Авторы: Алексей Береснев, Илья Баштанов, Павел Толмачев

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Расширение pgpro_stats

Агрегированная статистика

Дополнительные метрики

Статистика очистки

Статистика аннулирования кеша

Трассировка сеансов

Расширение pgpro_stats

Статистика запросов

Затраченные ресурсы, планы и события ожиданий

Основные настройки

pgpro_stats.max = 5000	предел количества запросов
pgpro_stats.track = top	верхний уровень или все (all)
pgpro_stats.query_sample_rate = 1.0	доля запросов
pgpro_stats.enable_profile = true	профиль ожиданий
pgpro_stats.profile_period = 10 ms	период семплирования

Представление pgpro_stats_statements

Расширение pgpro_stats собирает статистику по выполняемым операторам. По сравнению с pg_stat_statements добавлен сбор данных о планах запросов и событиях ожидания для каждого типа запроса. Расширения pgpro_stats и pg_stat_statements несовместимы.

Доступ к статистике через представление pgpro_stats_statements.

Максимальное число строк в представлении pgpro_stats_statements, определяет параметр *pgpro_stats.max*. Значение по умолчанию — 5000.

Параметр *pgpro_stats.track* определяет, будут ли отслеживаться лишь только непосредственно выполненные команды SQL (значение top — по умолчанию), или также и вложенные операторы (значение all).

Для снижения нагрузки можно уменьшить частоту выборки, заданную параметром *pgpro_stats.query_sample_rate*. Тогда для вычисления статистики будет случайно выбираться заданная доля запросов.

Семплинг событий ожидания выполняется с периодичностью раз в *pgpro_stats.profile_period* единиц времени (10 мс по умолчанию), если установлен параметр *pgpro_stats.enable_profile*.

Статистика ожиданий доступна в поле pgpro_stats_statements.wait_stats типа jsonb. Все показатели представлены в миллисекундах и кратны параметру конфигурации *pgpro_stats.profile_period*.

Также собирается информация о затраченных ресурсах с точки зрения операционной системы с помощью вызова rusage:

https://postgrespro.ru/docs/enterprise/13/pgpro-stats#PGPRO-STATS-RUSAGE

Агрегированная статистика

Статистика запросов, агрегированная по объектам

весь кластер, база данных, пользователь, адрес, приложение, процесс

Основные настройки

pgpro_stats.track_totals = on
pgpro_stats.track_cluster, *pgpro_stats.track_databases* и т. п.
pgpro_stats.max_totals = 1000 предел количества объектов

Представление pgpro_stats_totals

Модуль pgpro_stats позволяет агрегировать статистику запросов по различным объектам:

- cluster — весь кластер баз данных;
- database — отдельная база данных;
- user — пользователь;
- client_addr — адрес клиента;
- application — приложение;
- backend — обслуживающий процесс;
- session — сеанс.

Необходимость агрегации по конкретному типу объекта настраивается параметрами *pgpro_stats.track_cluster*, *pgpro_stats.track_databases* и т. п.

Агрегированная статистика доступна в представлении pgpro_stats_totals. Каждая строка в этом представлении содержит значения по конкретному объекту. Количество объектов, по которым отслеживается статистика, ограничено параметром *pgpro_stats.max_totals* (по умолчанию — 1000). Когда количество объектов превышает заданное, информация о наименее активно используемых объектах удаляется.

Сбор статистики по операторам и планам

```
student$ psql
```

Подключим разделяемую библиотеку pgpro_stats:

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

```
ALTER SYSTEM
```

```
student$ sudo systemctl restart postgrespro-ent-13.service
```

Создадим тестовую базу данных на основе demo:

```
student$ psql
```

```
=> CREATE DATABASE perfanalysis TEMPLATE demo;
```

```
CREATE DATABASE
```

Для удобства настроим путь поиска:

```
=> ALTER DATABASE perfanalysis SET search_path TO bookings, public;
```

```
ALTER DATABASE
```

В схеме bookings копия стандартной демобазы минимального размера.

```
=> \c perfanalysis
```

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

```
=> \dt
```

```
              List of relations
  Schema  |      Name       | Type  |  Owner
----------+-----------------+-------+---------
 bookings | aircrafts_data  | table | student
 bookings | airports_data   | table | student
 bookings | boarding_passes | table | student
 bookings | bookings        | table | student
 bookings | flights         | table | student
 bookings | seats           | table | student
 bookings | ticket_flights  | table | student
 bookings | tickets         | table | student
(8 rows)
```

Подключим расширение:

```
=> CREATE EXTENSION pgpro_stats;
```

```
CREATE EXTENSION
```

Выполним несколько запросов к таблице билетов tickets, предварительно добавив индекс по фамилиям пассажиров.

```
=> CREATE INDEX ticket_pname ON tickets(passenger_name);
```

```
CREATE INDEX
```

Пара запросов с параллельным последовательным сканированием таблицы.

```
=> SELECT count(book_ref) FROM tickets WHERE passenger_name < 'Z';
```

```
 count
--------
 363794
(1 row)
```

```
=> SELECT count(book_ref) FROM tickets WHERE passenger_name < 'Y';
```

```
 count
--------
 351813
(1 row)
```

Посмотрим, что накопилось в статистике операторов.

```
=> SELECT query, plan, calls
FROM pgpro_stats_statements
WHERE query LIKE 'SELECT count(book_ref) FROM tickets WHERE passenger_name%' \gx
-[ RECORD 1 ]-----------------------------------------------------------------------------------
query | SELECT count(book_ref) FROM tickets WHERE passenger_name < $1
plan  | Finalize Aggregate                                                                      +
      |   Output: count(book_ref)                                                               +
      |   ->  Gather                                                                            +
      |         Output: (PARTIAL count(book_ref))                                               +
      |         Workers Planned: 2                                                              +
      |         ->  Partial Aggregate                                                           +
      |               Output: PARTIAL count(book_ref)                                           +
      |               ->  Parallel Seq Scan on bookings.tickets                                 +
      |                     Output: ticket_no, book_ref, passenger_id, passenger_name, contact_data+
      |                     Filter: (tickets.passenger_name < $1)                               +
      |
calls | 2
```

Поле calls содержит информацию о количестве выполнений оператора.

Операторы DML объединяются в одну запись, если они имеют одинаковую структуру.

В следующих запросах используется индексный доступ:

```
=> SELECT count(book_ref) FROM tickets WHERE passenger_name > 'Z';

 count
-------
  2939
(1 row)


=> SELECT count(book_ref) FROM tickets WHERE passenger_name > 'Y';

 count
-------
 14920
(1 row)
```

Теперь у нас два разных плана.

```
=> SELECT query, plan, calls
FROM pgpro_stats_statements
WHERE query LIKE 'SELECT count(book_ref) FROM tickets WHERE passenger_name%' \gx
-[ RECORD 1 ]-----------------------------------------------------------------------------------
query | SELECT count(book_ref) FROM tickets WHERE passenger_name > $1
plan  | Aggregate                                                                               +
      |   Output: count(book_ref)                                                               +
      |   ->  Bitmap Heap Scan on bookings.tickets                                              +
      |         Output: ticket_no, book_ref, passenger_id, passenger_name, contact_data         +
      |         Recheck Cond: (tickets.passenger_name > $1)                                     +
      |         ->  Bitmap Index Scan on ticket_pname                                           +
      |               Index Cond: (tickets.passenger_name > $1)                                 +
      |
calls | 2
-[ RECORD 2 ]-----------------------------------------------------------------------------------
query | SELECT count(book_ref) FROM tickets WHERE passenger_name < $1
plan  | Finalize Aggregate                                                                      +
      |   Output: count(book_ref)                                                               +
      |   ->  Gather                                                                            +
      |         Output: (PARTIAL count(book_ref))                                               +
      |         Workers Planned: 2                                                              +
      |         ->  Partial Aggregate                                                           +
      |               Output: PARTIAL count(book_ref)                                           +
      |               ->  Parallel Seq Scan on bookings.tickets                                 +
      |                     Output: ticket_no, book_ref, passenger_id, passenger_name, contact_data+
      |                     Filter: (tickets.passenger_name < $1)                               +
      |
calls | 2
```

Для пользователя student получим агрегированную статистику по использованию некоторых системных ресурсов, а также по количеству выполненных запросов и событий ожидания.

```
=> SELECT object_type, object_name, queries_executed, total_exec_time,
  (total_exec_rusage).reads, (total_exec_rusage).writes,
  (total_exec_rusage).user_time, (total_exec_rusage).system_time,

  jsonb_pretty(wait_stats) AS wait_stats
FROM pgpro_stats_totals
WHERE object_type = 'user'
  AND object_id = 'student'::regrole \gx

-[ RECORD 1 ]----+-----------------------------------
object_type      | user
object_name      | student
queries_executed | 13
total_exec_time  | 1360.3785060000002
reads            | 176128
writes           | 302538752
user_time        | 0.18600600000000003
system_time      | 0.26420600000000005
wait_stats       | {                                +
                 |     "IO": {                      +
                 |         "BufFileRead": 10,       +
                 |         "CopyFileRead": 60,      +
                 |         "CopyFileWrite": 150,    +
                 |         "DataFileImmediateSync": 10+
                 |     },                           +
                 |     "IPC": {                     +
                 |         "CheckpointDone": 30,    +
                 |         "BgWorkerShutdown": 10   +
                 |     },                           +
                 |     "Total": {                   +
                 |         "IO": 230,               +
                 |         "IPC": 40,               +
                 |         "Total": 270             +
                 |     }                            +
                 | }
```

В выводе этого запроса получены количество выполненных запросов, общее время выполнения запросов, а также несколько полей записи в столбце total_exec_rusage:

- reads — количество байтов, прочитанное из файловой системы;
- writes — количество байтов, записанное в файловую систему;
- user_time — время, потраченное на работу запроса в непривилегированном режиме (user space);
- system_time — время, потраченное на работу запроса на стороне ядра ОС (kernel space).

Столбец total_exec_rusage составного типа pgpro_stats_rusage содержит статистику использования ресурсов на этапах планирования и выполнения запросов.

Собранная статистика сбрасывается функцией pgpro_stats_statements_reset для заданного пользователя (userid), базы данных (dbid), запроса (queryid) или плана (planid). Если не указать ни один аргумент, будет очищена вся статистика:

```
=> SELECT pgpro_stats_statements_reset();

 pgpro_stats_statements_reset
------------------------------

(1 row)
```

Статистика по запросам удалена:

```
=> SELECT query, plan, calls
FROM pgpro_stats_statements
WHERE query LIKE 'SELECT count(*) FROM tickets WHERE passenger_name%' \gx

(0 rows)
```

Агрегированная статистика для заданного типа и идентификатора объекта очищается функцией pgpro_stats_totals_reset.

Например, сбросим агрегатную статистику по роли student:

```
=> SELECT pgpro_stats_totals_reset('user', 'student'::regrole::bigint);

 pgpro_stats_totals_reset
--------------------------

(1 row)
```

```
=> SELECT object_type, object_name, queries_executed, total_exec_time,
  (total_exec_rusage).reads, (total_exec_rusage).writes,
  (total_exec_rusage).user_time, (total_exec_rusage).system_time,
  jsonb_pretty(wait_stats) AS wait_stats
FROM pgpro_stats_totals
WHERE object_type = 'user'
  AND object_id = 'student'::regrole \gx

-[ RECORD 1 ]----+----------------------
object_type      | user
object_name      | student
queries_executed | 1
total_exec_time  | 0.0028049999999999998
reads            | 0
writes           | 0
user_time        | 2.805e-06
system_time      | 0
wait_stats       |
```

После сброса статистики был только что выполнен единственный запрос. Поэтому queries_executed равно единице.

Статистика по базе данных сохранилась, так как она не была стерта:

```
=> SELECT object_type, object_name, queries_executed, total_exec_time,
  (total_exec_rusage).reads, (total_exec_rusage).writes,
  (total_exec_rusage).user_time, (total_exec_rusage).system_time,
  jsonb_pretty(wait_stats) AS wait_stats
FROM pgpro_stats_totals
WHERE object_type = 'database'
  AND object_id = (SELECT oid FROM pg_database WHERE datname = 'perfanalysis') \gx

-[ RECORD 1 ]----+-----------------------------------
object_type      | database
object_name      | perfanalysis
queries_executed | 15
total_exec_time  | 540.741133
reads            | 229376
writes           | 7299072
user_time        | 0.18600880500000003
system_time      | 0.021925
wait_stats       | {                                 +
                 |     "IO": {                       +
                 |         "BufFileRead": 10,        +
                 |         "DataFileImmediateSync": 10+
                 |     },                            +
                 |     "IPC": {                      +
                 |         "BgWorkerShutdown": 10    +
                 |     },                            +
                 |     "Total": {                    +
                 |         "IO": 20,                 +
                 |         "IPC": 10,                +
                 |         "Total": 30               +
                 |     }                             +
                 | }
```

Дополнительные метрики

Произвольные метрики на основе SQL-запросов

собираются фоновыми рабочими процессами
кольцевой буфер

Основные настройки

pgpro_stats.metrics_buffer_size = 16kB
pgpro_stats.metrics_workers = 2

*pgpro_stats.metric*_N_*name*	имя метрики
*pgpro_stats.metric*_N_*query*	SQL-запрос
*pgpro_stats.metric*_N_*period* = 1 min	интервал сбора
*pgpro_stats.metric*_N_*db*	список баз данных
*pgpro_stats.metric*_N_*user*	пользователь

Представление pg_stats_metrics

Используя pgpro_stats, можно определить дополнительные метрики. Собираемые статистические данные будут накапливаться в кольцевом буфере для последующей передачи их в систему мониторинга.

Это лучше, чем опрос базы данных системой мониторинга, так как в случае прерывания соединения собранные данные не будут потеряны, а сохранятся в буфере и могут быть получены позднее с помощью представления pgpro_stats_metrics.

Для определения дополнительной метрики в конфигурацию СУБД необходимо поместить соответствующие параметры. Номер N в имени параметра служит уникальным идентификатором метрики, к которой будет применяться задаваемое значение; он должен быть положительным и уникальным среди всех метрик. Параметры:

- *pgpro_stats.metric*_N_*name* — имя метрики;
- *pgpro_stats.metric*_N_*query* — запрос для получения метрики;
- *pgpro_stats.metric*_N_*period* — периодичность измерений;
- *pgpro_stats.metric*_N_*db* — список баз данных;
- *pgpro_stats.metric*_N_*user* — пользователь, от имени которого будет собираться метрика. Пользователь должен иметь доступ к базе данных, в которой будет выполняться запрос для сбора метрики.

Обсуждение того, какие именно метрики имеет смысл собирать, выходит за рамки курса, но порекомендуем книгу Алексея Лесовского «Мониторинг PostgreSQL»: https://postgrespro.ru/education/books/monitoring

Сбор дополнительных метрик

Предположим, что требуется раз в 30 секунд собирать количество выполнений контрольной точки по требованию.

Запишем параметры в файл conf.d/metrics.conf в каталоге данных кластера и перезапустим СУБД.

```
student$ cat << EOF | sudo -u postgres tee /var/lib/pgpro/ent-13/conf.d/metrics.conf
pgpro_stats.metric_1_name = 'bgwriter'
pgpro_stats.metric_1_query = 'SELECT * FROM pg_stat_bgwriter'
pgpro_stats.metric_1_db = 'perfanalysis'
pgpro_stats.metric_1_user = 'student'
pgpro_stats.metric_1_period = '30s'
EOF
```

```
pgpro_stats.metric_1_name = 'bgwriter'
pgpro_stats.metric_1_query = 'SELECT * FROM pg_stat_bgwriter'
pgpro_stats.metric_1_db = 'perfanalysis'
pgpro_stats.metric_1_user = 'student'
pgpro_stats.metric_1_period = '30s'
```

Добавленная метрика — первая, поэтому N=1.

```
student$ sudo systemctl restart postgrespro-ent-13.service
```

```
student$ psql -p 5432 -d perfanalysis
```

Заглянем в представление для дополнительных метрик:

```
=> SELECT metric_number, db_name, ts, value #> '{0,checkpoints_req}' AS checkpoints_req
FROM pgpro_stats_metrics;
```

```
 metric_number |   db_name    |              ts               | checkpoints_req
---------------+--------------+-------------------------------+-----------------
             1 | perfanalysis | 2024-01-16 21:33:45.959535+03 | 6
(1 row)
```

Столбец value содержит результат запроса, вычисляющего метрику, в виде массива объектов jsonb. Здесь извлечен первый по порядку (нулевой) элемент, в котором по ключу checkpoints_req получено искомое значение.

Во всех купленных авиабилетах изменим фамилии пассажиров так, чтобы первая буква была заглавной, а остальные — строчными:

```
=> UPDATE tickets SET passenger_name = initcap(passenger_name);
```

```
UPDATE 366733
```

Выполним контрольную точку и немного подождем.

```
=> CHECKPOINT;
```

```
CHECKPOINT
```

В представлении pgpro_stats_metrics появится одна или несколько записей. Обратите внимание на значение checkpoints_req:

```
=> SELECT metric_number, db_name, ts, value #> '{0,checkpoints_req}' AS checkpoints_req
FROM pgpro_stats_metrics;
```

```
 metric_number |   db_name    |              ts               | checkpoints_req
---------------+--------------+-------------------------------+-----------------
             1 | perfanalysis | 2024-01-16 21:33:45.959535+03 | 6
             1 | perfanalysis | 2024-01-16 21:34:15.991991+03 | 7
(2 rows)
```

Удалим настройки дополнительных метрик:

```
postgres$ rm /var/lib/pgpro/ent-13/conf.d/metrics.conf
```

Снова перезагрузим сервер.

```
student$ sudo systemctl restart postgrespro-ent-13.service
```

Статистика очистки

Собирается ядром Postgres Pro Enterprise

Представления

- pgpro_stats_vacuum_database — очистка баз данных
- pgpro_stats_vacuum_tables — очистка таблиц текущей базы
- pgpro_stats_vacuum_indexes — очистка индексов текущей базы

Представления pgpro_stats_vacuum_database, pgpro_stats_vacuum_tables и pgpro_stats_vacuum_indexes содержат статистику очистки, соответственно, баз данных, таблиц в текущей базе данных и индексов в ней. Все эти три представления основаны на одноименных функциях.

Эти данные собираются ядром Postgres Pro Enterprise как расширение стандартного сборщика статистики.

Отслеживается подробная информация на уровне страниц (в количественном и временно́м представлении) и журнальных записей.

Статистика очистки

```
student$ psql -p 5432 -d perfanalysis
```

Выключим автоочистку для таблицы tickets:

```
=> ALTER TABLE tickets SET (autovacuum_enabled = off);
```

```
ALTER TABLE
```

Поскольку статистика по очистке собирается ядром СУБД, для ее сброса используется стандартная функция:

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

```
 pg_stat_reset
---------------

(1 row)
```

Статистика очистки для таблицы tickets отсутствует:

```
=> SELECT * FROM pgpro_stats_vacuum_tables(
  (SELECT oid FROM pg_database WHERE datname = 'perfanalysis'),
  'tickets'::regclass
) \gx
```

```
(0 rows)
```

Вернем фамилии к написанию в верхнем регистре и выполним очистку.

```
=> UPDATE tickets SET passenger_name = upper(passenger_name);
```

```
UPDATE 366733
```

```
=> VACUUM tickets;
```

```
VACUUM
```

Немного ожидания...

Снова проверим статистику очистки для таблицы tickets. Теперь статистика очистки собрана:

```
=> SELECT * FROM pgpro_stats_vacuum_tables(
  (SELECT oid FROM pg_database WHERE datname = 'perfanalysis'),
  'tickets'::regclass
) \gx
```

```
-[ RECORD 1 ]---------+--------
relid                 | 16444
total_blks_read       | 5
total_blks_hit        | 32225
total_blks_dirtied    | 6
total_blks_written    | 0
rel_blks_read         | 1
rel_blks_hit          | 31403
pages_scanned         | 12321
pages_removed         | 0
pages_frozen          | 5547
pages_all_visible     | 12321
tuples_deleted        | 366733
tuples_frozen         | 0
dead_tuples           | 0
index_vacuum_count    | 1
rev_all_frozen_pages  | 5556
rev_all_visible_pages | 12309
wal_records           | 29451
wal_fpi               | 6
wal_bytes             | 4281406
blk_read_time         | 0
blk_write_time        | 0
delay_time            | 0
system_time           | 4.106
user_time             | 110.42
total_time            | 153.071
interrupts            | 0
```

Вернем автоочистку для таблицы tickets:

```
=> ALTER TABLE tickets SET (autovacuum_enabled = on);

ALTER TABLE

=> \q
```

Аннулирование кешей

Сообщения об аннулировании метаданных измененных объектов в локальных кешах процессов

Представления

pgpro_stats_inval_status — состояние очереди сообщений

pgpro_stats_statements
pgpro_stats_totals
} счетчики сообщений для операторов

Обслуживающие процессы кешируют метаданные объектов в локальных кешах. При изменении процессом метаданных объекта нужно обновить информацию в кешах других обслуживающих процессов. Для этого другим процессам отправляется сообщение аннулирования (Cache Invalidation), которое помещается в специальную очередь.

Обслуживающие процессы получают сообщения аннулирования из этой очереди. Если объект, для которого получено сообщение аннулирования, находится в кеше, его метаданные в кеше обновляются.

Когда обслуживающий процесс, добавляющий сообщения в очередь, сталкивается с пределом размера очереди, он удаляет сообщения, уже обработанные всеми обслуживающими процессами. Если какие-то обслуживающие процессы не успевают обрабатывать сообщения и задерживают очистку, им передается сигнал для сброса кеша.

В pgpro_stats значения большинства счетчиков сообщений аннулирования увеличиваются при генерации сообщений, а сообщения попадают в очередь после фиксации транзакции. При откате счетчики не уменьшаются, хотя сообщения при этом не попадают в очередь.

Представление pgpro_stats_inval_status содержит одну строку с текущим состоянием очереди аннулирования кеша. Представления pgpro_stats_statements и pgpro_stats_totals для каждого объекта показывают запись со счетчиками сообщений аннулирования кеша.

https://postgrespro.ru/docs/enterprise/13/pgpro-stats#PGPRO-STATS-CACHE-INVALIDATION-METRICS

Аннулирование кешей

Откроем новый сеанс и получим PID обслуживающего процесса:

```
student$ psql -d perfanalysis
```

```
=> SELECT pg_backend_pid();
```

```
 pg_backend_pid
----------------
         265192
(1 row)
```

Пока ни одного сообщения об аннулировании кеша нет:

```
=> SELECT object_type, (inval_msgs).total
FROM pgpro_stats_totals
WHERE object_type = 'backend'
  AND object_id = 265192;
```

```
 object_type | total
-------------+-------
 backend     |     0
(1 row)
```

Изменим, например, параметр хранения таблицы.

```
=> ALTER TABLE seats SET (autovacuum_enabled = on);
```

```
ALTER TABLE
```

Сообщения об аннулировании кеша, сгенерированные текущим обслуживающим процессом:

```
=> SELECT object_type,
  (inval_msgs).total,
  (inval_msgs).catcache,
  (inval_msgs).relcache
FROM pgpro_stats_totals
    WHERE object_type = 'backend'
      AND object_id = 265192;
```

```
 object_type | total | catcache | relcache
-------------+-------+----------+----------
 backend     |     3 |        2 |        1
(1 row)
```

Здесь:

- total — общее число сообщений аннулирования;
- catcache — число сообщений избирательного аннулирования кеша каталога;
- relcache — число сообщений избирательного аннулирования кеша отношений.

```
=> \q
```

Трассировка сеансов

Трассировка сеансов в журнал сообщений или файл

Трассировка выполняется только для команд, удовлетворяющих критериям фильтрации

pgpro_stats_trace_insert — создает фильтр
pgpro_stats_trace_update — изменяет фильтр
pgpro_stats_trace_delete — удаляет фильтр
pgpro_stats_trace_reset — удаляет все фильтры
pgpro_stats_trace_show — показывает фильтры

Расширение pgpro_stats позволяет выполнять трассировку сеансов, записывая данные о выполняемых командах в журнал сообщений или в файлы. Регистрируются команды, которые удовлетворяют фильтрам. Фильтры размещаются в таблице в общей памяти. Строки таблицы описывают фильтры, а столбцы содержат условия фильтрации.

Для управления фильтрами расширение pgpro_stats предоставляет функции, показанные на слайде.

Изменения фильтров сразу влияют на трассировку во всех сеансах экземпляра СУБД.

Условия фильтров включают:

- идентифицирующие поля — проверяются на равенство;
- ресурсные поля — проверяются на превышение порога;
- поля, определяющие параметры EXPLAIN.

Например, database_name — идентифицирующее поле: сеанс, в котором выполнена команда, должен быть в указанной базе данных. Если же в фильтре указано ресурсное поле duration, то команда должна выполняться более указанной длительности.

Подробности об условиях фильтров в документации: https://postgrespro.ru/docs/enterprise/13/pgpro-stats#PGPRO-STATS-TRACE-SHOW-OUTPUT

Трассировка сеансов

```
student$ psql -d perfanalysis
```

Откроем второй сеанс и получим PID обслуживающего процесса.

```
student$ psql -d perfanalysis
```

```
=> SELECT pg_backend_pid();
```

```
 pg_backend_pid
----------------
         265424
(1 row)
```

Добавим фильтр pa1, который будет записывать в файл трассировки pa1.trace вывод EXPLAIN ANALYZE для каждой команды, выполняющейся во втором сеансе:

```
=> SELECT pgpro_stats_trace_insert(
  'alias', 'pa1',
  'database_name', current_database(),
  'pid', 265424,
  'explain_analyze', true,
  'tracefile', 'pa1'
);
```

```
 pgpro_stats_trace_insert
--------------------------
                        1
(1 row)
```

Проверим получившийся фильтр трассировки:

```
=> SELECT * from pgpro_stats_trace_show() \gx
```

```
-[ RECORD 1 ]-------+-------------
filter_id           | 1
active              | t
alias               | pa1
tracefile           | pa1
pid                 | 265424
database_name       | perfanalysis
client_addr         |
application_name    |
username            |
queryid             |
planid              |
duration            |
plan_time           |
exec_time           |
user_time           |
system_time         |
rows                |
shared_blks_hit     |
shared_blks_read    |
shared_blks_fetched |
shared_blks_dirtied |
shared_blks_written |
local_blks_hit      |
local_blks_read     |
local_blks_fetched  |
local_blks_dirtied  |
local_blks_written  |
temp_blks_read      |
temp_blks_written   |
wal_bytes           |
total_wait_time     |
total_inval_msgs    |
explain_analyze     | t
explain_verbose     | f
explain_costs       | t
explain_settings    | f
explain_buffers     | f
explain_wal         | f
explain_timing      | t
explain_format      | text
```

Во втором сеансе выполним запрос.

```
=> SELECT count(*) FROM tickets WHERE passenger_name > 'Z';
```

```
 count
-------
  2939
(1 row)
```

Закроем второй сеанс и проверим файл трассировки.

```
=> \q
```

```
postgres$ cat /var/lib/pgpro/ent-13/pg_stat/pa1.trace

Query Text: SELECT count(*) FROM tickets WHERE passenger_name > 'Z';
Aggregate  (cost=12.76..12.77 rows=1 width=8) (actual time=0.367..0.367 rows=1 loops=1)
  ->  Index Only Scan using ticket_pname on tickets  (cost=0.42..11.72 rows=417 width=0) (actual time=0.019..0.267 rows=2939 loops=1)
        Index Cond: (passenger_name > 'Z'::text)
        Heap Fetches: 0
```

Удалим фильтр трассировки.

```
=> SELECT pgpro_stats_trace_delete(1);

 pgpro_stats_trace_delete
--------------------------
 t
(1 row)
```

Итоги

Расширение pgpro_stats позволяет выявлять и анализировать ресурсоемкие операции

Собирается статистика по затраченным ресурсам, планам и событиям ожиданий

Отдельно собирается агрегированная статистика

Можно использовать метрики, определенные пользователем

Имеется статистика по очистке и аннулированию кешей

Можно трассировать сеансы

Практика

1. Подготовьте СУБД для работы с расширением pgpro_stats.
2. Выполните запрос с двухпроходным хеш-соединением в условиях недостаточной памяти *work_mem*.
3. Получите отчет pgpro_stats.
4. Выполните этот же запрос в условиях достаточной памяти *work_mem* и снова получите отчет.

1. Подключите разделяемые библиотеки pgpro_stats, создайте базу данных — копию базы данных demo и подключите к ней расширение.

2. Запустите транзакцию с установленными локальными параметрами *work_mem* = '12MB' и *hash_mem_multiplier* = 1. В транзакции получите реальный план запроса

```
SELECT *
FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref
```

3. Получите отчет pgpro_stats с помощью представления pgpro_stats_statements, из которого необходимо вывести поля: query, plan, calls, temp_blks_read, temp_blks_written.

4. Очистите статистику и второй раз запустите транзакцию с настройками *work_mem* = '48MB' и *hash_mem_multiplier* = 3 и снова получите такой же отчет. Обратите внимание во втором отчете на отсутствие операций по работе с временными файлами.

1. Подготовка СУБД для работы с расширением pgpro_stats

```
student$ psql
```

Подключим разделяемые библиотеки.

Расширение pgpro_pwr получает сводную статистику ожиданий от расширения pg_wait_sampling:

```
=> ALTER SYSTEM SET shared_preload_libraries = pgpro_stats;
```

```
ALTER SYSTEM
```

Перезагрузим сервер.

```
student$ sudo systemctl restart postgrespro-ent-13.service
```

---

2. Запуск транзакции с недостаточной памятью work_mem

Создадим тестовую базу данных на основе demo:

```
student$ psql
```

```
=> CREATE DATABASE perfanalysis TEMPLATE demo;
```

```
CREATE DATABASE
```

Для удобства настроим путь поиска:

```
=> ALTER DATABASE perfanalysis SET search_path TO bookings, public;
```

```
ALTER DATABASE
```

Подключимся к базе данных.

```
=> \c perfanalysis
```

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

Подключим расширение:

```
=> CREATE EXTENSION pgpro_stats;
```

```
CREATE EXTENSION
```

Запустим транзакцию.

```
=> BEGIN;
```

```
BEGIN
```

```
=> SET LOCAL work_mem = '12MB';
```

```
SET
```

```
=> SET LOCAL hash_mem_multiplier = 1;
```

```
SET
```

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Hash Join (actual rows=366733 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   Buffers: shared hit=3 read=7844, temp read=3491 written=3491
   ->  Seq Scan on tickets t (actual rows=366733 loops=1)
         Buffers: shared read=6159
   ->  Hash (actual rows=262788 loops=1)
         Buckets: 262144  Batches: 2  Memory Usage: 9071kB
         Buffers: shared read=1685, temp written=690
         ->  Seq Scan on bookings b (actual rows=262788 loops=1)
               Buffers: shared read=1685
 Planning:
   Buffers: shared hit=140 read=29 dirtied=1
(12 rows)
```

```
=> COMMIT;
```

```
COMMIT
```

3. Отчет по нагрузке

Соединение хешированием при недостатке work_memory выполняется в двухпроходном режиме и использует временные файлы.

Проверим, что накопилось в статистике.

```
=> SELECT query, plan, calls, temp_blks_read, temp_blks_written
FROM pgpro_stats_statements
WHERE query LIKE 'EXPLAIN (analyze, buffers,%' \gx
```

```
-[ RECORD 1 ]-----+-------------------------------------------------------------
query             | EXPLAIN (analyze, buffers, costs off, timing off, summary off)+
                  | SELECT * FROM bookings b                                      +
                  |   JOIN tickets t ON b.book_ref = t.book_ref
plan              |
calls             | 1
temp_blks_read    | 3491
temp_blks_written | 3491
```

По столбцам temp_blks_read и temp_blks_written можно судить об активности использования временных файлов.

4. Отчет по нагрузке в улучшенных условиях

Сбросим статистику.

```
=> SELECT pgpro_stats_statements_reset();
```

```
 pgpro_stats_statements_reset
------------------------------

(1 row)
```

Снова запустим транзакцию с увеличенными значениями параметров.

Теперь соединение хешированием будет однопроходным и не будет использовать временные файлы.

```
=> BEGIN;
```

```
BEGIN
```

```
=> SET LOCAL work_mem = '48MB';
```

```
SET
```

```
=> SET LOCAL hash_mem_multiplier = 3;
```

```
SET
```

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM bookings b
  JOIN tickets t ON b.book_ref = t.book_ref;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Hash Join (actual rows=366733 loops=1)
   Hash Cond: (t.book_ref = b.book_ref)
   Buffers: shared hit=7844
   ->  Seq Scan on tickets t (actual rows=366733 loops=1)
         Buffers: shared hit=6159
   ->  Hash (actual rows=262788 loops=1)
         Buckets: 524288  Batches: 1  Memory Usage: 18190kB
         Buffers: shared hit=1685
         ->  Seq Scan on bookings b (actual rows=262788 loops=1)
               Buffers: shared hit=1685
 Planning:
   Buffers: shared hit=8
(12 rows)
```

```
=> COMMIT;
```

```
COMMIT
```

Проверим, что накопилось в статистике.

```
=> SELECT query, plan, calls, temp_blks_read, temp_blks_written
FROM pgpro_stats_statements
WHERE query LIKE 'EXPLAIN (analyze, buffers,%' \gx
```

```
-[ RECORD 1 ]-----+-------------------------------------------------------------
query             | EXPLAIN (analyze, buffers, costs off, timing off, summary off)+
                  | SELECT * FROM bookings b                                      +
                  |   JOIN tickets t ON b.book_ref = t.book_ref
plan              |
calls             | 1
temp_blks_read    | 0
temp_blks_written | 0
```

Временные файлы не были использованы.