

# Задачи администрирования Мониторинг



## Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Алексей Береснев

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

## Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Средства операционной системы  
Накопительная статистика сервера  
Журнал сообщений сервера  
Внешние системы мониторинга

## Процессы

ps, pgrep...

параметр *update\_process\_title* для обновления статуса процессов

параметр *cluster\_name* для установки имени кластера

## Использование ресурсов

iostat, vmstat, sar, top...

## Дисковое пространство

df, du, quota...

PostgreSQL работает под управлением операционной системы и в известной степени зависит от ее настроек.

Используя инструменты операционной системы, можно посмотреть информацию о процессах PostgreSQL. При включенном (по умолчанию) параметре сервера *update\_process\_title* в имени процесса отображается его текущее состояние. Параметр *cluster\_name* задает имя экземпляра, по которому его можно отличать в списке процессов.

Для изучения использования системных ресурсов (процессор, память, диски) в Unix имеются различные инструменты: iostat, vmstat, sar, top и др.

Необходимо следить и за размером дискового пространства. Место, занимаемое базой данных, можно смотреть как из самой БД (см. модуль «Организация данных»), так из ОС (команда du). Размер доступного дискового пространства надо смотреть в ОС (команда df). Если используются дисковые квоты, надо принимать во внимание и их.

В целом набор инструментов и подходы могут сильно различаться в зависимости от используемой ОС и файловой системы, поэтому подробно здесь не рассматриваются.

<https://postgrespro.ru/docs/postgresql/16/monitoring-ps>

<https://postgrespro.ru/docs/postgresql/16/diskusage>

Процесс сбора статистики

Текущие активности системы

Отслеживание выполнения команд

Дополнительные расширения

Существует два основных источника информации о происходящем в системе. Первый из них — статистическая информация, которая собирается PostgreSQL и хранится в кластере.

## Настройки накопительной статистики

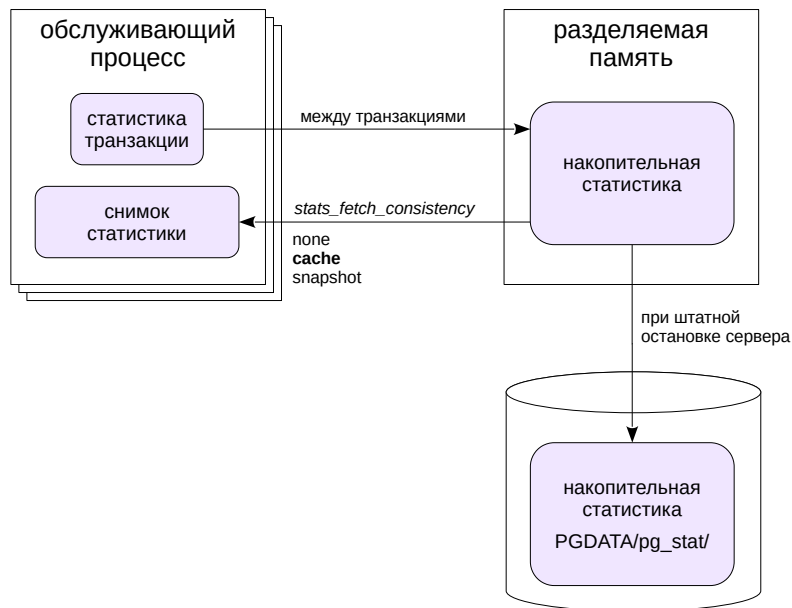
<i>параметр</i>	<i>действие</i>
<i>track_activities</i>	включает мониторинг текущих команд
<i>track_counts</i>	сбор статистики по обращениям к таблицам и индексам
<i>track_functions</i>	отслеживание использования пользовательских функций выключен по умолчанию
<i>track_io_timing</i>	мониторинг времени чтения и записи блоков выключен по умолчанию
<i>track_wal_io_timing</i>	мониторинг времени записи WAL выключен по умолчанию

Система накопительной статистики в PostgreSQL собирает и позволяет получать информацию о работе сервера. Накопительная статистика отслеживает обращения к таблицам и индексам как на уровне блоков на диске, так и на уровне отдельных строк. Кроме того, для каждой таблицы собираются сведения о количестве строк и действиях по очистке и анализу.

Можно также учитывать количество вызовов пользовательских функций и время, затраченное на их выполнение.

Количеством собираемой информации управляют несколько параметров сервера, так как чем больше информации собирается, тем больше и накладные расходы.

<https://postgrespro.ru/docs/postgresql/16/monitoring-stats>



Обслуживающие процессы собирают статистику в рамках транзакций. Затем эта статистика самим процессом записывается в разделяемую память, но не чаще, чем раз в одну секунду (задано при компиляции).

Накопительная статистика запоминается в PGDATA/pg\_stat/ при штатной остановке сервера и считывается при его запуске. При аварийной остановке все счетчики сбрасываются.

Обслуживающий процесс может кешировать данные статистики при обращении к ней. Уровнем кеширования управляет параметр *stats\_fetch\_consistency*:

- none — без кеширования, статистика только в разделяемой памяти;
- cache — кешируется статистика по одному объекту;
- snapshot — кешируется вся статистика текущей базы данных.

По умолчанию используется значение *cache* — это компромисс между согласованностью и эффективностью.

Закешированная статистика не перечитывается и сбрасывается в конце транзакции или при вызове `pg_stat_clear_snapshot()`.

Из-за задержек и кеширования обслуживающий процесс использует не самую свежую статистику, но обычно это и не требуется.

## Накопительная статистика

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

```
CREATE DATABASE
```

```
=> \c admin_monitoring
```

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

Вначале включим сбор статистики ввода-вывода:

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

```
ALTER SYSTEM
```

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

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

Смотреть на активности сервера имеет смысл, когда какие-то активности на самом деле есть. Чтобы симитировать нагрузку, воспользуемся pgbench — штатной утилитой для запуска эталонных тестов.

Сначала утилита создает набор таблиц и заполняет их данными.

```
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 0.61 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 1.03 s (drop tables 0.00 s, create tables 0.02 s, client-side generate 0.69 s,
vacuum 0.14 s, primary keys 0.19 s).
```

Сбросим накопленную ранее статистику по базе данных:

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

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

А также статистику экземпляра по вводу-выводу:

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

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

Запускаем тест TPC-B на несколько секунд:

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

```
pgbench (16.9 (Ubuntu 16.9-1.pgdg24.04+1))
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: 1360
number of failed transactions: 0 (0.000%)
```

latency average = 7.352 ms  
initial connection time = 3.828 ms  
tps = 136.022702 (without initial connection time)

Теперь мы можем посмотреть статистику обращений к таблицам в терминах строк:

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

-[ RECORD 1 ]-----+-----
relid          | 16393
schemaname     | public
relname        | pgbench_accounts
seq_scan       | 0
last_seq_scan  |
seq_tup_read   | 0
idx_scan       | 2720
last_idx_scan  | 2025-06-23 23:25:35.249459+03
idx_tup_fetch  | 2720
n_tup_ins      | 0
n_tup_upd      | 1360
n_tup_del      | 0
n_tup_hot_upd  | 429
n_tup_newpage_upd | 931
n_live_tup     | 0
n_dead_tup     | 1248
n_mod_since_analyze | 1360
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
```

---

И в терминах страниц:

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

-[ RECORD 1 ]---+-----
relid          | 16393
schemaname     | public
relname        | pgbench_accounts
heap_blks_read | 0
heap_blks_hit  | 9199
idx_blks_read  | 274
idx_blks_hit   | 7032
toast_blks_read |
toast_blks_hit |
tidx_blks_read |
tidx_blks_hit  |
```

---

Существуют аналогичные представления для индексов:

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

-[ RECORD 1 ]-----+-----
relid          | 16393
indexrelid     | 16407
schemaname     | public
relname        | pgbench_accounts
indexrelname    | pgbench_accounts_pkey
idx_scan       | 2720
last_idx_scan  | 2025-06-23 23:25:35.249459+03
idx_tup_read   | 3654
idx_tup_fetch  | 2720
```

---

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



```
-[ RECORD 1 ]-----
reloid       | 16393
indexrelid   | 16407
schemaname   | public
relname      | pgbench_accounts
indexrelname  | pgbench_accounts_pkey
idx_blks_read | 274
idx_blks_hit  | 7032
```

Эти представления, в частности, могут помочь определить неиспользуемые индексы. Такие индексы не только бессмысленно занимают место на диске, но и тратят ресурсы на обновление при каждом изменении данных в таблице.

Есть также представления для пользовательских и системных объектов (all, user, sys), для статистики текущей транзакции (pg\_stat\_xact\*) и другие.

Можно посмотреть общую статистику по базе данных:

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

```
-[ RECORD 1 ]-----
datid          | 16386
datname        | admin_monitoring
numbackends    | 1
xact_commit    | 1376
xact_rollback  | 0
blks_read      | 276
blks_hit       | 26827
tup_returned   | 20098
tup_fetched    | 3351
tup_inserted   | 1360
tup_updated    | 4081
tup_deleted    | 0
conflicts      | 0
temp_files     | 0
temp_bytes     | 0
deadlocks      | 0
checksum_failures |
checksum_last_failure |
blk_read_time  | 15.291
blk_write_time | 1.131
session_time   | 21463.672
active_time    | 9224.644
idle_in_transaction_time | 682.323
sessions       | 2
sessions_abandoned | 0
sessions_fatal | 0
sessions_killed | 0
stats_reset    | 2025-06-23 23:25:25.082128+03
```

Здесь есть много полезной информации о количестве произошедших взаимоблокировок, зафиксированных и отмененных транзакций, использовании временных файлов, ошибках подсчета контрольных сумм. Здесь же хранится статистика общего количества сеансов и количества прерванных по разным причинам сеансов.

Столбец numbackends показывает текущее количество обслуживающих процессов, подключенных к базе данных.

Статистика ввода-вывода на уровне сервера доступна в представлении pg\_stat\_io. Например, выполним контрольную точку и посмотрим количество операций чтения и записи страниц по типам процессов:

```
=> 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
client backend	27953	276	0
walsender	0	0	0
standalone backend	0	0	0
autovacuum worker	584	0	0
autovacuum launcher	0	0	0
background writer			0
startup	0	0	0
checkpointer			2878
(9 rows)			

## Настройка

*статистика*

текущие активности  
и ожидания обслуживающих  
и фоновых процессов

*параметр*

*track\_activities*  
включен по умолчанию

Текущие активности всех обслуживающих и фоновых процессов отображаются в представлении `pg_stat_activity`. Подробнее на нем мы остановимся в демонстрации.

Работа этого представления зависит от параметра *track\_activities*, включенного по умолчанию.

## Текущие активности

Воспроизведем сценарий, в котором один процесс блокирует выполнение другого, и попробуем разобраться в ситуации с помощью системных представлений.

Создадим таблицу с одной строкой:

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

CREATE TABLE

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

INSERT 0 1

Запустим два сеанса, один из которых изменяет таблицу и не завершает транзакцию:

```
students$ psql -d admin monitoring
```

```
=> BEGIN;
```

| BEGIN

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

UPDATE 1

А второй пытается изменить ту же строку и блокируется:

```
students$ psql -d admin monitoring
```

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

Посмотрим информацию об обслуживающих процессах:

```
=> 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           | 20272
query         | UPDATE t SET n = n + 1;
state         | idle in transaction
wait_event    | ClientRead
wait_event_type | Client
pg_blocking_pids | {}
-[ RECORD 2
]-----+-----
pid           | 19290
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           | 20361
query         | UPDATE t SET n = n + 2;
state         | active
wait_event    | transactionid
wait_event_type | Lock
pg_blocking_pids | {20272}

```

Состояние «idle in transaction» означает, что сеанс начал транзакцию, но в настоящее время ничего не делает, а транзакция осталась незавершенной. Это может стать проблемой, если ситуация возникает систематически (например, из-за некорректной реализации приложения или из-за ошибок в драйвере), поскольку открытый сеанс удерживает снимок данных и таким образом препятствует очистке.

В арсенале администратора имеется параметр `idle_in_transaction_session_timeout`, позволяющий принудительно завершать сеансы, в которых транзакция простаивает больше указанного времени. Также имеется параметр `idle_session_timeout` — принудительно завершает сеансы, простаивающие больше указанного времени вне транзакции.

А мы покажем, как завершить блокирующий сеанс вручную. Сначала узнаем номер заблокированного процесса при помощи функции `pg_blocking_pids`:

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

```
blocked_pid
-----
20361
(1 row)
```

Блокирующий процесс можно вычислить и без функции `pg_blocking_pids`, используя запросы к таблице блокировок. Запрос покажет две строки: одна транзакция получила блокировку (`granted`), а другая ее ожидает.

```
=> SELECT locktype, transactionid, pid, mode, granted
FROM pg_locks
WHERE transactionid IN (
  SELECT transactionid FROM pg_locks WHERE pid = 20361 AND NOT granted
);
```

locktype	transactionid	pid	mode	granted
transactionid	2112	20272	ExclusiveLock	t
transactionid	2112	20361	ShareLock	f

(2 rows)

В общем случае нужно аккуратно учитывать тип блокировки.

Выполнение запроса можно прервать функцией `pg_cancel_backend`. В нашем случае транзакция простаивает, так что просто прерываем сеанс, вызвав `pg_terminate_backend`:

```
=> SELECT pg_terminate_backend(b.pid)
FROM unnest(pg_blocking_pids(20361)) AS b(pid);
```

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

Функция `unnest` нужна, поскольку `pg_blocking_pids` возвращает массив идентификаторов процессов, блокирующих искомый обслуживающий процесс. В нашем примере блокирующий процесс один, но в общем случае их может быть несколько.

Подробнее о блокировках рассказывается в курсе DBA2.

Проверим состояние обслуживающих процессов.

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

```
-[ RECORD 1 ]-----+-----
pid          | 19290
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          | 20361
query        | UPDATE t SET n = n + 2;
state        | idle
wait_event   | ClientRead
wait_event_type | Client
```

Осталось только два, причем заблокированный успешно завершил транзакцию.

Представление `pg_stat_activity` показывает информацию не только про обслуживающие процессы, но и про служебные фоновые процессы экземпляра:

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

pid	backend_type	backend_start	state
19194	autovacuum launcher	2025-06-23 23:25:15.787314+03	
19195	logical replication launcher	2025-06-23 23:25:15.793676+03	
19290	client backend	2025-06-23 23:25:23.674757+03	active
20361	client backend	2025-06-23 23:25:41.083253+03	idle
19191	background writer	2025-06-23 23:25:15.719773+03	
19190	checkpointer	2025-06-23 23:25:15.718874+03	
19193	walwriter	2025-06-23 23:25:15.784309+03	

(7 rows)

Сравним с тем, что показывает операционная система:

```
student$ sudo head -n 1 /var/lib/postgresql/16/main/postmaster.pid
```

19189

```
student$ ps -o pid,command --ppid 19189
```

```

PID COMMAND
19190 postgres: 16/main: checkpointer
19191 postgres: 16/main: background writer
19193 postgres: 16/main: walwriter
19194 postgres: 16/main: autovacuum launcher
19195 postgres: 16/main: logical replication launcher
19290 postgres: 16/main: student admin_monitoring [local] idle
20361 postgres: 16/main: student admin_monitoring [local] idle
```

## Представления для отслеживания выполнения

<i>команда</i>	<i>представление</i>
ANALYZE	pg_stat_progress_analyze
CREATE INDEX, REINDEX	pg_stat_progress_create_index
VACUUM включая процессы автоочистки	pg_stat_progress_vacuum
CLUSTER, VACUUM FULL	pg_stat_progress_cluster
Создание базовой резервной копии	pg_stat_progress_basebackup
COPY	pg_stat_progress_copy

Следить за ходом выполнения некоторых потенциально долгих команд можно, выполняя запросы к соответствующим представлениям.

Структуры представлений описаны в документации:

<https://postgrespro.ru/docs/postgresql/16/progress-reporting>

Создание резервных копий рассматривается в модуле «Резервное копирование».

## Расширения в поставке

<code>pg_stat_statements</code>	статистика по запросам
<code>pgstattuple</code>	статистика по версиям строк
<code>pg_bufferscache</code>	состояние буферного кеша

## Другие расширения

<code>pg_wait_sampling</code>	статистика ожиданий
<code>pg_stat_kcache</code>	статистика по процессору и вводу-выводу
<code>pg_qualstats</code>	статистика по предикатам
...	

Существуют расширения, позволяющие собирать дополнительную статистику, как входящие в поставку, так и внешние.

Например, расширение `pg_stat_statements` сохраняет информацию о запросах, выполняемых СУБД; `pg_bufferscache` позволяет заглянуть в содержимое буферного кеша и т. п.

Многие важные расширения рассматриваются в курсах DBA2 и DEV2.



Настройка журнальных записей

Ротация файлов журнала

Анализ журнала

Второй важный источник информации о происходящем на сервере — журнал сообщений.

## Приемник сообщений (*log\_destination = cnuсок*)

<code>stderr</code>	поток ошибок
<code>csvlog</code>	формат CSV (только с коллектором)
<code>jsonlog</code>	формат JSON (только с коллектором)
<code>syslog</code>	демон syslog
<code>eventlog</code>	журнал событий Windows

## Коллектор сообщений (*logging\_collector = on*)

позволяет собирать дополнительную информацию  
никогда не теряет сообщения (в отличие от syslog)  
записывает `stderr`, `csvlog` и `jsonlog` в файл *log\_directory/log\_filename*

Журнал сообщений сервера можно направлять в разные приемники и выводить в разных форматах. Основной параметр, который определяет приемник и формат — `log_destination` (можно указать один или несколько приемников через запятую).

Значение `stderr` (установленное по умолчанию) выводит сообщения в стандартный поток ошибок в текстовом виде. Значение `syslog` направляет сообщения демону `syslog` в Unix-системах, а `eventlog` — в журнал событий Windows.

Обычно дополнительно включают специальный процесс — коллектор сообщений. Он позволяет записать больше информации, поскольку собирает ее со всех процессов, составляющих PostgreSQL. Он спроектирован так, что никогда не теряет сообщения; как следствие, при большой нагрузке он может стать узким местом.

Коллектор сообщений включается параметром *logging\_collector*. При значении `stderr` информация записывается в каталог, определяемый параметром *log\_directory*, в файл, определяемый параметром *log\_filename*.

Включенный коллектор сообщений позволяет также указать приемник `csvlog`; в этом случае информация будет сбрасываться в формате CSV в файл *log\_filename* с расширением `csv`. При использовании приемника `jsonlog` содержимое файла отчета будет записываться в формате JSON, а имя файла будет иметь расширение `json`.

## Настройки

<i>информация</i>	<i>параметр</i>
сообщения определенного уровня	<i>log_min_messages</i>
время выполнения длинных команд	<i>log_min_duration_statement</i>
время выполнения команд	<i>log_duration</i>
имя приложения	<i>application_name</i>
контрольные точки	<i>log_checkpoints</i>
подключения и отключения	<i>log_(dis)connections</i>
длинные ожидания	<i>log_lock_waits</i>
текст выполняемых команд	<i>log_statement</i>
использование временных файлов	<i>log_temp_files</i>
...	

В журнал сообщений сервера можно выводить множество полезной информации. По умолчанию почти весь вывод отключен, чтобы не превратить запись журнала в узкое место для подсистемы ввода-вывода. Администратор должен решить, какая информация важна, обеспечить необходимое место на диске для ее хранения и оценить влияние записи журнала на общую производительность системы.

## С помощью коллектора сообщений

<i>настройка</i>	<i>параметр</i>
маска имени файла	<code>log_filename</code>
время ротации, мин	<code>log_rotation_age</code>
размер файла для ротации, КБ	<code>log_rotation_size</code>
перезаписывать ли файлы	<code>log_truncate_on_rotation = on</code>
комбинируя маску файла и время ротации, получаем разные схемы:	
<code>'postgresql-%N.log', '1h'</code>	24 файла в сутки
<code>'postgresql-%a.log', '1d'</code>	7 файлов в неделю

## Внешние средства

-  системная утилита `logrotate`

15

Если записывать журнал в один файл, рано или поздно он вырастет до огромных размеров, что крайне неудобно для администрирования и анализа. Поэтому обычно используется та или иная схема ротации журналов.

<https://postgrespro.ru/docs/postgresql/16/logfile-maintenance>

Коллектор сообщений имеет встроенные средства ротации, которые настраиваются несколькими параметрами, основные из которых приведены на слайде.

Параметр `log_filename` позволяет задавать не просто имя, а маску имени файла с помощью спецсимволов даты и времени.

Параметр `log_rotation_age` задает время переключения на следующий файл в минутах (а `log_rotation_size` — размер файла, при котором надо переключаться на следующий).

Включение `log_truncate_on_rotation` перезаписывает уже существующие файлы.

Таким образом, комбинируя маску и время переключения, можно получать разные схемы ротации.

<https://postgrespro.ru/docs/postgresql/16/runtime-config-logging#RUNTIME-CONFIG-LOGGING-WHERE>

В качестве альтернативы можно воспользоваться внешними программами ротации, например пакетный дистрибутив для Ubuntu использует системную утилиту `logrotate` (ее настройки находятся в файле `/etc/logrotate.d/postgresql-common`).

## Средства операционной системы

grep, awk...

## Специальные средства анализа

pgBadger — требует определенных настроек журнала

Анализировать журналы можно по-разному.

Можно искать определенную информацию средствами ОС или специально разработанными скриптами.

Стандартом де-факто для анализа является программа PgBadger <https://github.com/darold/pgbadger>, но надо иметь в виду, что она накладывает определенные ограничения на содержимое журнала. В частности, допускаются сообщения только на английском языке.

## Анализ журнала

Посмотрим самый простой случай. Например, нас интересуют сообщения FATAL:

```
student$ sudo grep FATAL /var/log/postgresql/postgresql-16-main.log | tail -n 10
```

```
2025-06-23 23:21:10.187 MSK [2665] student@student FATAL: terminating connection due to
administrator command
2025-06-23 23:25:42.484 MSK [20272] student@admin_monitoring FATAL: terminating
connection due to administrator command
```

Сообщение «terminating connection» вызвано тем, что мы завершали блокирующий процесс.

---

Обычное применение журнала — анализ наиболее продолжительных запросов. Включим вывод всех команд и времени их выполнения:

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

```
ALTER SYSTEM
```

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

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

Теперь выполним какую-нибудь команду:

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

```
sum
-----
500095.8921310923
(1 row)
```

И посмотрим журнал:

```
student$ sudo tail -n 1 /var/log/postgresql/postgresql-16-main.log
```

```
2025-06-23 23:25:43.228 MSK [19290] student@admin_monitoring LOG: duration: 150.194 ms
statement: SELECT sum(random()) FROM generate_series(1,1_000_000);
```

## Универсальные системы мониторинга

Zabbix, Munin, Cacti...  
в облаке: Okmeter, NewRelic, Datadog...

## Системы мониторинга PostgreSQL

pg\_profile, pgpro\_pwr  
PGObserver  
PostgreSQL Workload Analyzer (PoWA)  
Open PostgreSQL Monitoring (OPM)  
...

На практике требуется полноценная система мониторинга, которая собирает различные метрики как с PostgreSQL, так и с операционной системы, хранит историю этих метрик, отображает их в виде понятных графиков, имеет средства оповещения при выходе определенных метрик за установленные границы и т. д.

Собственно PostgreSQL не располагает такой системой; он только предоставляет средства для получения информации о себе (которые мы рассмотрели). Поэтому для полноценного мониторинга нужно выбрать внешнюю систему. Таких систем существует довольно много. Есть универсальные системы, имеющие плагины или агенты для PostgreSQL. К ним относятся Zabbix, Munin, Cacti, облачные сервисы Okmeter, NewRelic, Datadog и другие.

Есть и системы, ориентированные специально на PostgreSQL, такие, как PGObserver, PoWA, OPM и т. д. Расширение pg\_profile позволяет строить снимки статических данных и сравнивать их, выявляя ресурсоемкие операции и их динамику. Расширенная коммерческая версия этого расширения — pgpro\_pwr.

<https://postgrespro.ru/docs/enterprise/16/pgpro-pwr>

Неполный, но представительный список систем мониторинга можно посмотреть на странице <https://wiki.postgresql.org/wiki/Monitoring>

Для более глубокого погружения в эту тему можно прочитать книгу Алексея Лесовского «Мониторинг PostgreSQL»:  
<https://edu.postgrespro.ru/monitoring.pdf>

Мониторинг заключается в контроле работы сервера  
как со стороны операционной системы,  
так и со стороны самого сервера

PostgreSQL предоставляет накопительную статистику  
и журнал сообщений сервера

Для полноценного мониторинга требуется внешняя система



1. В новой базе данных создайте таблицу, выполните вставку нескольких строк, а затем удалите все строки.  
Посмотрите статистику обращений к таблице и сопоставьте цифры (`n_tup_ins`, `n_tup_del`, `n_live_tup`, `n_dead_tup`) с вашей активностью.  
Выполните очистку (`vacuum`), снова проверьте статистику и сравните с предыдущими цифрами.
2. Создайте ситуацию взаимоблокировки двух транзакций.  
Посмотрите, какая информация записывается при этом в журнал сообщений сервера.

2. Взаимоблокировка (deadlock) — ситуация, в которой две (или больше) транзакций ожидают друг друга. В отличие от обычной блокировки при взаимоблокировке у транзакций нет возможности выйти из этого «тупика» и СУБД вынуждена принимать меры — одна из транзакций будет принудительно прервана, чтобы остальные могли продолжить выполнение.

Проще всего воспроизвести взаимоблокировку на таблице с двумя строками. Первая транзакция меняет (и, соответственно, блокирует) первую строку, а вторая — вторую. Затем первая транзакция пытается изменить вторую строку и «повисает» на блокировке. А потом вторая транзакция пытается изменить первую строку — и тоже ждет освобождения блокировки.

## Статистика обращений к таблице

Создаем базу данных и таблицу:

```
=> 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

Проверяем статистику обращений.

```
=> 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   | 2025-06-23 23:33:48.69345+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
```

Мы вставили 1000 строк (n\_tup\_ins = 1000), удалили 1000 строк (n\_tup\_del = 1000).

После этого не осталось активных версий строк (n\_live\_tup = 0), все 1000 строк не актуальны на текущий момент (n\_dead\_tup = 1000).

Выполним очистку.

```
=> 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   | 2025-06-23 23:33:48.69345+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     | 2025-06-23 23:33:49.953862+03
last_autovacuum |
last_analyze    |
last_autoanalyze |
vacuum_count    | 1
autovacuum_count | 0
analyze_count   | 0
autoanalyze_count | 0

```

Неактуальные версии строк убраны при очистке (n\_dead\_tup = 0), очистка обрабатывала таблицу один раз (vacuum\_count = 1).

## 2. Взаимоблокировка

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

```
INSERT 0 2
```

Одна транзакция блокирует первую строку таблицы...

```
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

```

Затем другая транзакция блокирует вторую строку...

```
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

```

Теперь первая транзакция пытается изменить вторую строку и ждет ее освобождения...

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

А вторая транзакция пытается изменить первую строку...

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

...и происходит взаимоблокировка. Сервер обрывает одну из транзакций:

```
ERROR: deadlock detected
DETAIL: Process 39842 waits for ShareLock on transaction 739; blocked by process 39713.
Process 39713 waits for ShareLock on transaction 740; blocked by process 39842.
HINT: See server log for query details.

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

Другая транзакция разблокируется:

```
| UPDATE 1
```

Проверим информацию в журнале сообщений:

```
student$ sudo tail -n 8 /var/log/postgresql/postgresql-16-main.log
```

```
2025-06-23 23:33:53.254 MSK [39842] student@admin_monitoring ERROR: deadlock detected
2025-06-23 23:33:53.254 MSK [39842] student@admin_monitoring DETAIL: Process 39842 waits
for ShareLock on transaction 739; blocked by process 39713.
    Process 39713 waits for ShareLock on transaction 740; blocked by process 39842.
    Process 39842: UPDATE t SET n = 100 WHERE n = 1;
    Process 39713: UPDATE t SET n = 20 WHERE n = 2;
2025-06-23 23:33:53.254 MSK [39842] student@admin_monitoring HINT: See server log for
query details.
2025-06-23 23:33:53.254 MSK [39842] student@admin_monitoring CONTEXT: while updating
tuple (0,1) in relation "t"
2025-06-23 23:33:53.254 MSK [39842] student@admin_monitoring STATEMENT: UPDATE t SET n =
100 WHERE n = 1;
```

1. Установите расширение `pg_stat_statements`.  
Выполните несколько произвольных запросов.  
Посмотрите, какую информацию показывает представление `pg_stat_statements`.

1. Для установки расширения потребуется перед выполнением команды `CREATE EXTENSION` изменить значение параметра `shared_preload_libraries` с последующей перезагрузкой сервера.  
<https://postgrespro.ru/docs/postgresql/16/pgstatstatements>

## 1. Расширение pg\_stat\_statements

Расширение собирает статистику планирования и выполнения всех запросов.

Для работы расширения требуется загрузить одноименный модуль. Для этого имя модуля нужно прописать в параметре `shared_preload_libraries` и перезагрузить сервер. Изменять этот параметр лучше в файле `postgresql.conf`, но для целей демонстрации установим параметр с помощью команды `ALTER SYSTEM`.

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

```
ALTER SYSTEM
```

```
=> \q
```

```
student$ sudo pg_ctlcluster 16 main restart
```

```
student$ psql
```

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

```
CREATE DATABASE
```

```
=> \c admin_monitoring
```

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

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

```
CREATE EXTENSION
```

Теперь выполним несколько запросов.

```
=> 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
```

Посмотрим на статистику запроса, который выполнялся чаще всего.

```
=> SELECT query, calls, total_exec_time
FROM pg_stat_statements
ORDER BY calls DESC LIMIT 1;
```

query	calls	total_exec_time
INSERT INTO t VALUES (\$1)	5	0.170704

(1 row)

Разделяемая библиотека больше не требуется, восстановим исходное значение параметра:

```
=> ALTER SYSTEM RESET shared_preload_libraries;
```

```
ALTER SYSTEM
```

```
=> \q
```

```
student$ sudo pg_ctlcluster 16 main restart
```