

# Расширяемость Фоновые процессы



## Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

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

## Использование материалов курса

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

## Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

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

## Отказ от ответственности

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

Механизм фоновых процессов

Применение в ядре PostgreSQL

Возможности для прикладных задач

Расширение dblink

Расширение pg\_background

## Фиксированный набор служебных процессов

- postmaster
- walwriter
- checkpointer
- autovacuum
- и другие

## Динамически порождаемые фоновые процессы

- контролируются процессом postmaster
- имеют доступ к разделяемой памяти
- могут устанавливать внутреннее соединение с базами данных
- реализуются на языке C

PostgreSQL состоит из набора взаимодействующих процессов. За основными процессами закреплены собственные имена и они имеют особый смысл для сервера. Например, при остановке экземпляра нужно записать в журнал сообщение о завершении контрольной точки, поэтому процесс walwriter надо останавливать после checkpointer, и т. п. Обычно (хотя и не всегда) такие процессы работают постоянно от старта сервера до его останова.

Но вместе с тем часто возникает необходимость временно запустить процесс, который выполнит необходимую работу и завершится. Такую возможность предоставляет механизм **фоновых процессов** (background workers): <https://postgrespro.ru/docs/postgresql/16/bgworker>

Как и все остальные, фоновые процессы порождаются и контролируются процессом postmaster. Фоновые процессы имеют все возможности, такие как доступ к разделяемой памяти сервера и соединение с базами данных по внутреннему протоколу SPI (рассматривается в теме «Языки программирования»).

Фоновые процессы должны быть написаны на языке C, пример можно найти в исходном коде (src/test/modules/worker\_spi).

Заметим, что процесс автоочистки (autovacuum launcher) тоже динамически порождает рабочие процессы (autovacuum worker), но исторически делает это по-своему, не используя общий механизм фоновых процессов.

## Параллельное выполнение запросов

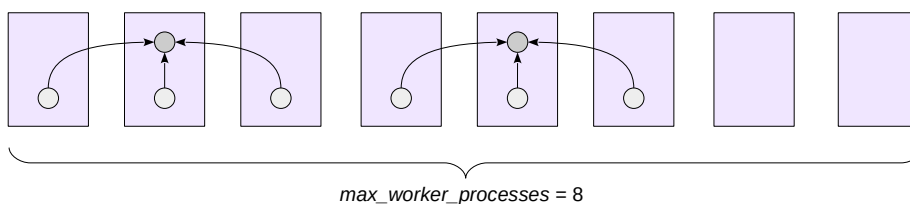
`max_parallel_workers = 8`

## Параллельное выполнение служебных команд

`max_parallel_maintenance_workers = 2`

## Логическая репликация

`max_logical_replication_workers = 4`



Фоновые процессы используются в PostgreSQL для различных целей.

### 1. Параллельное выполнение запросов.

### 2. Параллельное выполнение служебных команд (CREATE INDEX и REINDEX (в т. ч. CONCURRENTLY) для B-деревьев, VACUUM).

В этих случаях имеется ведущий процесс, инициирующий запуск рабочих процессов и получающий от них информацию.

### 3. Логическая репликация (для процессов на стороне сервера-подписчика).

Общее количество фоновых процессов в системе ограничено значением параметра `max_worker_processes`, и отдельными параметрами ограничено количество процессов для каждой из трех категорий. Если в пуле недостаточно фоновых процессов, то обработка будет выполняться последовательно, а не параллельно (а логическая репликация и вовсе не будет работать).

## Использование фоновых процессов

Вот простой наглядный пример использования фоновых процессов для распараллеливания запросов. Если требуется посчитать количество строк в большой таблице, ведущий процесс запускает несколько рабочих процессов (в данном случае 2):

```
=> EXPLAIN (analyze, costs off, timing off)
SELECT count(*) FROM mail_messages;
```

### QUERY PLAN

```
-----
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Seq Scan on mail_messages (actual rows=118708 loops=3)
Planning Time: 321.965 ms
Execution Time: 394.497 ms
(8 rows)
```

Запланированное количество и число реально запущенных процессов могут отличаться, если на момент запуска запроса пул фоновых процессов будет исчерпан.

В конце частичные результаты собираются и агрегируются лидирующим процессом чтобы получить итоговое значение.

## Некоторые идеи

- планировщик заданий внутри СУБД
- асинхронная обработка событий
- распараллеливание сложной обработки данных
- автономные транзакции
- и другое

Но пользовательские задачи неудобно писать на языке С

Кроме штатного использования, фоновые процессы можно приспособить и для решения пользовательских задач.

Например, можно организовать планировщик заданий внутри СУБД. В отличие от, например, cron, такой планировщик не будет зависеть от используемой операционной системы. (Планировщик внутри СУБД реализован в Postgres Pro Enterprise и рассматривается в курсе PGPRO).

Можно выполнять асинхронную обработку событий (что будет подробно рассмотрено в одноименной теме).

Можно распараллеливать сложную длинную обработку больших объемов данных.

Можно реализовать автономные транзакции, хотя и весьма дорогим способом. (В Postgres Pro Enterprise имеются автономные транзакции, они рассматриваются в курсе PGPRO).

И так далее.

Однако писать прикладные задачи на языке С крайне неудобно и требует неоправданно высокой квалификации.

## Назначение

выполнение произвольных SQL-команд на удаленном сервере  
в качестве удаленного сервера может выступать и локальный

## Плюсы

стандартное расширение

## Минусы

фоновые процессы не используются,  
устанавливается новое соединение

Одно из возможных решений — использовать расширение dblink, которое предназначено для выполнения SQL-запросов на удаленном сервере PostgreSQL (в качестве удаленного может выступать и локальный сервер).

Это проверенное временем стандартное расширение. Однако оно не использует механизм фоновых процессов — устанавливается полноценное новое соединение с сервером, что существенно дороже.

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

## Расширение dblink

Это стандартное расширение. Установим его:

```
=> CREATE DATABASE ext_bgworkers;
```

```
CREATE DATABASE
```

```
=> \c ext_bgworkers
```

You are now connected to database "ext\_bgworkers" as user "student".

```
=> CREATE EXTENSION dblink;
```

```
CREATE EXTENSION
```

Ниже — самый простой способ выполнить одиночный запрос на удаленном сервере. Первый параметр функции — строка соединения, второй — команда, которую надо выполнить.

Функция возвращает множество строк типа record, поэтому структуру составного типа необходимо указывать явно при ее вызове.

```
=> SELECT * FROM dblink(
    'host=localhost port=5432 dbname=postgres user=postgres password=postgres',
    $$ SELECT * FROM generate_series(1,3); $$
) AS (result integer);

result
-----
      1
      2
      3
(3 rows)
```

---

Команду, не возвращающую строки, можно выполнить с помощью другой функции:

```
=> SELECT * FROM dblink_exec(
    'host=localhost port=5432 dbname=postgres user=postgres password=postgres',
    $$ VACUUM; $$
);

dblink_exec
-----
VACUUM
(1 row)
```

---

Обе функции открывают соединение, выполняют команду и тут же закрывают соединение. Но есть возможность явно управлять соединением. Откроем его, указав имя:

```
=> SELECT * FROM dblink_connect(
    'remote',
    'host=localhost port=5432 dbname=postgres user=postgres password=postgres'
);

dblink_connect
-----
OK
(1 row)
```

Можно открыть и несколько соединений. Текущие открытые соединения показывает функция:

```
=> SELECT * FROM dblink_get_connections();

dblink_get_connections
-----
{remote}
(1 row)
```

Теперь можно выполнять команды, используя открытое соединение. В том числе можно вручную управлять транзакциями:

```
=> SELECT * FROM dblink_exec(
    'remote',
    $$ BEGIN; $$
);
```



```
dblink_exec
```

```
-----
```

```
BEGIN
```

```
(1 row)
```

```
=> SELECT * FROM dblink(  
    'remote',  
    $$ SELECT pg_backend_pid(); $$  
) AS (pid integer);
```

```
pid
```

```
-----
```

```
93607
```

```
(1 row)
```

```
=> SELECT * FROM dblink_exec(  
    'remote',  
    $$ COMMIT; $$  
);
```

```
dblink_exec
```

```
-----
```

```
COMMIT
```

```
(1 row)
```

---

Важная возможность — асинхронные вызовы. Следующая функция отправит запрос на сервер и тут же вернет управление:

```
=> SELECT * FROM dblink_send_query(  
    'remote',  
    $$ SELECT 'done' FROM pg_sleep(10); $$  
);
```

```
dblink_send_query
```

```
-----
```

```
1
```

```
(1 row)
```

- 1 — успешно;
- 0 — ошибка.

Далее мы можем проверить, выполняется ли еще запрос:

```
=> SELECT CASE dblink_is_busy('remote')  
    WHEN 1 THEN 'еще выполняется'  
    ELSE 'уже выполнен'  
END;
```

```
case
```

```
-----
```

```
еще выполняется
```

```
(1 row)
```

Результат получаем так (если запрос еще не выполнен, функция сама дожидается результатов):

```
=> SELECT * FROM dblink_get_result(  
    'remote'  
) AS (result text);
```

```
result
```

```
-----
```

```
done
```

```
(1 row)
```

---

Не забываем закрыть соединение:

```
=> SELECT * FROM dblink_disconnect('remote');
```

```
dblink_disconnect
```

```
-----
```

```
OK
```

```
(1 row)
```



# Расширение pg\_background

## Назначение

возможность выполнить команду SQL в фоновом процессе  
команда может вызвать подпрограмму на любом серверном языке

## Плюсы

используются фоновые процессы

## Минусы

стороннее расширение

9

Есть и другое решение, которое позволяет прикладным разработчикам воспользоваться фоновыми процессами — стороннее расширение pg\_background:

[https://github.com/vibhorkum/pg\\_background](https://github.com/vibhorkum/pg_background)

Фактически это небольшая обертка над низкоуровневым API фоновых процессов, позволяющая запускать в фоновом режиме произвольные команды SQL. Разумеется, таким образом можно вызывать хранимые функции и процедуры, написанные на любых серверных языках программирования (например, PL/pgSQL).

## Расширение pg\_background

Расширение уже собрано и доступно для установки в виртуальной машине курса:

```
=> CREATE EXTENSION pg_background;
```

```
CREATE EXTENSION
```

Расширение предоставляет всего три функции.

Функция pg\_background\_launch запускает фоновый процесс, выполняющий одну SQL-команду.

Например, выполним в фоне простой запрос. Для удобства он будет работать 10 секунд:

```
=> SELECT pg_background_launch(
    $$ SELECT 2+2 FROM (SELECT pg_sleep(10)) $$
);
```

```
pg_background_launch
-----
                94006
(1 row)
```

Пока запрос выполняется, мы можем увидеть процесс в pg\_stat\_activity:

```
=> SELECT query, backend_type, wait_event_type, wait_event
FROM pg_stat_activity WHERE pid = 94006 \gx
```

```
-[ RECORD 1 ]---+-----
query          | SELECT 2+2 FROM (SELECT pg_sleep(10))
backend_type    | pg_background
wait_event_type | Timeout
wait_event      | PgSleep
```

Обратите внимание на ожидание.

Функция pg\_background\_result выводит результат выполнения фоновой команды (при необходимости дожидаясь ее окончания).

Функция возвращает значения типа record, поэтому для вывода необходимо конкретизировать названия и типы полей составного типа.

```
=> SELECT * FROM pg_background_result(94006) AS (result integer);
```

```
result
-----
      4
(1 row)
```

Функция pg\_background\_detach отключает текущий процесс от ожидания результатов фонового процесса.

Передача результатов выполняется через очередь сообщений в разделяемой памяти сервера. Поэтому при переполнении очереди фоновый процесс будет ждать, пока мы не прочитаем накопившиеся сообщения, даже если они нас не интересуют.

Запустим процесс, возвращающий много информации:

```
=> SELECT pg_background_launch(
    $$ SELECT * FROM generate_series(1,1_000_000) $$
);
```

```
pg_background_launch
-----
                94158
(1 row)
```

```
=> SELECT query, backend_type, wait_event_type, wait_event
FROM pg_stat_activity WHERE pid = 94158 \gx
```

```
-[ RECORD 1 ]---+-----
query          | SELECT * FROM generate_series(1,1_000_000)
backend_type    | pg_background
wait_event_type | IPC
wait_event      | MessageQueuePutMessage
```

Обратите внимание на ожидание.

Отключимся от процесса:

```
=> SELECT * FROM pg_background_detach(94158);
```

```
pg_background_detach
```

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

```
=> SELECT query, backend_type, wait_event_type, wait_event  
FROM pg_stat_activity WHERE pid = 94158 \gx
```

```
-[ RECORD 1 ]---+-----  
query          | SELECT * FROM generate_series(1,1_000_000)  
backend_type    | pg_background  
wait_event_type |  
wait_event      |
```

Больше фоновый процесс ничего не ждет (и, возможно, уже отработал).

Заметим, что в случае dblink сложностей с переполнением буфера не возникает, потому что используется не межпроцессное взаимодействие, а устанавливается обычное соединение по клиент-серверному протоколу.

-----  
Интересно, что изначально в состав расширения pg\_background планировалось включить четвертую функция pg\_background\_run(pid, query), которая должна была передавать новое задание уже запущенному процессу, избегая затрат на создание очередного фонового процесса. Однако пока что это не реализовано.

Сервер предоставляет механизм фоновых процессов  
Фоновые процессы используются для внутренних целей,  
но могут быть использованы и для прикладных задач  
Расширение `pg_background` позволяет реализовывать  
фоновые процессы на процедурных языках



1. Реализуйте расчет рейтинга книг на основе данных о голосах пользователей «за» и «против». Чтобы не допускать разрастание таблицы, выполняйте обновление пакетами по  $N$  строк. Между частичными обновлениями запускайте очистку. Оформите обновление в виде хранимой процедуры, принимающей  $N$  как параметр.
2. Выполните процедуру обновления в фоновом режиме.

1. Рейтинг книги оценивается числом от 0 до 1.

«Наивная» формула для определения рейтинга может быть такой:

$\text{votes\_up} / (\text{votes\_up} + \text{votes\_down})$ , если  $\text{votes\_up} + \text{votes\_down} > 0$ ;  
0, иначе.

Недостаток этой формулы показывает пример: книга, набравшая 100 голосов «за» и 1 голос «против», получит меньший рейтинг, чем книга, набравшая всего один голос «за».

Можно подумать над более справедливой формулой, а можно воспользоваться уже готовой функцией `public.rating`.

Пакетное обновление рассматривалось в практике к теме «Очистка». Здесь применим тот же подход. Однако подумайте над тем, как пересчитывать рейтинг только тех книг, для которых это необходимо.

Команда `VACUUM` не может быть вызвана внутри транзакции, но ее можно выполнить как отдельную SQL-команду в фоновом процессе.

## 1. Процедура расчета рейтинга

Для вычисления рейтинга мы будем пользоваться готовой функцией `public.rating`. Эта функция вычисляет рейтинг с учетом баланса между долей положительных оценок и неопределенностью малого числа наблюдений (нижняя граница доверительного интервала Вильсона для параметра Бернулли).

Один из возможных способов не пересчитывать рейтинг без надобности состоит в том, чтобы добавить к таблице книг столбец логического типа, указывающий на необходимость пересчета. Процедура расчета рейтинга будет сбрасывать этот флаг, а любое изменение количества голосов — устанавливать.

Мы реализуем другой способ. Добавим к таблице столбец с датой последнего пересчета рейтинга. Процедура пересчета будет выполняться только для тех книг, «возраст» рейтинга которых больше определенного порога. Это позволит при необходимости (например, если изменится формула) пересчитать рейтинг всех книг, не обновляя предварительно все строки таблицы.

```
=> ALTER TABLE books ADD COLUMN rating_at timestamp DEFAULT NULL;
```

ALTER TABLE

Обратите внимание: добавление столбца затрагивает только системный каталог и не приводит к перезаписи всех строк таблицы.

Для обработки строк пакетами нужно определиться:

- с размером пакета;
- с тем, как из всех строк выбирать для обработки только нужное количество.

Важный момент: обработка пакета может — в принципе — завершиться ошибкой. Важно, чтобы это не повлияло на возможность продолжить обработку и довести ее до конца.

Для выбора пакета строк мы используем подход, показанный ранее в практике к теме «Очистка». Будем выбирать и блокировать первые подходящие N строк, игнорируя уже заблокированные:

```
=> \set N 5

=> \set AGE '1 day'

=> WITH batch AS (
    SELECT * FROM books
    WHERE rating IS NULL OR age(rating_at) > interval :AGE'
    LIMIT :N
    FOR UPDATE SKIP LOCKED
)
SELECT book_id FROM batch;

 book_id
-----
      56
      70
      28
      22
      68
(5 rows)
```

Здесь мы выбираем строки, для которых рейтинг не рассчитан или рассчитан «давно».

Когда прекращать обновление? Если анализировать количество обновленных строк и сравнивать с размером пакета, то можно пропустить строки, которые были заблокированы каким-нибудь другим процессом. Поэтому лучше явно проверять наличие строк, требующих обработки.

И еще один момент: поскольку команду `VACUUM` нельзя использовать в функциях и процедурах, мы будем запускать ее в фоновом процессе.

Теперь мы готовы к тому, чтобы написать процедуру обновления. Для удобства передадим в качестве параметра не только размер пакета, но и порог возраста.



```
=> CREATE PROCEDURE update_rating(
    batch_size integer,
    age_threshold interval
    DEFAULT interval '0 sec' -- по умолчанию обработать все
) AS $$
DECLARE
    rows bigint;
    -- будет несколько транзакций, поэтому время запуска надо запомнить
    start_at timestamp := current_timestamp;
BEGIN
    LOOP
        -- обновляем один пакет строк

        WITH batch AS (
            SELECT * FROM books
            WHERE rating IS NULL
            OR start_at - rating_at > age_threshold
            LIMIT batch_size
            FOR UPDATE SKIP LOCKED
        )
        UPDATE books
        SET rating = rating(votes_up, votes_down),
            rating_at = current_timestamp
        WHERE book_id IN (SELECT book_id FROM batch);
        GET DIAGNOSTICS rows := ROW_COUNT;
        RAISE NOTICE 'updated % rows', rows;

        -- фиксируем, чтобы не держать горизонт транзакций
        -- и позволить очистке выполнить свою работу
        COMMIT;

        -- вызываем очистку и ждем окончания
        PERFORM * FROM pg_background_result(pg_background_launch(
            'VACUUM books'
        )) AS (result text);
        RAISE NOTICE 'vacuumed, backend_xmin = %',
            (SELECT backend_xmin FROM pg_stat_activity
             WHERE pid = pg_backend_pid());

        -- остались необработанные строки?
        EXIT WHEN NOT EXISTS (
            SELECT NULL FROM books
            WHERE rating IS NULL
            OR start_at - rating_at > age_threshold
        );
    END LOOP;
    RAISE NOTICE 'all done';
END;
$$ LANGUAGE plpgsql;
```

CREATE PROCEDURE

Проверим:

```
=> CALL update_rating(10, interval '1 day');
```

```
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 869
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 870
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 871
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 872
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 873
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 874
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 875
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 876
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 877
NOTICE: updated 10 rows
NOTICE: vacuumed, backend_xmin = 878
NOTICE: all done
CALL
```

## 2. Расчета рейтинга в фоновом режиме

Просто вызовем процедуру в фоновом процессе. Таким образом, из одного фонового процесса можно вызывать другой (конечно, необходимо, чтобы не было превышено предельное количество фоновых процессов, установленное в системе).

```
=> SELECT * FROM pg_background_result(pg_background_launch(  
    'CALL update_rating(10)'  
)) AS (result text);
```

```
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 879  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 880  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 881  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 882  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 883  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 884  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 885  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 886  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 887  
NOTICE: updated 10 rows  
NOTICE: vacuumed, backend_xmin = 888  
NOTICE: all done  
result  
-----  
CALL  
(1 row)
```

1. Фоновый процесс не будет запущен, если запуск приведет к превышению числа допустимых процессов.  
Напишите функцию-обертку для `pg_background_launch`, которая пытается запустить процесс  $N$  раз с интервалом в одну секунду.
2. Сравните накладные расходы на порождение процесса расширений `dblink` и `pg_background`, многократно выполняя простой запрос.

1. Предварительно исследуйте, как функция `pg_background_launch` сигнализирует об ошибке. Для этого удобно уменьшить значение параметра `max_worker_processes`, но учтите, что один фоновый процесс в системе уже запущен (logical replication launcher) — не уменьшайте значение до 1.

Обработка ошибок рассматривается в курсе DEV1.

2. Разумеется, накладные расходы `dblink` будут существенно зависеть от способа аутентификации (в нашем случае — проверка пароля `scram-sha-256`).

## 1. Превышение числа фоновых процессов

```
=> CREATE DATABASE ext_bgworkers;
```

```
CREATE DATABASE
```

```
=> \c ext_bgworkers
```

You are now connected to database "ext\_bgworkers" as user "student".

```
=> CREATE EXTENSION pg_background;
```

```
CREATE EXTENSION
```

Уменьшим значение параметра-ограничителя до 2, учитывая, что один фоновый процесс (logical replication launcher) запускается системой автоматически. Изменение требует рестарта сервера.

```
=> ALTER SYSTEM SET max_worker_processes = 2;
```

```
ALTER SYSTEM
```

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

```
student$ psql ext_bgworkers
```

```
=> SHOW max_worker_processes;
```

```
max_worker_processes
-----
2
(1 row)
```

Запустим один длительный фоновый процесс:

```
=> SELECT * FROM pg_background_launch(
    'SELECT pg_sleep(10)'
);
```

```
pg_background_launch
-----
129200
(1 row)
```

Теперь попробуем запустить еще один, перехватывая исключение:

```
=> DO $$
DECLARE
    hint text;
BEGIN
    PERFORM pg_background_launch('SELECT 42');
EXCEPTION
    WHEN others THEN
        GET STACKED DIAGNOSTICS hint := PG_EXCEPTION_HINT;
        RAISE NOTICE E'sqlstate = %\nsqlerrm = %\nhint = %',
            SQLSTATE, SQLERRM, hint;
END;
$$;
```

```
NOTICE:  sqlstate = 53000
sqlerrm = could not register background process
hint = You may need to increase max_worker_processes.
DO
```

Ошибка с кодом 53000 относится к группе `insufficient_resources`.

Функция может выглядеть следующим образом:

```

=> CREATE FUNCTION try_background_launch(sql text, retries integer)
RETURNS integer
AS $$
DECLARE
BEGIN
    LOOP
        BEGIN
            RETURN pg_background_launch(sql);
        EXCEPTION
            WHEN insufficient_resources THEN
                IF retries > 0 THEN
                    retries := retries - 1;
                    RAISE NOTICE 'Sleeping for 1 sec, % retries left',
                        retries;
                    PERFORM pg_sleep(1);
                ELSE
                    RAISE;
                END IF;
            END;
        END LOOP;
    END
    $$ LANGUAGE plpgsql VOLATILE;

```

CREATE FUNCTION

Проверим:

```

=> SELECT * FROM pg_background_launch(
    'SELECT pg_sleep(10)'
);

```

```

pg_background_launch
-----
                129729
(1 row)

```

```

=> SELECT * FROM pg_background_result(try_background_launch(
    'SELECT 42', 15
)) AS (result integer);

```

```

NOTICE: Sleeping for 1 sec, 14 retries left
NOTICE: Sleeping for 1 sec, 13 retries left
NOTICE: Sleeping for 1 sec, 12 retries left
NOTICE: Sleeping for 1 sec, 11 retries left
NOTICE: Sleeping for 1 sec, 10 retries left
NOTICE: Sleeping for 1 sec, 9 retries left
NOTICE: Sleeping for 1 sec, 8 retries left
NOTICE: Sleeping for 1 sec, 7 retries left
NOTICE: Sleeping for 1 sec, 6 retries left
NOTICE: Sleeping for 1 sec, 5 retries left
result
-----
      42
(1 row)

```

Восстановим значение параметра по умолчанию:

```

=> ALTER SYSTEM RESET ALL;

```

ALTER SYSTEM

```

student$ sudo pg_ctlcluster 16 main restart

```

## 2. Сравнение dblink и pg\_background

```

student$ psql ext_bgworkers

```

```

=> CREATE EXTENSION dblink;

```

CREATE EXTENSION

```

=> \timing on

```

Timing is on.

Просто запрос (выполняем с помощью динамического SQL, чтобы исключить кеширование плана запроса):

```
=> DO $$
DECLARE
    result integer;
BEGIN
    FOR i IN 1 .. 1000 LOOP
        EXECUTE 'SELECT 1' INTO result;
    END LOOP;
END;
$$;
```

DO  
Time: 7,336 ms

Расширение pg\_background:

```
=> DO $$
DECLARE
    result integer;
BEGIN
    FOR i IN 1 .. 1000 LOOP
        SELECT * INTO result FROM pg_background_result(
            pg_background_launch('SELECT 1')
        ) AS (result integer);
    END LOOP;
END;
$$;
```

DO  
Time: 11560,002 ms (00:11,560)

Расширение dblink:

```
=> DO $$
DECLARE
    result integer;
BEGIN
    FOR i IN 1 .. 1000 LOOP
        SELECT * INTO result FROM dblink(
            'host=localhost port=5432 dbname=postgres user=postgres password=postgres',
            'SELECT 1'
        ) AS (result integer);
    END LOOP;
END;
$$;
```

DO  
Time: 22622,650 ms (00:22,623)