**Copyright**
© Postgres Professional, 2017, 2018, 2019.
Authors: Egor Rogov, Pavel Luzanov

**Feedback**
Send feedback, comments and suggestions to:
edu@postgrespro.ru

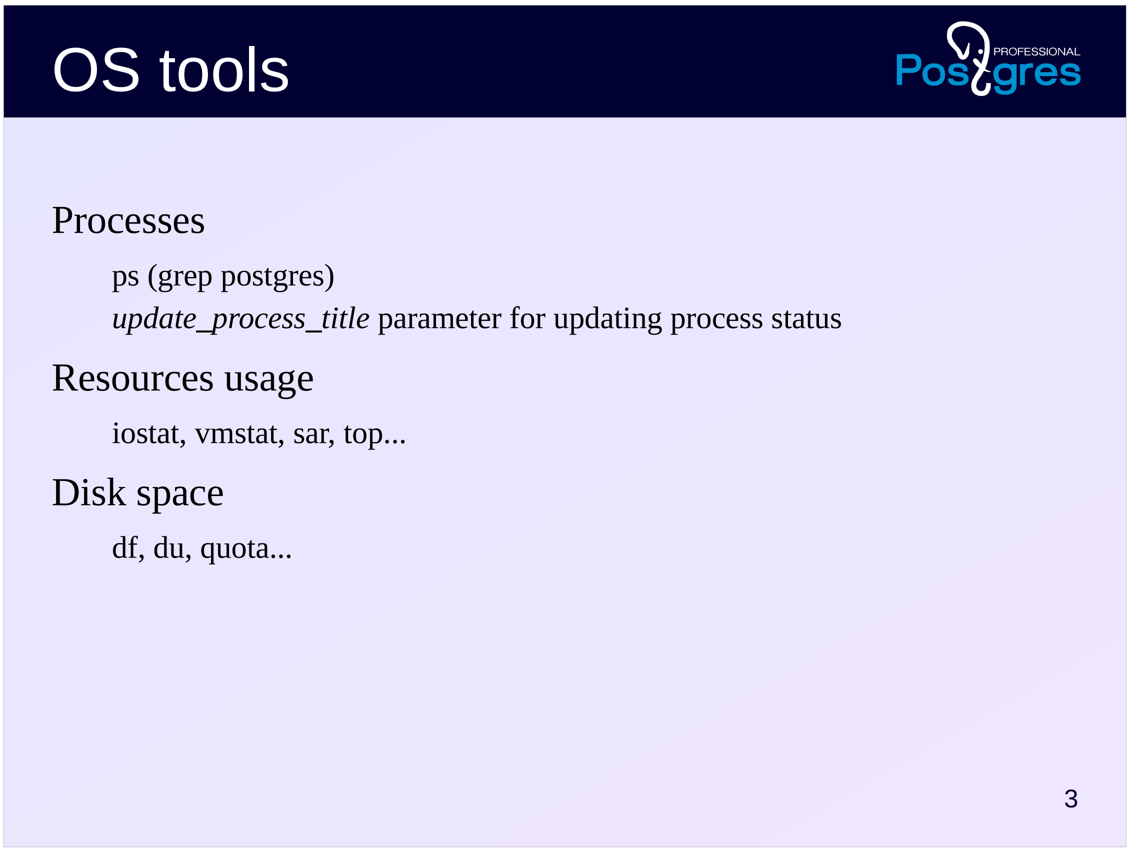
## Topics

Operating system tools

Statistics tracked by the database

Server messages log

External monitoring systems



## OS tools

Processes

- ps (grep postgres)
- *update_process_title* parameter for updating process status

Resources usage

- iostat, vmstat, sar, top...

Disk space

- df, du, quota...

PostgreSQL is running on top of the operating system and to a certain extent depends on its settings.

Unix provides many tools for analyzing of overall health and performance.

In particular, you can look at the processes belonging to PostgreSQL. This is especially useful when the *update_process_title* server parameter is enabled (by default) to show the current state in the process name.

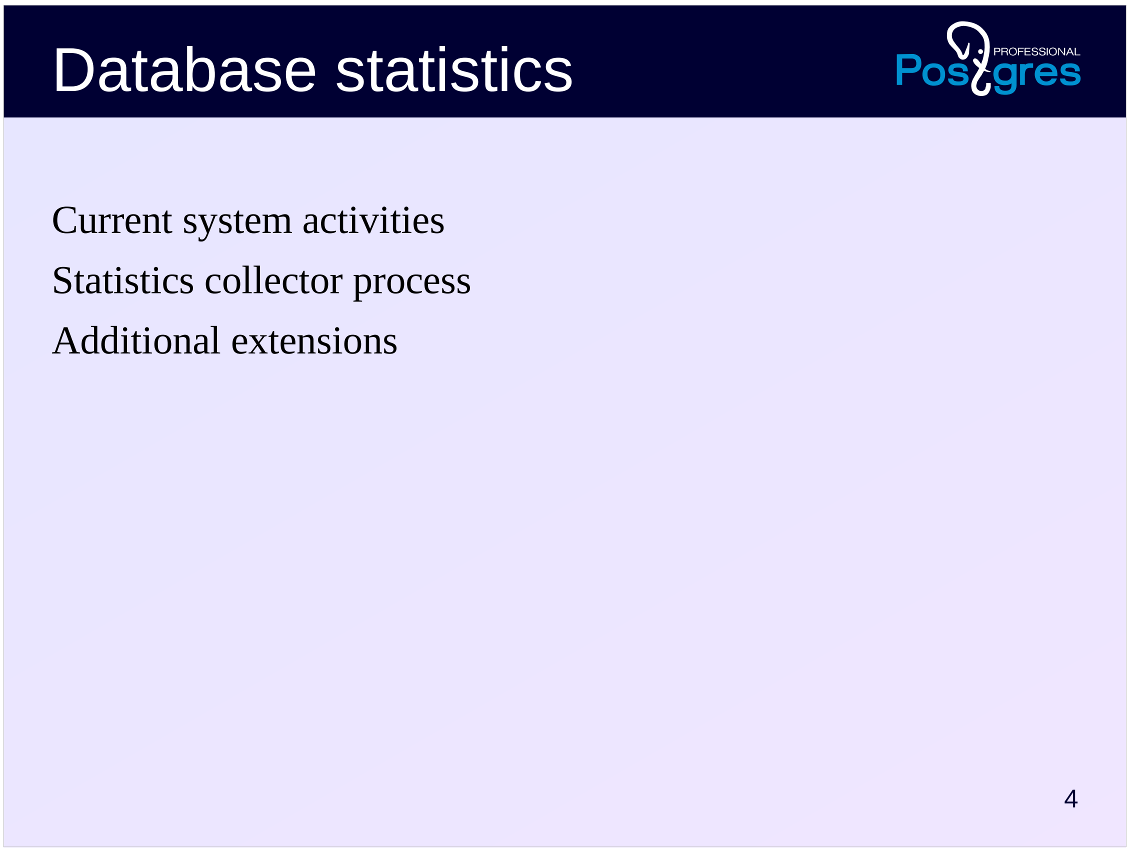
To study the usage of system resources (processor, memory, disks) there are various tools: `iostat`, `vmstat`, `sar`, `top`, etc.

It is necessary to monitor the size of available disk space. Space occupied by the database can be viewed both from the database itself (see the «Data Organization» module), and from the OS (`du` command). The size of the available disk space must be viewed in the OS (`df` command). If disk quotas are used, they should be taken into account.

In general, the set of tools and approaches can vary widely depending on the operating system and file system used, so they are not discussed in detail here.

https://postgrespro.com/docs/postgresql/11/monitoring-ps

https://postgrespro.com/docs/postgresql/11/diskusage



## Database statistics

Current system activities

Statistics collector process

Additional extensions

There are two main sources of information about what is happening in the system. The first one is statistical information that is collected by PostgreSQL and stored inside the database.

## Current activities

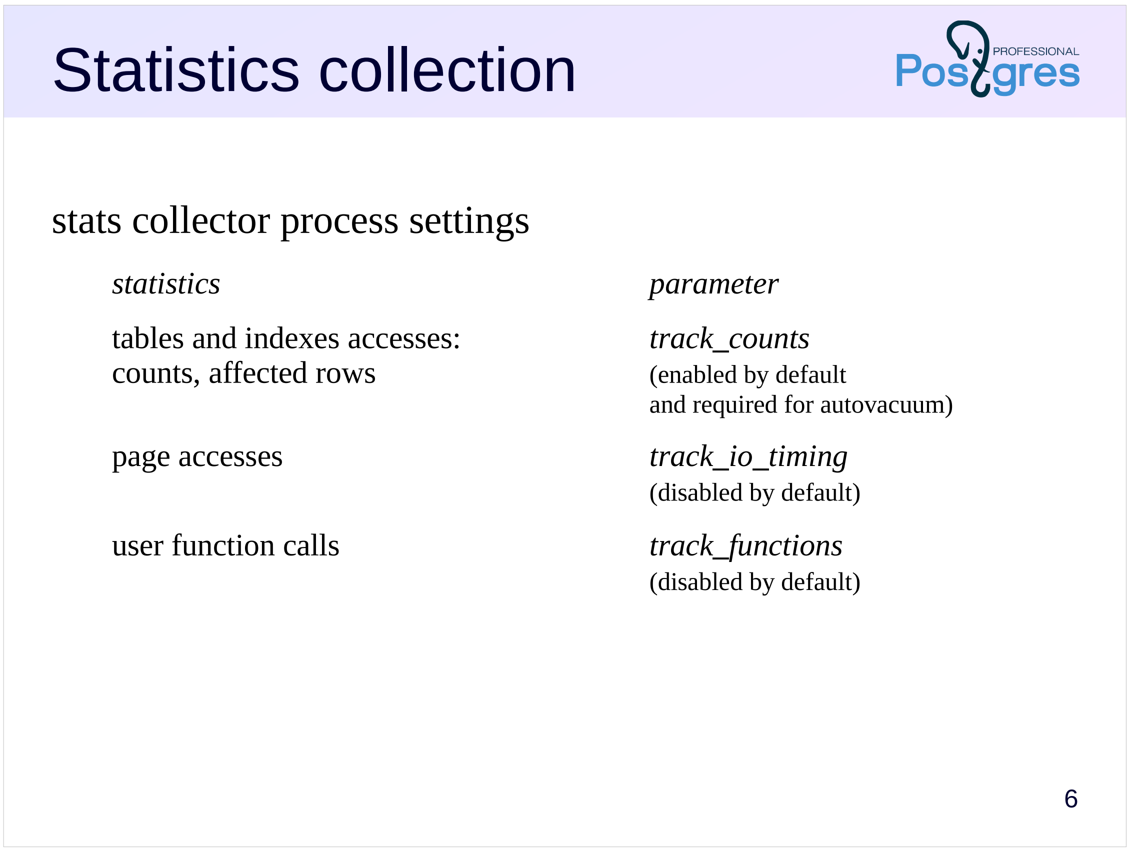
Settings

| *statistics* | *parameter* |
|---|---|
| current activities and wait events for backends and background processes | *track_activities* (enabled by default) |

The current activity of all backend processes and (starting from version 10) background processes is tracked and displayed in the `pg_stat_activity` view. More on it we will discuss in the demonstration.

In addition, there are several other views showing the current server activity (vacuuming, replication, etc).

The possibility to track current activities can be disabled by the parameter *track_activities*, but this should not be done.



## Statistics collection

stats collector process settings

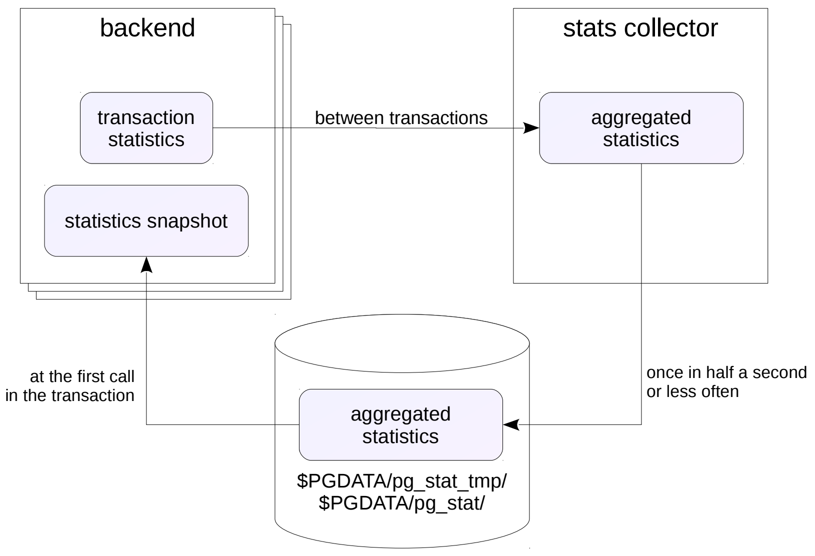
| *statistics* | *parameter* |
|---|---|
| tables and indexes accesses: counts, affected rows | *track_counts* (enabled by default and required for autovacuum) |
| page accesses | *track_io_timing* (disabled by default) |
| user function calls | *track_functions* (disabled by default) |

In addition to showing the current activities, PostgreSQL collects some statistics.

Statistics is 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/11/monitoring-stats

## Architecture

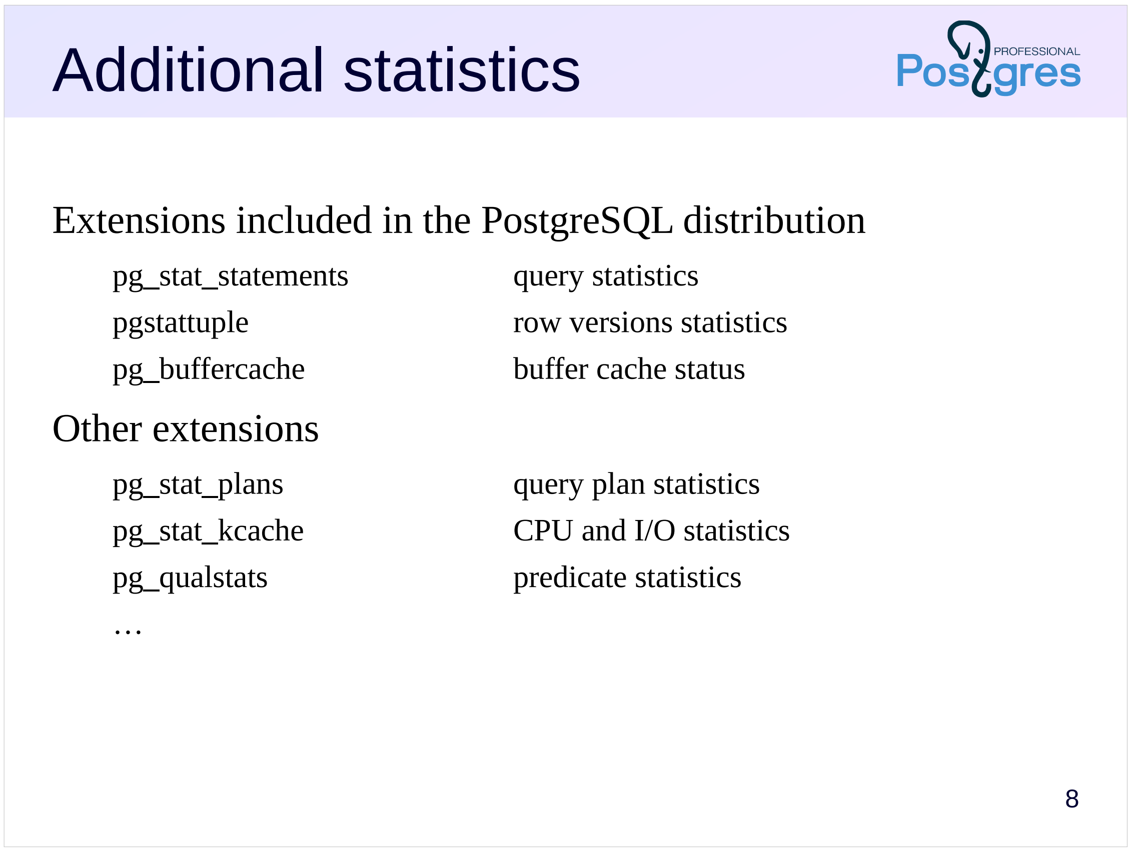


Each backend process collects the necessary statistics as part of each transaction performed. This statistic is then passed to the collector process. The collector receives and aggregates statistics from all backends. Not more often than once every half second (time is configured when compiling PostgreSQL), the collector dumps statistics into temporary files in the `$PGDATA/pg_stat_tmp` directory (therefore, transferring this directory to the in-memory file system usually have a positive effect on performance).

When a backend requests information on statistics (through views or functions), the latest available version of statistics is read into its memory — this is called a statistics snapshot. Unless explicitly requested, the snapshot will not be re-read until the end of the transaction to ensure consistency.

Thus, due to delays, the server process does not receive the most recent statistics, but usually this is not required.

When the server is cleanly stopped, the collector flushes the statistics to permanent files in the `$PGDATA/pg_stat` directory. Thus, the statistics is saved when the server is restarted. Counters resetting occurs at the administrator command, as well as when restoring the server after a failure.



## Additional statistics

Extensions included in the PostgreSQL distribution

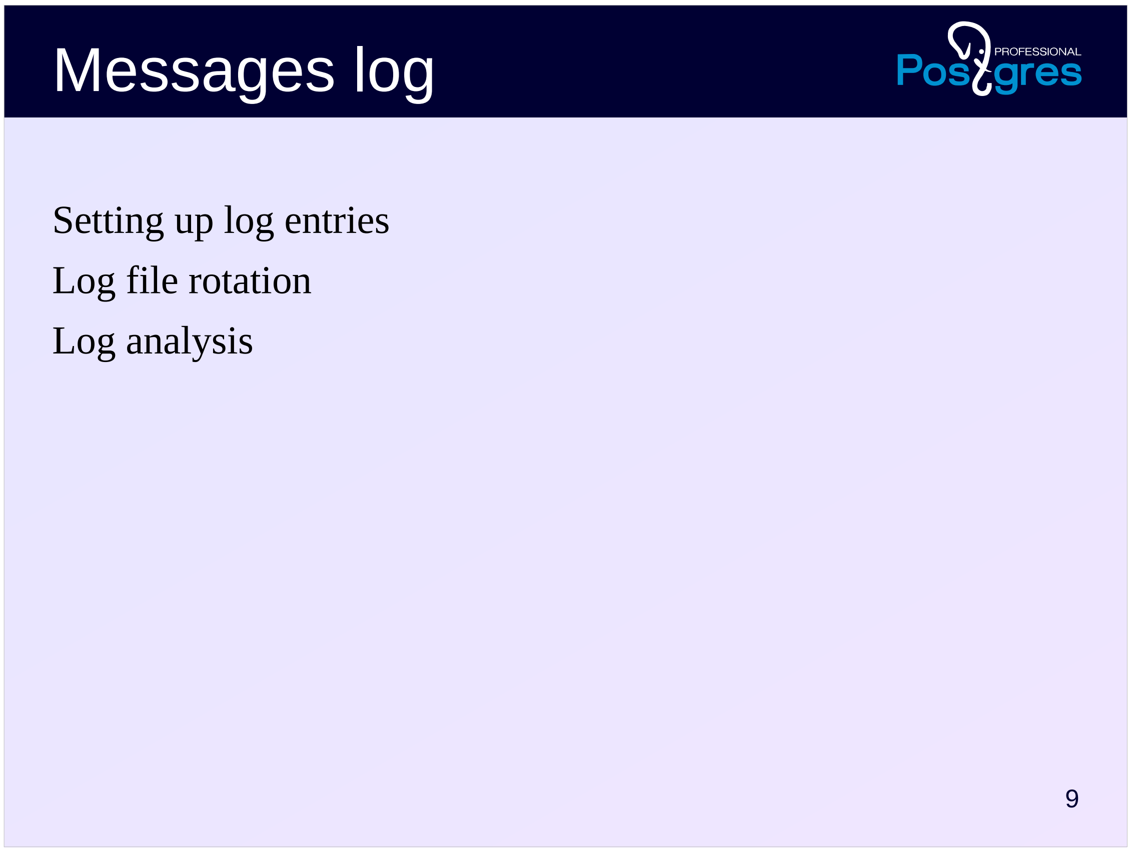
| | |
|---|---|
| pg_stat_statements | query statistics |
| pgstattuple | row versions statistics |
| pg_buffercache | buffer cache status |

Other extensions

| | |
|---|---|
| pg_stat_plans | query plan statistics |
| pg_stat_kcache | CPU and I/O statistics |
| pg_qualstats | predicate statistics |
| … | |

There are extensions that allow to collect additional statistics, both included in the official PostgreSQL distribution as well as external ones.

For example, `pg_stat_statements` extension stores information about queries performed by the DBMS; `pg_buffercache` allows you to look into the contents of the buffer cache, etc.



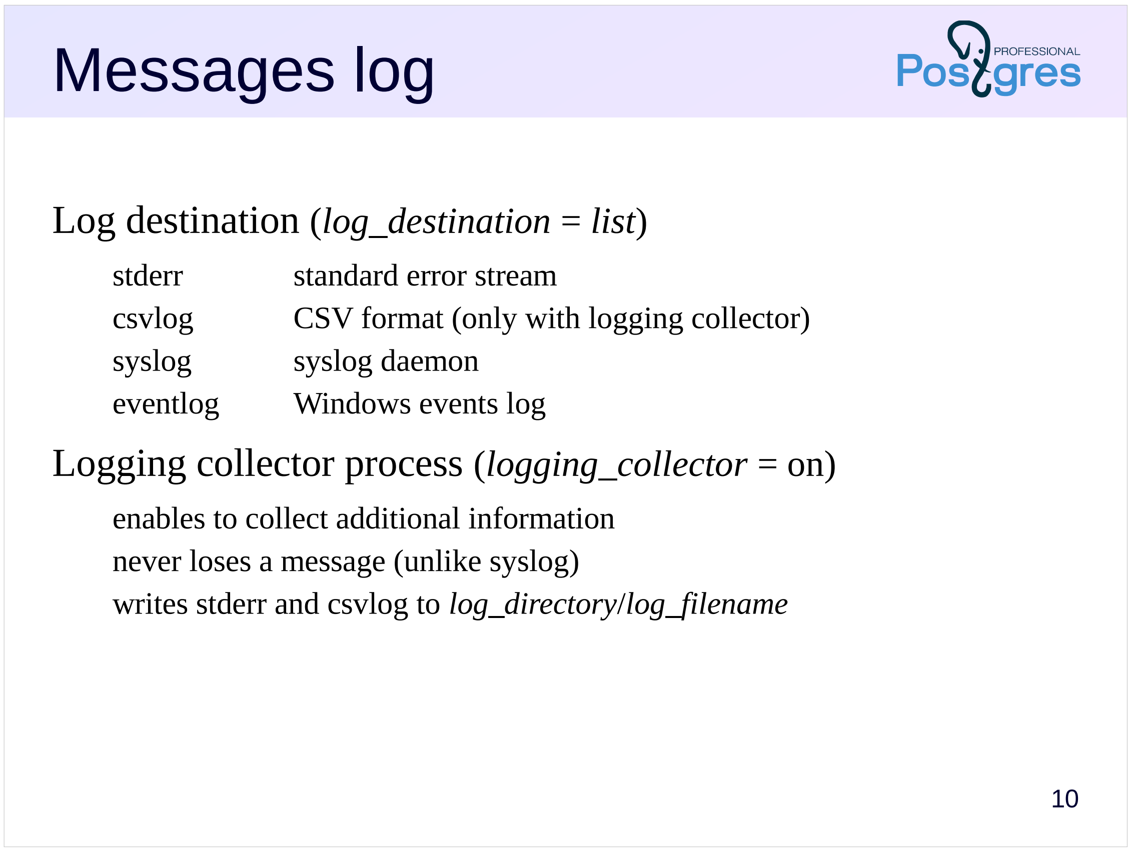
## Messages log

Setting up log entries

Log file rotation

Log analysis

The second important source of information about what is happening on the server is the messages log.



## Messages log

Log destination (*log_destination* = *list*)

| | |
|---|---|
| stderr | standard error stream |
| csvlog | CSV format (only with logging collector) |
| syslog | syslog daemon |
| eventlog | Windows events log |

Logging collector process (*logging_collector* = on)

enables to collect additional information
never loses a message (unlike syslog)
writes stderr and csvlog to *log_directory*/*log_filename*

10

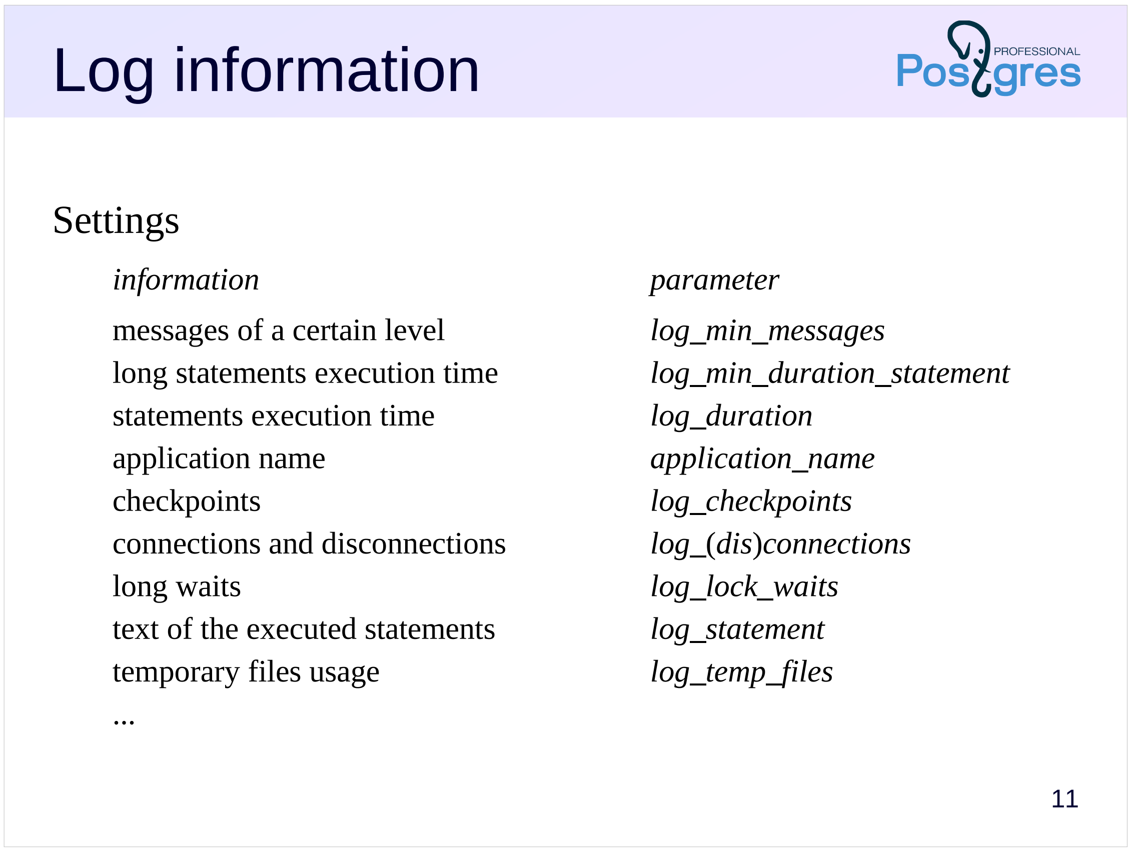
The server messages log can have different destinations and can be output in different formats. The main parameter that defines both the destination and the format is *log_destination* (you can specify one or several values separated by commas).

The value of `stderr` (the default) prints messages to the standard error stream in text form. The `syslog` value sends messages to the syslog daemon on Unix systems, and the `eventlog` sends messages to the Windows event log.

Usually an additional logging collector process is set up. It allows to write more information because it collects it from all the processes that make up PostgreSQL. It is designed to never lose a message; as a result, with a large load it can become a bottleneck.

The message collector is enabled with the *logging_collector* parameter. When `stderr` is set, information is written to the directory specified by the *log_directory* parameter in the file specified by the *log_filename* parameter.

The logging collector also allows to specify the `csvlog` destination; in this case, the information will be written in CSV format to a *log_filename* file with the `.csv` extension.

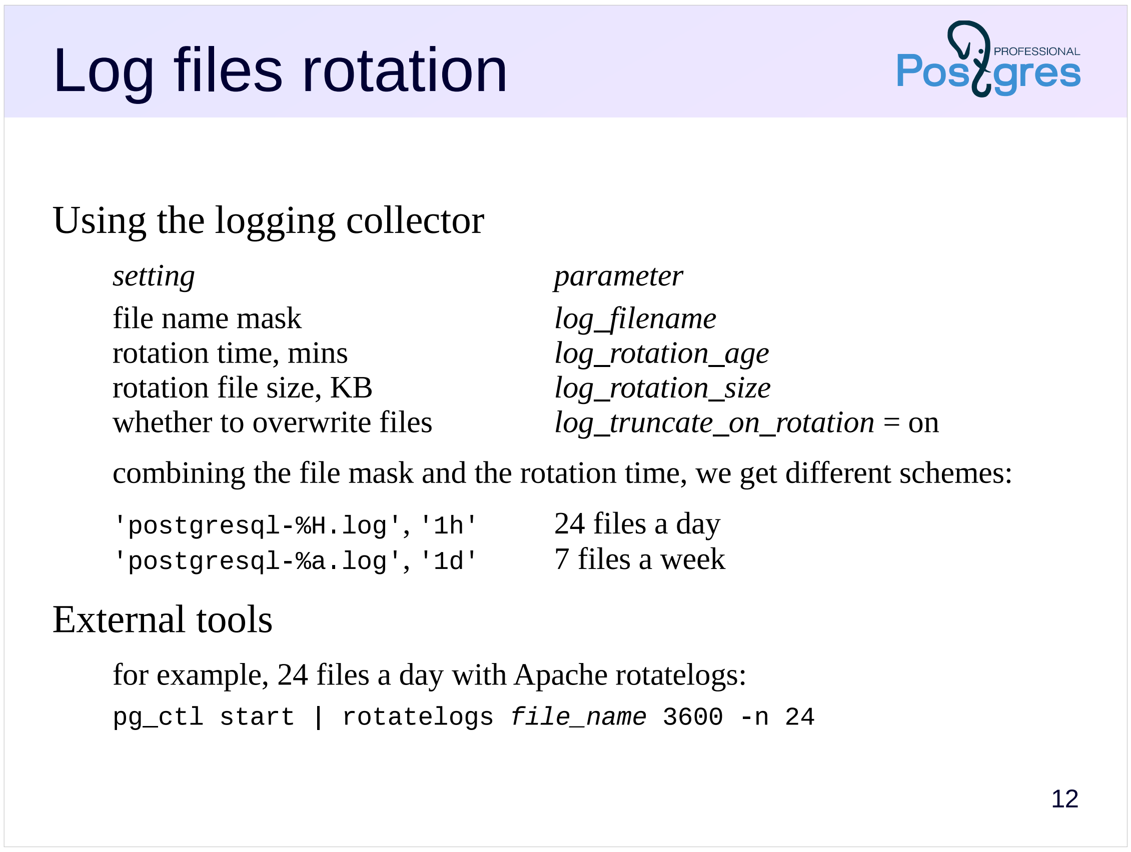


## Log information

Settings

| *information* | *parameter* |
|---|---|
| messages of a certain level | *log_min_messages* |
| long statements execution time | *log_min_duration_statement* |
| statements execution time | *log_duration* |
| application name | *application_name* |
| checkpoints | *log_checkpoints* |
| connections and disconnections | *log_(dis)connections* |
| long waits | *log_lock_waits* |
| text of the executed statements | *log_statement* |
| temporary files usage | *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 the message log into a bottleneck for the disk subsystem. The administrator must decide which information is important, provide the necessary disk space to store it, and evaluate the impact of the log collecting on the overall system performance.



## Log files rotation

Using the logging collector

| *setting* | *parameter* |
|---|---|
| file name mask | *log_filename* |
| rotation time, mins | *log_rotation_age* |
| rotation file size, KB | *log_rotation_size* |
| whether to overwrite files | *log_truncate_on_rotation* = on |

combining the file mask and the rotation time, we get different schemes:

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

External tools

for example, 24 files a day with Apache rotatelogs:

```
pg_ctl start | rotatelogs file_name 3600 -n 24
```

If you write the log into one file, sooner or later it will grow to a huge size, which is extremely inconvenient for administration and analysis. Therefore, one or another logs rotation scheme is usually used.

https://postgrespro.com/docs/postgresql/11/logfile-maintenance

The logging collector has built-in rotation ability that is configured by several parameters, the main of which are listed on the slide.

The *log_filename* parameter can specify not just the name, but the mask of the file name using some special characters for date and time.

The *log_rotation_age* parameter specifies the time to switch to the next file in minutes (and *log_rotation_size* is the size of the file at which to switch to the next).

Enabling *log_truncate_on_rotation* overwrites existing files.

Thus, by combining the mask and the switching time, you can get different rotation schemes.

https://postgrespro.com/docs/postgresql/11/runtime-config-logging.html#RUNTIME-CONFIG-LOGGING-WHERE

Alternatively, you can use external rotation programs, for example `rotatelogs`.

## Log analysis

General-purpose operating system tools

- grep, awk...

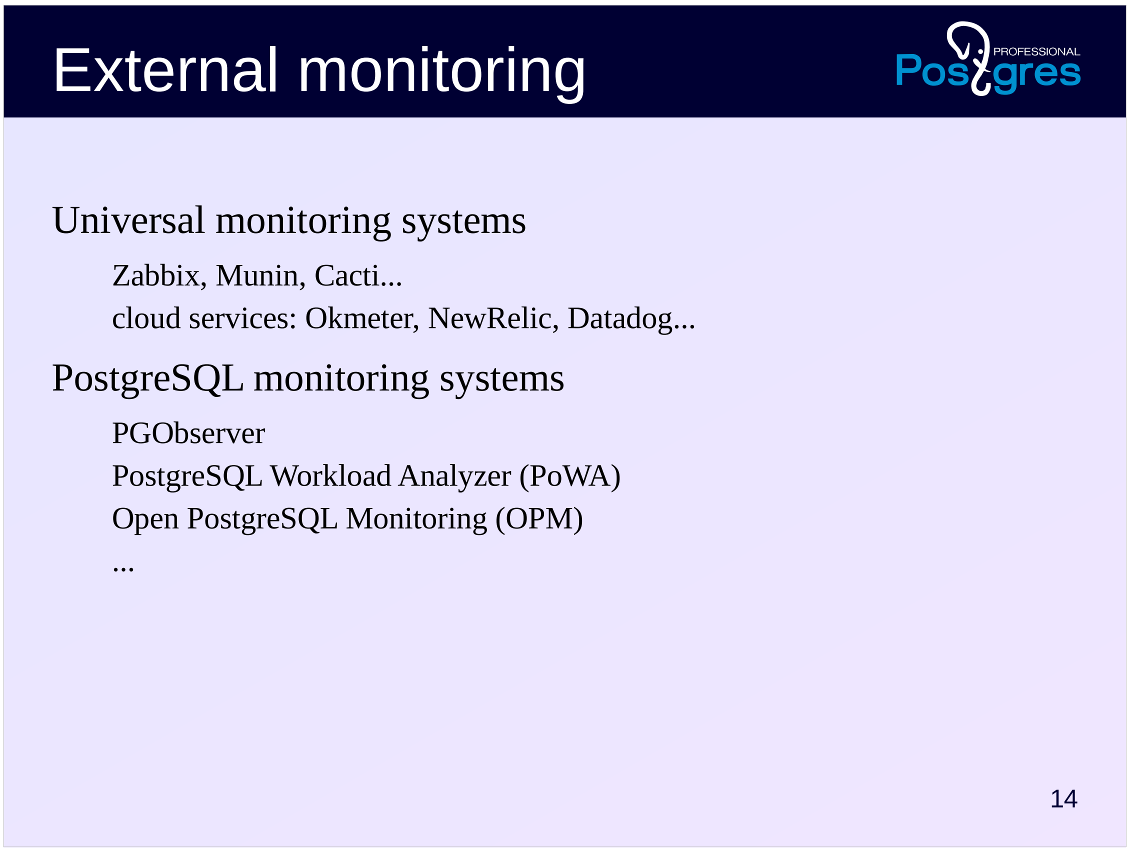
Specially designed tools

- pgBadger — requires certain log settings

You can analyze logs in different ways.

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

The de facto standard for the analysis is PgBadger utility https://github.com/dalibo/pgbadger. It imposes certain restrictions on the contents of the log. In particular, messages are allowed only in English.



## External monitoring

Universal monitoring systems

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

PostgreSQL monitoring systems

- PGObserver
- PostgreSQL Workload Analyzer (PoWA)
- Open PostgreSQL Monitoring (OPM)
- ...

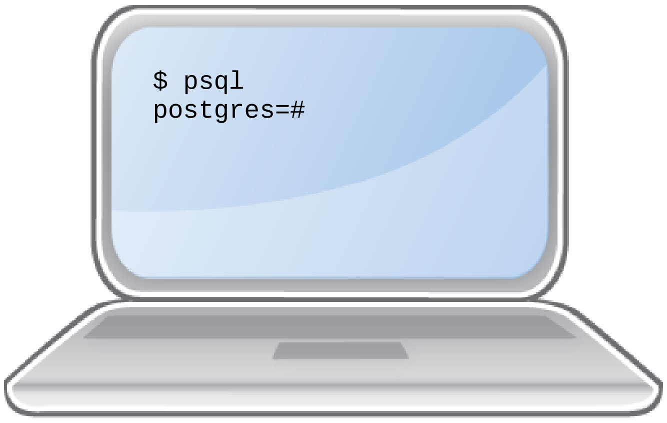
In practice, if you take the matter seriously, you need a comprehensive monitoring system that collects various metrics from both PostgreSQL and the operating system, stores the history of these metrics, displays them as graphs, and has warning features when certain metrics go beyond the established limits, etc.

PostgreSQL itself does not have such a system; it only provides the means to obtain information about yourself (which we have reviewed). Therefore, for comprehensive monitoring you need to chose an external system.

There are quite a few universal monitoring systems that have plugins or agents for PostgreSQL. These include Zabbix, Munin, Cacti, Okmeter, NewRelic, Datadog and other cloud services.There are systems specifically targeted at PostgreSQL, such as PGObserver, PoWA, OPM, etc.

Not complete, but representative list of monitoring systems can be found on page https://wiki.postgresql.org/wiki/Monitoring

## Demonstration



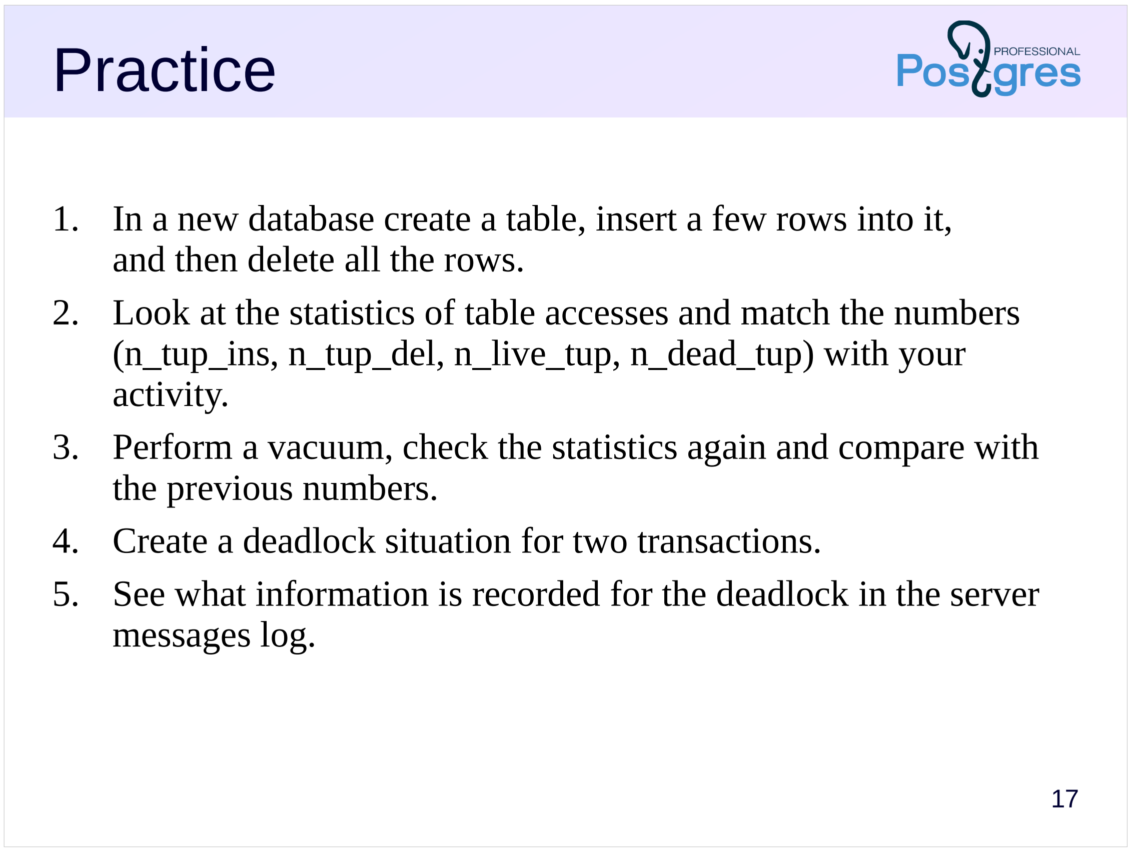
```
$ psql
postgres=#
```

## Summary

Monitoring is aims to track the server activity from both the operating system and the database perspective

PostgreSQL provides raw data in form of collected statistics and server messages log

An external system is required for comprehensive monitoring



## Practice

1. In a new database create a table, insert a few rows into it, and then delete all the rows.
2. Look at the statistics of table accesses and match the numbers (n_tup_ins, n_tup_del, n_live_tup, n_dead_tup) with your activity.
3. Perform a vacuum, check the statistics again and compare with the previous numbers.
4. Create a deadlock situation for two transactions.
5. See what information is recorded for the deadlock in the server messages log.

4. Deadlock is a situation in which two (or more) transactions are waiting for each other. Unlike regular locking, transactions do not have an opportunity to break out the deadlock and the DBMS is forced to take action: one of the transactions will be forcibly interrupted so that the others can continue execution.

The easiest way to reproduce the deadlock is on a table with two rows. The first transaction changes (and, accordingly, locks) the first row, and the second transactions changes the second row. Then the first transaction tries to change the second row and «hangs» on the lock. And then the second transaction tries to change the first row and also waits for the lock to be released.