Авторы: Алексей Береснев, Илья Баштанов, Павел Толмачев

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

Темы

Назначение

Архитектура

Планируемые задания

Управление и наблюдение

Разовые задания

Назначение

Расширение pgpro_scheduler

- планировщик периодических (планируемых) и разовых заданий
- предоставляет возможности мониторинга
- обеспечивает разграничение прав

Задание — команды SQL

- каждая команда в отдельной транзакции или все в одной

Планировщик заданий требуется для периодического запуска заданий по расписанию, а также для запуска разовых отложенных заданий. В Postgres Pro планировщик реализован в виде расширения pgpro_scheduler.

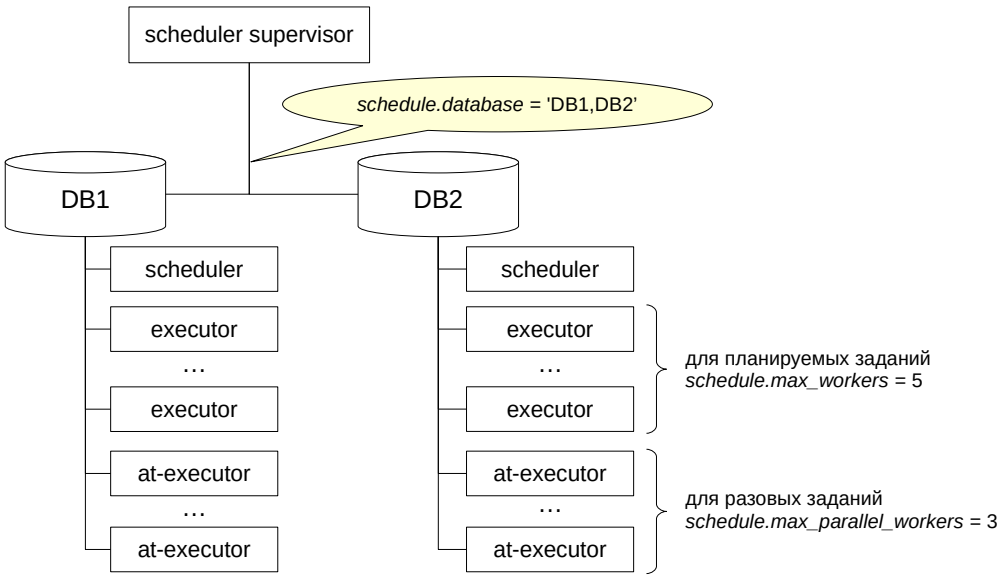
С его помощью можно планировать регулярные задачи администрирования без доступа к файловой системе сервера и необходимости использования внешнего программного обеспечения.

Задание планировщика состоит из команд SQL, которые могут быть выполнены как в одной транзакции, так и в разных.

Планировщик заданий pgpro_scheduler позволяет управлять как периодическими (планируемыми), так и разовыми заданиями, а также наблюдать за ходом их выполнения. Планировщик заданий pgpro_scheduler обеспечивает разделение полномочий: каждое задание принадлежит определенному пользователю и выполняется от его имени.

https://postgrespro.ru/docs/enterprise/13/pgpro-scheduler

Архитектура



Планировщик задач использует механизм фоновых рабочих процессов.

Основной процесс-диспетчер планировщика (один на весь кластер баз данных) запускается функцией schedule.enable, а для автоматического его запуска при старте сервера нужно установить параметр *schedule.auto_enabled* = true.

Диспетчерский процесс проверяет настройки и для каждой из указанных в параметре *schedule.database* баз данных запускает отдельный процесс-менеджер. Также сразу запускаются обработчики разовых заданий планировщика, их количество ограничено значением *schedule.max_parallel_workers*.

Максимальное количество обработчиков для планируемых заданий в каждой базе ограничено параметром *schedule.max_workers*.

Оба ограничения можно задавать на уровне базы данных командой ALTER DATABASE. Обработчики планируемых и разовых заданий работают независимо.

При этом общее количество рабочих процессов сервера ограничено параметром *max_worker_processes*, его значение должно учитывать потребности других задач, использующих фоновые процессы.

Если нет свободных процессов для выполнения, задание будет ожидать освобождения обработчика. По умолчанию время ожидания не ограничено, поэтому, планируя задание, разумно указывать ограничение времени, при превышении которого pgpro_scheduler отменит выполнение.

Установка расширения pgpro_scheduler

Создадим базу данных:

```
=> CREATE DATABASE scheduler;
```

```
CREATE DATABASE
```

Для работы расширения необходимо подключить библиотеку:

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

```
ALTER SYSTEM
```

Общее ограничение на количество фоновых рабочих процессов в системе, значения по умолчанию достаточно:

```
=> SHOW max_worker_processes;
```

```
 max_worker_processes
----------------------
 8
(1 row)
```

Подключение библиотеки требует перезагрузки СУБД.

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

```
student$ psql
```

Установим расширение pgpro_scheduler в базе данных, в которой оно будет использоваться.

```
=> \c scheduler
```

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

```
=> CREATE EXTENSION pgpro_scheduler;
```

```
CREATE EXTENSION
```

Объекты расширения находятся в схеме schedule. Например, представления для мониторинга разовых заданий:

```
=> \dv schedule.*
```

```
             List of relations
  Schema  |      Name      | Type |  Owner
----------+----------------+------+---------
 schedule | all_job_status | view | student
 schedule | all_jobs_log   | view | student
 schedule | job_status     | view | student
 schedule | jobs_log       | view | student
(4 rows)
```

Настройка

Укажем имена баз данных, для которых будут запускаться задания:

```
=> ALTER SYSTEM SET schedule.database = 'scheduler';
```

```
ALTER SYSTEM
```

Применим изменение конфигурации.

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

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

Установим ограничения на уровне базы данных.

Максимальное число рабочих процессов для периодических заданий:

```
=> ALTER DATABASE demo SET schedule.max_workers = 3;
```

```
ALTER DATABASE
```

Ограничение на число рабочих процессов, выполняющих разовые задания:

```
=> ALTER DATABASE demo SET schedule.max_parallel_workers = 3;

ALTER DATABASE
```

Включим планировщик заданий:

```
=> SELECT schedule.enable();

 enable
--------
 t
(1 row)
```

Состояние процессов планировщика заданий:

```
=> SELECT * FROM schedule.status();

  pid   | database  |       type
--------+-----------+------------------
 272034 | postgres  | supervisor
 272035 | scheduler | database manager
 272036 | scheduler | at job executor
 272037 | scheduler | at job executor
(4 rows)
```

Информация об этих процессах доступна и в pg_stat_activity:

```
=> SELECT pid, datname, application_name, backend_type, query
FROM pg_stat_activity
WHERE application_name ~ '^pgp-s';

  pid   |  datname  |     application_name      |          backend_type           |           query
--------+-----------+---------------------------+---------------------------------+---------------------------
 272034 | postgres  | pgp-s supervisor          | pgpro scheduler                 | working on: scheduler
 272035 | scheduler | pgp-s manager [scheduler] | scheduler scheduler             | vanish expired tasks done
 272036 | scheduler | pgp-s at executor         | scheduler at-executor scheduler | waiting for a job
 272037 | scheduler | pgp-s at executor         | scheduler at-executor scheduler | waiting for a job
(4 rows)
```

Планирование заданий

Команды SQL

- выполняются в одной или в нескольких транзакциях

Расписание

- в формате cron
- массив дат
- объект JSON
- выражение SQL — время следующего запуска
- диапазон времени

Дополнительные условия

- пользователь, длительность, время ожидания, обработчик ошибок

Функция schedule.create_job позволяет запланировать периодическое задание. Параметры планирования задаются в виде одного объекта jsonb.

Атрибут *commands* задает одну или несколько команд для выполнения. Если задать команды в виде символьной строки, они выполнятся в одной транзакции. Если же задан массив строк, то каждому элементу массива будет соответствовать отдельная транзакция.

Расписание для задания задается атрибутами:

- *cron* — в формате crontab;
- *dates* — значение или массив значений типа timestamptz;
- *rule* — объектом jsonb.

Атрибутами *start_date* и *end_date* можно ограничить время, в которое задание может выполняться; вне диапазона работающее задание будет остановлено, а новое не запустится.

Время следующего запуска задания будет определяться динамически, если в атрибуте *next_time_statement* задать SQL-выражение. Оно вычисляется после каждого выполнения задания.

Также можно запланировать выполнение задания с правами другого пользователя (атрибут *run_as*), ограничить длительность выполнения и ожидания, подключить обработчик ошибок в виде SQL-команды.

https://postgrespro.ru/docs/enterprise/13/pgpro-scheduler#PGPRO-SCHEDULER-USAGE

Управление и мониторинг

Управление

- изменение свойств
- активация
- деактивация
- удаление

Мониторинг

- свойства задания
- список заданий (всех, выполняющихся, выполненных)
- список запланированных выполнений
- очистка журнала

Расширение pgpro_scheduler использует функции в схеме schedule для управления и мониторинга периодических заданий:

- deactivate_job — деактивировать задание;
- activate_job — активировать задание;
- drop_job — удалить задание;
- get_job — информация о задании;
- get_owned_cron — задания пользователя;
- get_cron — список заданий, выполняемых пользователем;
- get_active_jobs — список выполняющихся заданий;
- get_log — список всех завершенных заданий;
- get_user_log — список завершенных заданий пользователя;
- clean_log — удалить все записи о завершенных заданиях.

Имеется функция timetable, которая возвращает таблицу с описанием всех запланированных выполнений заданий (многократно повторяемых и разовых), попадающих в заданный диапазон времени.

Планирование заданий

Создадим таблицу:

```
=> CREATE TABLE t (
  bknd int DEFAULT pg_backend_pid(),
  txid bigint DEFAULT txid_current(),
  ttime timestamptz DEFAULT now()
);
```

```
CREATE TABLE
```

А теперь создадим периодическое задание, используя формат crontab. Формат использует пять полей для идентификации времени:

- минуты часа (0-59);
- часы (0-23);
- дни месяца (1-31);
- месяц (1-12 или трехбуквенное сокращение);
- дни недели (0 — воскресенье ... 6 — суббота, или трехбуквенные сокращения).

Если в поле после звездочки через косую черту указано число — это шаг выполнения. Например,

*/10 * * * *

означает «раз в десять минут».

Расширенный формат cron использует шесть полей с секундами. Например,

*/30 * * * * *

означает «раз в тридцать секунд».

Запланируем задание на выполнение раз в минуту:

```
=> SELECT schedule.create_job(
  '{ "commands": "INSERT INTO t DEFAULT VALUES",
     "cron": "*/1 * * * *" }'
);
```

```
 create_job
------------
          1
(1 row)
```

Получим информацию о запланированном задании:

```
=> SELECT id, commands, run_as, active
FROM schedule.get_cron();
```

```
 id |             commands             | run_as  | active
----+----------------------------------+---------+--------
  1 | {"INSERT INTO t DEFAULT VALUES"} | student | t
(1 row)
```

Функция get_job информирует о задании, возвращая значение типа cron_rec.

Проверим поле rule, описывающее расписание, преобразовав расписание из представления JSONB в текстовые строки и урезав их по ширине:

```
=> SELECT substring(jsonb_each_text(rule)::text FOR 90) AS rule
FROM schedule.get_job(1);
```

```
                                          rule
-------------------------------------------------------------------------------------------
 (days,"[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
 (hours,"[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
 (wdays,"[0, 1, 2, 3, 4, 5, 6]")
 (months,"[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]")
 (crontab,"*/1 * * * *")
 (minutes,"[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 2
 (seconds,[0])
(7 rows)
```

Через минуту сработает первое задание. Подождем...

Проверим, что было добавлено в таблицу t:

```
=> SELECT * FROM t;
```

```
  bknd  | txid |             ttime
--------+------+-------------------------------
 273444 |  746 | 2024-01-16 21:36:00.019585+03
(1 row)
```

Функция timetable информирует о выполненных и запланированных заданиях. Получим список заданий за 5 минут:

```
=> SELECT id, type, commands, scheduled_at, status
FROM schedule.timetable(
  now() - interval '3 minutes',
  now() + interval '2 minutes'
)
ORDER BY scheduled_at;
```

```
 id |    type    |             commands             |      scheduled_at      |  status
----+------------+----------------------------------+------------------------+-----------
  1 | periodical | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:36:00+03 | done
  1 | periodical | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:37:00+03 | submitted
  1 | periodical | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:38:00+03 | submitted
(3 rows)
```

Отчет о выполненных заданиях планировщика:

```
=> SELECT cron, scheduled_at, started, status, message
FROM schedule.get_log();
```

```
 cron |      scheduled_at      |            started            | status | message
------+------------------------+-------------------------------+--------+---------
    1 | 2024-01-16 21:36:00+03 | 2024-01-16 21:36:00.009891+03 | done   |
(1 row)
```

Управление запланированными заданиями

До сих пор задание выполнялось в ноль секунд каждой минуты:

```
=> SELECT jsonb_array_elements(rule->'seconds') FROM schedule.get_job(1);
```

```
 jsonb_array_elements
----------------------
 0
(1 row)
```

Пусть задание выполняется чаще — раз в десять секунд.

Используем расширенный формат cron из шести полей, где первое поле — секунды:

```
=> SELECT schedule.set_job_attribute(1, 'cron','*/10 * * * * *');
```

```
 set_job_attribute
-------------------
 t
(1 row)
```

Проверим в rule: задание должно быть запланировано для исполнения с периодичностью раз в десять секунд.

```
=> SELECT jsonb_array_elements(rule->'seconds') FROM schedule.get_job(1);
```

```
 jsonb_array_elements
----------------------
 0
 10
 20
 30
 40
 50
(6 rows)
```

Подождем немного...

Проверим, что было добавлено в таблицу t:

```
=> SELECT * FROM t;
```

```
  bknd  | txid |             ttime
--------+------+-------------------------------
 273444 |  746 | 2024-01-16 21:36:00.019585+03
 273701 |  750 | 2024-01-16 21:36:10.00786+03
(2 rows)
```

Временно приостановим задание. Новые записи не будут попадать в таблицу.

```
=> SELECT schedule.deactivate_job(1);

 deactivate_job
----------------
 t
(1 row)
```

Проверим состояние задания — оно должно быть не активно:

```
=> SELECT id, commands, run_as, active FROM schedule.get_job(1);

 id |             commands             | run_as  | active
----+----------------------------------+---------+--------
  1 | {"INSERT INTO t DEFAULT VALUES"} | student | f
(1 row)
```

Подождем немного...

Снова запустим задание и проверим состояние:

```
=> SELECT schedule.activate_job(1);

 activate_job
--------------
 t
(1 row)


=> SELECT id, commands, run_as, active FROM schedule.get_job(1);

 id |             commands             | run_as  | active
----+----------------------------------+---------+--------
  1 | {"INSERT INTO t DEFAULT VALUES"} | student | t
(1 row)
```

Подождем еще немного...

Проверим записи в таблице t:

```
=> SELECT * FROM t;

  bknd  | txid |             ttime
--------+------+-------------------------------
 273444 |  746 | 2024-01-16 21:36:00.019585+03
 273701 |  750 | 2024-01-16 21:36:10.00786+03
 273883 |  755 | 2024-01-16 21:36:40.014583+03
 273885 |  758 | 2024-01-16 21:36:50.016327+03
(4 rows)
```

Выполненные задания отображаются в журнале:

```
=> SELECT cron, scheduled_at, started, status, message
FROM schedule.get_log() WHERE cron = 1;

 cron |      scheduled_at      |            started            | status | message
------+------------------------+-------------------------------+--------+---------
    1 | 2024-01-16 21:36:00+03 | 2024-01-16 21:36:00.009891+03 | done   |
    1 | 2024-01-16 21:36:10+03 | 2024-01-16 21:36:10.000386+03 | done   |
    1 | 2024-01-16 21:36:40+03 | 2024-01-16 21:36:40.00741+03  | done   |
    1 | 2024-01-16 21:36:50+03 | 2024-01-16 21:36:50.009257+03 | done   |
(4 rows)
```

Снова приостановим задание и очистим таблицу.

```
=> SELECT schedule.deactivate_job(1);

 deactivate_job
----------------
 t
(1 row)
```

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

```
DELETE 4
```

Команды в отдельных транзакциях

В атрибуте commands значения могут задаваться в виде текста или массива.

Если отдельные команды SQL заданы текстовой строкой через точку с запятой, то все команды задания будут выполнены в одной транзакции.

Если команды SQL заданы JSON-массивом, то каждая из них будет выполнена в отдельной транзакции.

Создадим планируемое задание для заполнения этой таблицы раз в десять секунд.

Каждая SQL-команда выполняется в отдельной транзакции. Значения now() для каждой транзакции — свои, как и идентификаторы транзакции.

```
=> SELECT schedule.create_job(
  '{ "commands": [
       "INSERT INTO t DEFAULT VALUES",
       "SELECT pg_sleep(5)",
       "INSERT INTO t DEFAULT VALUES"
     ],
     "cron": "*/10 * * * * *" }'
);
```

```
 create_job
------------
          2
(1 row)
```

Снова немного подождем...

```
=> SELECT schedule.drop_job(2);
```

```
 drop_job
----------
 t
(1 row)
```

Проверим, что было добавлено в таблицу:

```
=> SELECT * FROM t;
```

```
  bknd  | txid |             ttime
--------+------+-------------------------------
 274415 |  766 | 2024-01-16 21:37:00.019046+03
 274415 |  768 | 2024-01-16 21:37:05.033023+03
(2 rows)
```

Команды были выполнены одним обслуживающим процессом, но в отдельных транзакциях.

Команды в одной транзакции

Даже если команды заданы массивом, можно задать атрибуту use_same_transaction значение true, и все команды будут выполнены в одной транзакции.

Удалим все записи в таблице.

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

```
DELETE 2
```

В одной транзакции значения времени, которые возвращает функция now(), будут одинаковыми.

```
=> SELECT schedule.create_job(
  '{ "commands": [
       "INSERT INTO t DEFAULT VALUES",
       "SELECT pg_sleep(5)",
       "INSERT INTO t DEFAULT VALUES"
     ],
     "cron": "*/10 * * * * *",
     "use_same_transaction": true}'
);
```

```
 create_job
------------
          3
(1 row)
```

Подождем чуть-чуть...

Удалим задание и проверим, что было добавлено в таблицу.

```
=> SELECT schedule.drop_job(3);

 drop_job
----------
 t
(1 row)


=> SELECT * FROM t;

  bknd  | txid |             ttime
--------+------+-------------------------------
 274973 |  774 | 2024-01-16 21:37:10.010166+03
 274973 |  774 | 2024-01-16 21:37:10.010166+03
(2 rows)
```

Все три оператора задания выполнились в одной транзакции.

Снова очистим таблицу и запустим первое задание.

```
=> DELETE FROM t;

DELETE 2

=> SELECT schedule.activate_job(1);

 activate_job
--------------
 t
(1 row)
```

Разовые задания

Выполнить задание

- немедленно
- в определенный момент
- после завершения других заданий

Назначить повторное выполнение

Отменить еще не выполняющееся задание

Работают независимо от запланированных

Функция submit_job создает разовое задание. По умолчанию задание будет выполнено немедленно. Можно задать время запуска параметром *run_after*. Также можно в параметре *depends_on* передать массив идентификаторов других заданий — в этом случае данное задание будет выполнено сразу после их завершения.

Функция resubmit, вызванная в SQL-запросе задания, позволяет задать время следующего запуска этого задания.

Функция cancel_job отменяет задание. Если задание уже выполняется, оно не будет прервано, но повторно назначить его будет нельзя.

Получить список выполненных и запланированных заданий (как периодических, так и разовых) позволяет функция timetable.

Запланированные и разовые задания выполняются в разных процессах, поэтому они могут работать параллельно.

Разовые задания

Запланируем разовое задание, удаляющее через десять секунд таблицу t:

```
=> SELECT schedule.submit_job(
  'DROP TABLE t',
  run_after := now() + interval '10 seconds'
);
```

```
 submit_job
------------
          1
(1 row)
```

Таблица t пока на месте:

```
=> SELECT * FROM t;
```

```
 bknd | txid | ttime
------+------+-------
(0 rows)
```

Подождем немного...

```
=> SELECT * FROM t;
```

```
ERROR:  relation "t" does not exist
LINE 1: SELECT * FROM t;
                      ^
```

Таблица уже удалена.

В таблице заданий выводится информация и о разовых заданиях:

```
=> SELECT id, type, commands, scheduled_at, status
FROM schedule.timetable(
  now() - interval '1 minutes',
  now()
)
WHERE id = 1
ORDER BY scheduled_at;
```

```
 id |    type    |             commands             |         scheduled_at         | status
----+------------+----------------------------------+------------------------------+--------
  1 | periodical | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:36:40+03       | done
  1 | periodical | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:36:50+03       | done
  1 | periodical | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:37:20+03       | done
  1 | onetime    | {"DROP TABLE t"}                 | 2024-01-16 21:37:25.55903+03 | done
(4 rows)
```

Свойство onrollback периодического задания задает действия при сбое команды задания — будем восстанавливать таблицу, предполагая ее отсутствие:

```
=> SELECT schedule.set_job_attribute(1, 'onrollback',
  'CREATE TABLE t (
     bknd int DEFAULT pg_backend_pid(),
     txid bigint DEFAULT txid_current(),
     ttime timestamptz DEFAULT now()
   )'
);
```

```
 set_job_attribute
-------------------
 t
(1 row)
```

Проверим состояние задания:

```
=> SELECT id, commands, run_as, active
FROM schedule.get_job(1);
```

```
 id |             commands             | run_as  | active
----+----------------------------------+---------+--------
  1 | {"INSERT INTO t DEFAULT VALUES"} | student | t
(1 row)
```

Подождем еще немного...

Теперь таблица восстановлена и пока пуста:

```
=> SELECT * FROM t;
```

```
 bknd | txid | ttime
------+------+-------
(0 rows)
```

Подождем еще...

Теперь таблица уже не пуста:

```
=> SELECT * FROM t;
```

```
  bknd  | txid |             ttime
--------+------+-------------------------------
 276609 |  797 | 2024-01-16 21:37:40.013737+03
(1 row)
```

Посмотрим записи отчета:

```
=> SELECT cron, scheduled_at, commands, started, status, message
FROM schedule.get_log()
WHERE cron = 1;
```

```
 cron |      scheduled_at      |             commands             |            started            | status |                      message
------+------------------------+----------------------------------+-------------------------------+--------+-----------------------------------------------
    1 | 2024-01-16 21:36:00+03 | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:36:00.009891+03 | done   |
    1 | 2024-01-16 21:36:10+03 | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:36:10.000386+03 | done   |
    1 | 2024-01-16 21:36:40+03 | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:36:40.00741+03  | done   |
    1 | 2024-01-16 21:36:50+03 | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:36:50.009257+03 | done   |
    1 | 2024-01-16 21:37:20+03 | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:37:20.003055+03 | done   |
    1 | 2024-01-16 21:37:30+03 | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:37:30.002356+03 | error  | error in command #1: relation "t" does not exist
    1 | 2024-01-16 21:37:40+03 | {"INSERT INTO t DEFAULT VALUES"} | 2024-01-16 21:37:40.006808+03 | done   |
(7 rows)
```

Удалим запланированное задание:

```
=> SELECT schedule.drop_job(1);

 drop_job
----------
 t
(1 row)
```

Итоги

Расширение pgpro_scheduler позволяет планировать задания

Задания выполняются либо периодически по расписанию, либо один раз в запланированное время

Задания принадлежат пользователям и выполняются от их имени

Каждое задание выполняется в отдельном процессе

Имеются средства управления и мониторинга заданий

Практика

1. Подготовьте систему к работе с расширениями pgpro_scheduler и pg_stat_statements.
2. Установите расширения pgpro_scheduler и pg_stat_statements в базу данных и настройте их.
3. Запланируйте задание, собирающее статистику по наиболее часто выполняемым запросам с периодичностью в несколько секунд, а также разовое задание, копирующее набранную статистику в файл.

1. Для установки расширений потребуется изменить значение параметра shared_preload_libraries с последующей перезагрузкой сервера.

2. Подключив расширения с помощью CREATE EXTENSION, выполните настройку числа рабочих процессов по аналогии с демонстрацией, после чего перечитайте конфигурацию и запустите планировщик.

3. Запустите задание на выполнение с периодичностью, например, раз в двадцать секунд. Задание должно собирать статистику по наиболее часто выполняющимся командам SQL с помощью расширения pg_stat_statements. Для этого создайте таблицу с такой же структурой, как у представления pg_stat_statements, но с дополнительным полем, в которое будет записываться момент сбора статистики. Ограничьте время работы периодического задания, например, одной минутой, используйте для этого атрибут *end_date* в аргументе функции schedule.create_job. После этого определите разовое задание, которое скопирует собранную статистику в файл.

1. Подготовка СУБД для работы с расширениями pgpro_scheduler и pg_stat_statements

```
student$ psql
```

Создадим базу данных scheduler:

```
=> CREATE DATABASE scheduler;
```

```
CREATE DATABASE
```

Подключим разделяемые библиотеки:

```
=> ALTER SYSTEM SET shared_preload_libraries = pg_stat_statements, pgpro_scheduler;
```

```
ALTER SYSTEM
```

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

```
ALTER SYSTEM
```

Общее ограничение на количество параллельных процессов:

```
=> SHOW max_worker_processes;
```

```
 max_worker_processes
----------------------
 8
(1 row)
```

Перезапустим СУБД.

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

```
student$ psql scheduler
```

2. Подключение расширений. Запуск планировщика

Установим расширения:

```
=> CREATE EXTENSION pgpro_scheduler;
```

```
CREATE EXTENSION
```

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

```
CREATE EXTENSION
```

База данных для нагрузочного тестирования:

```
=> ALTER SYSTEM SET schedule.database = 'scheduler';
```

```
ALTER SYSTEM
```

Применим изменения конфигурации:

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

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

Включим планировщик заданий:

```
=> SELECT schedule.enable();
```

```
 enable
--------
 t
(1 row)
```

Состояние процессов планировщика заданий:

```
=> SELECT * FROM schedule.status();
```

```
  pid   | database  |       type
--------+-----------+------------------
 305721 | postgres  | supervisor
 305722 | scheduler | database manager
 305723 | scheduler | at job executor
 305724 | scheduler | at job executor
(4 rows)
```

3. Планирование задач

Создадим таблицу для сохранения выборок статистики:

```
=> CREATE TABLE stat_tab (LIKE pg_stat_statements);
```

```
CREATE TABLE
```

```
=> ALTER TABLE stat_tab
  ADD COLUMN time timestamp with time zone DEFAULT now();
```

```
ALTER TABLE
```

Запланируем копирование статистики в таблицу stat_tab раз в 20 секунд, причем периодическое задание должно работать одну минуту:

```
=> SELECT schedule.create_job(
  ('{
    "commands": "INSERT INTO stat_tab SELECT * FROM pg_stat_statements ORDER BY calls DESC LIMIT 3;",
    "cron": "*/20 * * * * *",
    "end_date": "'||(now()+'1 minute'::interval)||'"
  }')::jsonb
);
```

```
 create_job
------------
          1
(1 row)
```

Через минуту содержимое таблицы нужно скопировать в файл с помощью разового задания.

```
student$ sudo rm -f /tmp/rep.stat
```

```
=> SELECT schedule.submit_job(
  'COPY stat_tab TO ''/tmp/rep.stat''',
  run_after => now() + interval '1 minutes'
);
```

```
 submit_job
------------
          1
(1 row)
```

Подождем минуту...

Проверим выполненные задания:

```
=> SELECT cron, scheduled_at, started, status, message
FROM schedule.get_log();
```

```
 cron |      scheduled_at      |            started            | status | message
------+------------------------+-------------------------------+--------+---------
    1 | 2024-01-16 21:46:40+03 | 2024-01-16 21:46:40.000507+03 | done   |
    1 | 2024-01-16 21:47:00+03 | 2024-01-16 21:47:00.019065+03 | done   |
    1 | 2024-01-16 21:47:20+03 | 2024-01-16 21:47:20.019847+03 | done   |
(3 rows)
```

В таблице собраны сведения о наиболее частых запросах:

```
=> SELECT time, queryid, calls FROM stat_tab;
```

```
             time              |       queryid       | calls
-------------------------------+---------------------+-------
 2024-01-16 21:46:40.008272+03 | 1980254974847749176 |     8
 2024-01-16 21:46:40.008272+03 | 3147765651446690248 |     2
 2024-01-16 21:46:40.008272+03 | 6439533008511123628 |     1
 2024-01-16 21:47:00.033803+03 | 1980254974847749176 |    48
 2024-01-16 21:47:00.033803+03 | 3147765651446690248 |     2
 2024-01-16 21:47:00.033803+03 | 8225083834985279947 |     2
 2024-01-16 21:47:20.027399+03 | 1980254974847749176 |    88
 2024-01-16 21:47:20.027399+03 | 3147765651446690248 |     3
 2024-01-16 21:47:20.027399+03 | 8225083834985279947 |     3
(9 rows)
```

Разовое задание скопировало таблицу в файл:

```
student$ sudo ls -l /tmp/rep.stat

-rw-r--r-- 1 postgres postgres 4419 янв 16 21:47 /tmp/rep.stat
```