135423] student@admin_monitoring LOG: duration: 191.569 ms statement: SELECT sum(random()) FROM generate_series(1,1000000);
```

Universal monitoring systems

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

PostgreSQL monitoring systems

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

In practice, for any serious environment, 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 reach certain thresholds, etc.

PostgreSQL does not come with such a system by itself, it only provides the means by which such information can be acquired. We've 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 settings. 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.

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

Monitoring collects data on server operations
both from the operating system
and from the database points of view

PostgreSQL provides collected statistics
and the server message log

Full-scale monitoring requires an external system

1. In a new database, create a table, 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.

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 is forced 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's locked, and starts waiting. And then the second transaction tries to change the first row, and also waits for the lock to be released.

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);
```

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          | 16719
schemaname     | public
relname        | t
seq_scan       | 1
seq_tup_read   | 1000
idx_scan       |
idx_tup_fetch  |
n_tup_ins      | 1000
n_tup_upd      | 0
n_tup_del      | 1000
n_tup_hot_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
```

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 ]-----+-----
relid          | 16719
schemaname     | public
relname        | t
seq_scan       | 1
seq_tup_read   | 1000
idx_scan       |
idx_tup_fetch  |
n_tup_ins      | 1000
n_tup_upd      | 0
n_tup_del      | 1000
n_tup_hot_upd  | 0
n_live_tup     | 0
n_dead_tup     | 0
n_mod_since_analyze | 2000
n_ins_since_vacuum | 0
last_vacuum    | 2024-03-07 13:54:28.194582+03
last_autovacuum |
last_analyze   |
last_autoanalyze |
vacuum_count   | 1
autovacuum_count | 0
analyze_count  | 0
autoanalyze_count | 0
```

Dead row versions vacuumed (n_dead_tup = 0), vacuuming performed in one pass (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.

```
||      UPDATE 1
|
| ERROR:  deadlock detected
| DETAIL:  Process 154989 waits for ShareLock on transaction 6479; blocked by process 155108.
|          Process 155108 waits for ShareLock on transaction 6478; blocked by process 154989.
|          HINT:  See server log for query details.
|          CONTEXT:  while updating tuple (0,2) in relation "t"
```

Check the message log:

```
student$ sudo tail -n 8 /var/log/postgresql/postgresql-13-main.log
```

```
2024-03-07 13:54:31.068 MSK [154989] student@admin_monitoring ERROR:  deadlock detected
2024-03-07 13:54:31.068 MSK [154989] student@admin_monitoring DETAIL:  Process 154989 waits for ShareLock on transaction 6479; blocked by process 155108.
          Process 155108 waits for ShareLock on transaction 6478; blocked by process 154989.
          Process 154989: UPDATE t SET n = 20 WHERE n = 2;
          Process 155108: UPDATE t SET n = 100 WHERE n = 1;
2024-03-07 13:54:31.068 MSK [154989] student@admin_monitoring HINT:  See server log for query details.
2024-03-07 13:54:31.068 MSK [154989] student@admin_monitoring CONTEXT:  while updating tuple (0,2) in relation "t"
2024-03-07 13:54:31.068 MSK [154989] student@admin_monitoring STATEMENT:  UPDATE t SET n = 20 WHERE n = 2;
```

1. Install the `pg_stat_statements` extension.
Execute several queries.
See what information gets into the `pg_stat_statements` view.

1. To install the extension, in addition to executing the `CREATE EXTENSION` command, you need to change the value of the `shared_preload_libraries` parameter and restart the server.

<https://postgrespro.com/docs/postgresql/13/pgstatstatements>

1. The pg_stat_statements extension

The extension collects planning and execution statistics for all queries.

For the extension to work, a module with the same name has to be loaded. To do that, add the module name to `shared_preload_libraries` and restart the server. This is usually done through `postgresql.conf`, but for the purpose of the demo we will set it using the `ALTER SYSTEM` command.

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';
```

ALTER SYSTEM

```
=> \q
```

```
student$ sudo pg_ctlcluster 13 main restart
```

```
student$ psql
```

Verify the parameter value and enable the extension. Since it will track queries for all databases within the cluster, it should be installed into a database that is always present, such as the `postgres` database.

```
=> \c postgres
```

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

```
=> SHOW shared_preload_libraries;
```

```
shared_preload_libraries
-----
pg_stat_statements
(1 row)
```

```
=> CREATE EXTENSION pg_stat_statements;
```

CREATE EXTENSION

Now, run some queries:

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

CREATE TABLE

```
=> SELECT format('INSERT INTO t VALUES (%L)', x)
FROM generate_series(1,5) AS x \gexec
```

```
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
INSERT 0 1
```

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

DELETE 5

```
=> DROP TABLE t;
```

DROP TABLE

Check the statistics for the most frequently executed query.

```
=> SELECT * FROM pg_stat_statements ORDER BY calls DESC LIMIT 1 \gx
```

```

-[ RECORD 1 ]-----+-----
userid       | 16384
dbid         | 13485
queryid      | 9098096990651905112
query        | INSERT INTO t VALUES ($1)
plans        | 0
total_plan_time | 0
min_plan_time | 0
max_plan_time | 0
mean_plan_time | 0
stddev_plan_time | 0
calls        | 5
total_exec_time | 0.185624
min_exec_time | 0.012253
max_exec_time | 0.12164799999999999
mean_exec_time | 0.0371248
stddev_exec_time | 0.04248937570452171
rows         | 5
shared_blks_hit | 4
shared_blks_read | 0
shared_blks_dirtied | 1
shared_blks_written | 1
local_blks_hit | 0
local_blks_read | 0
local_blks_dirtied | 0
local_blks_written | 0
temp_blks_read | 0
temp_blks_written | 0
blk_read_time | 0
blk_write_time | 0
wal_records   | 5
wal_fpi       | 0
wal_bytes     | 300

```