

Задачи администрирования Мониторинг



Авторские права

© Postgres Professional, 2015–2022

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

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

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

Статистика внутри базы

Журнал сообщений сервера

Внешние системы мониторинга

Процессы

ps (grep postgres)

параметр *update_process_title* для обновления статуса процессов

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.

Unix предоставляет множество инструментов для анализа состояния и производительности.

В частности, можно посмотреть процессы, принадлежащие PostgreSQL. Это особенно полезно при включенном (по умолчанию) параметре сервера *update_process_title*, когда в имени процесса отображается его текущее состояние.

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

Необходимо следить и за размером дискового пространства. Место, занимаемое базой данных, можно посмотреть 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 может сильно различаться в зависимости от используемой ОС и файловой системы, поэтому подробно здесь не рассматриваются.

<https://postgrespro.ru/docs/postgresql/13/monitoring-ps>

<https://postgrespro.ru/docs/postgresql/13/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 внутри базы данных.

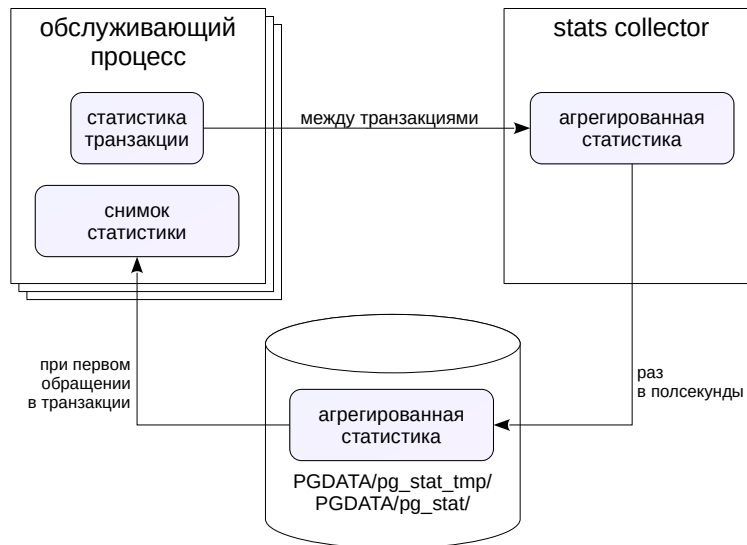
Настройки процесса stats collector

<i>статистика</i>	<i>параметр</i>
обращения к таблицам и индексам (доступы, затронутые строки)	<i>track_counts</i> включен по умолчанию и нужен для автоочистки
обращения к страницам	<i>track_io_timing</i> выключен по умолчанию
вызовы пользовательских функций	<i>track_functions</i> выключен по умолчанию

Кроме показа действий, непосредственно происходящих в данный момент, PostgreSQL собирает и некоторую статистику.

Сбором статистики занимается фоновый процесс stats collector. Количеством собираемой информации управляют несколько параметров сервера, так как чем больше информации собирается, тем больше и накладные расходы.

<https://postgrespro.ru/docs/postgresql/13/monitoring-stats>



Каждый обслуживающий процесс собирает необходимую статистику в рамках каждой выполняемой транзакции. Затем эта статистика передается процессу-коллектору. Коллектор собирает и агрегирует статистику со всех обслуживающих процессов. Раз в полсекунды (время настраивается при компиляции) коллектор сбрасывает статистику во временные файлы в каталог PGDATA/pg_stat_tmp. (Поэтому перенесение этого каталога в файловую систему в памяти может положительно сказаться на производительности.)

Когда обслуживающий процесс запрашивает информацию о статистике (через представления или функции), в его память читается последняя доступная версия статистики — это называется *снимком статистики*. Если не попросить явно, снимок не будет перечитываться до конца транзакции, чтобы обеспечить согласованность.

Таким образом, из-за задержек серверный процесс получает не самую свежую статистику — но обычно это и не требуется.

При останове сервера коллектор сбрасывает статистику в постоянные файлы в каталог PGDATA/pg_stat. Таким образом, статистика сохраняется при перезапуске сервера. Обнуление счетчиков происходит по команде администратора, а также при восстановлении сервера после сбоя.

Статистика внутри базы

```
=> 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 0.23 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.74 s (drop tables 0.00 s, create tables 0.03 s, client-side generate 0.32 s, vacuum 0.20 s, primary keys 0.18 s).
```

Сбросим все накопленные ранее статистики:

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

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

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

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

Теперь запускаем тест TPC-B на несколько секунд:

```
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: 1151
latency average = 8.696 ms
tps = 115.001408 (including connections establishing)
tps = 115.120387 (excluding connections establishing)
```

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

```
=> 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      | 2302
idx_tup_fetch | 2302
n_tup_ins     | 0
n_tup_upd     | 1151
n_tup_del     | 0
n_tup_hot_upd | 310
n_live_tup    | 0
n_dead_tup    | 1087
n_mod_since_analyze | 1151
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 страницы:

```

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

```

```

-[ RECORD 1 ]-----+-----
reloid        | 16546
schemaname    | public
relname       | pgbench_accounts
heap_blks_read | 14
heap_blks_hit  | 6857
idx_blks_read  | 272
idx_blks_hit   | 6017
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 ]-----+-----
reloid        | 16546
indexrelid    | 16560
schemaname    | public
relname       | pgbench_accounts
indexrelname   | pgbench_accounts_pkey
idx_scan      | 2302
idx_tup_read   | 3145
idx_tup_fetch  | 2302

```

```

=> 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  | 272
idx_blks_hit   | 6017

```

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 глобальную статистику по всей базе данных:


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

-[RECORD 1]-----+	
datid	16539
datname	admin_monitoring
numbackends	1
xact_commit	1167
xact_rollback	0
blks_read	378
blks_hit	20322
tup_returned	18433
tup_fetched	3182
tup_inserted	1151
tup_updated	3454
tup_deleted	0
conflicts	0
temp_files	0
temp_bytes	0
deadlocks	0
checksum_failures	
checksum_last_failure	
blk_read_time	22.641
blk_write_time	0
stats_reset	2022-05-11 13:16:20.488286+03

Здесь есть много полезной информации о количестве произошедших взаимоблокировок, зафиксированных и отмененных транзакций, использовании временных файлов, ошибках подсчета контрольных сумм.

В 14-й версии появилась и статистика пользовательских сеансов.

Отдельно доступна статистика по процессам фоновой записи и контрольной точки, ввиду ее важности для мониторинга экземпляра:

```
=> CHECKPOINT;
```

CHECKPOINT

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

-[RECORD 1]-----+	
checkpoints_timed	0
checkpoints_req	1
checkpoint_write_time	83
checkpoint_sync_time	288
buffers_checkpoint	1998
buffers_clean	0
maxwritten_clean	0
buffers_backend	1701
buffers_backend_fsync	0
buffers_alloc	380
stats_reset	2022-05-11 13:16:20.585929+03

- buffers_clean — количество страниц, записанных фоновой записью;
- buffers_checkpoint — количество страниц, записанных контрольной точкой;
- buffers_backend — количество страниц, записанных серверными процессами.

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 ничего не делает:

```
student$ /usr/lib/postgresql/13/bin/psql -d admin_monitoring
```

```
| => BEGIN;
```

```
| BEGIN
```

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

```
| UPDATE 1
```

А второй пытается изменить ту же строку и блокируется:

```
student$ /usr/lib/postgresql/13/bin/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          | 33531
query        | UPDATE t SET n = n + 1;
state        | idle in transaction
wait_event   | ClientRead
wait_event_type | Client
pg_blocking_pids | {}
-[ RECORD 2 ]-----+-----
pid          | 32771
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          | 33592
query        | UPDATE t SET n = n + 2;
state        | active
wait_event   | transactionid
wait_event_type | Lock
pg_blocking_pids | {33531}
```

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.

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
-----
      33592
(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`), другая — нет и ожидает.

```
=> SELECT locktype, transactionid, pid, mode, granted
FROM pg_locks
WHERE transactionid IN (
  SELECT transactionid FROM pg_locks WHERE pid = 33592 AND NOT granted
);
```

locktype	transactionid	pid	mode	granted
transactionid	1737	33531	ExclusiveLock	t
transactionid	1737	33592	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(33592)) AS b(pid);
```

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

Функция `unnest` нужна, поскольку `pg_blocking_pids` возвращает массив идентификаторов процессов, блокирующих искомый серверный 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	query	state	wait_event	wait_event_type
32771	SELECT pid, query, state, wait_event, wait_event_type+ FROM pg_stat_activity WHERE backend_type = 'client backend'	active		
-[RECORD 2]-----				
pid	query	state	wait_event	wait_event_type
33592	UPDATE t SET n = n + 2;	idle	ClientRead	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
27360	autovacuum launcher	2022-05-11 13:15:13.858182+03	
27362	logical replication launcher	2022-05-11 13:15:13.859902+03	
32771	client backend	2022-05-11 13:16:19.0773+03	active
33592	client backend	2022-05-11 13:16:34.692787+03	idle
27358	background writer	2022-05-11 13:15:13.854533+03	
27357	checkpointer	2022-05-11 13:15:13.853104+03	
27359	walwriter	2022-05-11 13:15:13.856069+03	

(7 rows)

Сравним с тем, что показывает операционная система:

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

27355

```
student$ sudo ps -o pid,command --ppid 27355
```

```

PID COMMAND
27357 postgres: 13/main: checkpointer
27358 postgres: 13/main: background writer
27359 postgres: 13/main: walwriter
27360 postgres: 13/main: autovacuum launcher
27361 postgres: 13/main: stats collector
27362 postgres: 13/main: logical replication launcher
32771 postgres: 13/main: student admin_monitoring [local] idle
33592 postgres: 13/main: student admin_monitoring [local] idle
```

Можно заметить, что в pg_stat_activity не попадает процесс stats collector.

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

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

Структуры представлений описаны в документации:

<https://postgrespro.ru/docs/postgresql/13/progress-reporting>

Создание резервных копий рассматривается в модуле «Резервное копирование».

В 14-й версии в этот список добавилось представление `pg_stat_progress_copy` для отслеживания работы команды `COPY`.

Расширения в поставке

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

Другие расширения

<code>pg_stat_plans</code>	статистика по планам запросов
<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_buffercache` позволяет заглянуть в содержимое буферного кеша и т. п.

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>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` и `csvlog` 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`. А в 15-й версии для приемника можно будет указать `jsonlog`.

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 дисковой подсистемы. 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/13/logfile-maintenance>

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

Параметр `log_filename` может 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/13/runtime-config-logging.html#RUNTIME-CONFIG-LOGGING-WHERE>

Альтернативно 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/dalibo/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-13-main.log | tail -n 10
```

```
2022-05-11 13:13:42.784 MSK [20066] student@student FATAL: terminating connection due to administrator command
2022-05-11 13:15:01.528 MSK [26737] student@student FATAL: terminating connection due to administrator command
2022-05-11 13:15:06.277 MSK [26951] student@student FATAL: terminating connection due to unexpected postmaster exit
2022-05-11 13:16:36.539 MSK [33531] 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(1,1000000);
```

```
sum
```

```
500519.56142244255
```

```
(1 row)
```

И посмотрим журнал:

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

```
2022-05-11 13:16:38.223 MSK [32771] student@admin_monitoring LOG: duration: 297.136 ms statement: SELECT sum(random()) FROM generate_series(1,1000000);
```

Универсальные системы мониторинга

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

Системы мониторинга PostgreSQL

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

На практике, если подходить к делу серьезно, 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. Если 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/postgrespro/13/pgpro-pwr>

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

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

PostgreSQL предоставляет собираемую 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          | 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
```

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 ]-----+-----
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    | 2022-05-11 13:21:31.032183+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$ /usr/lib/postgresql/13/bin/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$ /usr/lib/postgresql/13/bin/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.

```
| UPDATE 1
```

```
| ERROR: deadlock detected
```

```
| DETAIL: Process 50813 waits for ShareLock on transaction 1915; blocked by process 50895.
```

```
| Process 50895 waits for ShareLock on transaction 1914; blocked by process 50813.
```

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

```
| CONTEXT: while updating tuple (0,2) in relation "t"
```

Проверим информацию в журнале сообщений:

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

```
2022-05-11 13:21:36.150 MSK [50813] student@admin_monitoring ERROR: deadlock detected
```

```
2022-05-11 13:21:36.150 MSK [50813] student@admin_monitoring DETAIL: Process 50813 waits for ShareLock on transaction 1915; blocked by process 50895.
```

```
Process 50895 waits for ShareLock on transaction 1914; blocked by process 50813.
```

```
Process 50813: UPDATE t SET n = 20 WHERE n = 2;
```

```
Process 50895: UPDATE t SET n = 100 WHERE n = 1;
```

```
2022-05-11 13:21:36.150 MSK [50813] student@admin_monitoring HINT: See server log for query details.
```

```
2022-05-11 13:21:36.150 MSK [50813] student@admin_monitoring CONTEXT: while updating tuple (0,2) in relation "t"
```

```
2022-05-11 13:21:36.150 MSK [50813] student@admin_monitoring STATEMENT: UPDATE t SET n = 20 WHERE n = 2;
```

1. Установите расширение `pg_stat_statements`.
Выполните несколько произвольных запросов.
Посмотрите, какая информация записывается при этом
в представление `pg_stat_statements`.

1. Для установки расширения потребуется не только выполнить команду `CREATE EXTENSION`, но и изменить значение параметра `shared_preload_libraries` с последующей перезагрузкой сервера.
<https://postgrespro.ru/docs/postgresql/13/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 13 main restart
```

```
student$ /usr/lib/postgresql/13/bin/psql
```

Проверим значение параметра и создадим расширение. Поскольку отслеживаться будут запросы из любых баз данных кластера, расширение лучше установить в ту базу данных, которая существует всегда, например `postgres`.

```
=> \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

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 * 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 | 0.274911
min_exec_time | 0.020066999999999998
max_exec_time | 0.098548000000000001
mean_exec_time | 0.0549822
stddev_exec_time | 0.027787474382534306
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

```