Авторы: Алексей Береснев, Илья Баштанов, Павел Толмачев

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

Темы

64-битные номера транзакций

Автономные транзакции

Встроенный пул соединений

64-битные транзакции

Формат данных

Заморозка

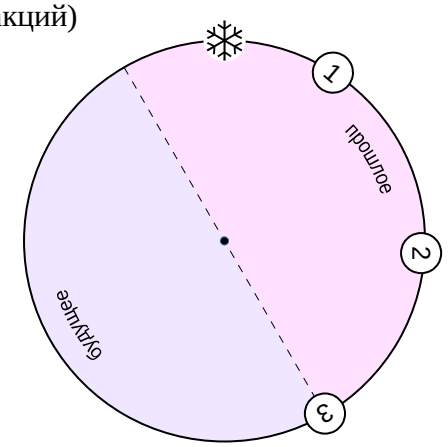
32-битные номера

PostgreSQL, Postgres Pro Standard

32-х разрядный xid (~ 4 млрд. транзакций)

зацикливание идентификаторов

необходимость заморозки



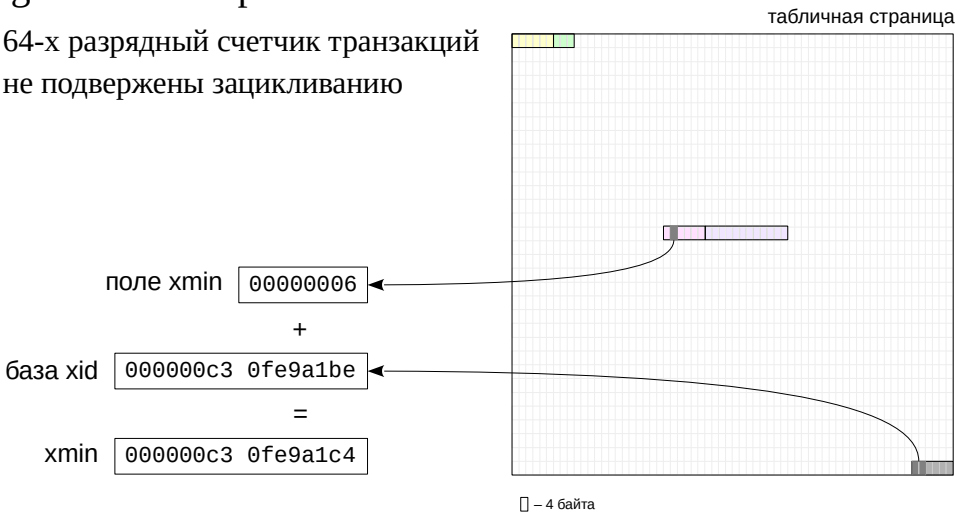
32-битные идентификаторы транзакций допускают примерно четыре миллиарда уникальных значений. В современных реалиях при большой нагрузке идентификаторы могут закончиться через несколько дней и работа СУБД станет невозможна. Поэтому применяют заморозку: версии строк с достаточно большими xmin помечаются как очень старые и, следовательно, видимые во всех снимках данных. После заморозки версий с определенным номером во всех таблицах кластера номер можно использовать повторно (т. н. transaction wraparound).

Заморозка выполняется как одна из задач процесса очистки и активно использует подсистему ввода-вывода. При большой нагрузке требуется постоянно мониторить сервер и принимать оперативные меры, если заморозка не успевает освобождать номера транзакций.

64-битные номера

Postgres Pro Enterprise

64-х разрядный счетчик транзакций
не подвержены зацикливанию



5

Использование 64-битного счетчика позволило бы иметь практически бесконечное число идентификаторов транзакций, отказаться от их повторного использования и связанных с этим действий по администрированию.

Но простое увеличение разрядности идентификаторов привело бы к увеличению размера заголовков версий строк и, как следствие, к повышенному расходу дискового пространства и ресурсов ввода-вывода при обработке данных.

Поэтому в Postgres Pro Enterprise реализован компромиссный вариант 64-х битных идентификаторов: в специальной области в конце страницы хранится 64-битное *базовое значение* xid, а в полях xmin и xmax заголовка версии строки — 32-битное *смещение*. 64-битный номер транзакции xmin или xmax вычисляется сложением базового значения и значения, хранящегося в заголовке версии строки.

Такой же формат применяется для идентификаторов мультитранзакций: базовое значение хранится в специальной области страницы, а смещение — в поле xmax заголовка версии строки.

https://habr.com/ru/company/postgrespro/blog/707968/

Заморозка

Синхронная

если смещения не хватает и необходимо увеличить базовое значение

При очистке

уменьшает pg_xact и pg_multixact

vacuum_freeze_min_age = 50 млн

vacuum_freeze_table_age = 150 млн

autovacuum_freeze_max_age = 10 млрд

vacuum_multixact_freeze_min_age = 5 млн

vacuum_multixact_freeze_table_age = 150 млн

autovacuum_multixact_freeze_max_age = 20 млрд

Поскольку смещения в заголовке версии строки 32-битные, диапазон значений xid в пределах страницы — от базового до базового плюс $2^{32}-1$. Если в какую-либо версию строки на странице нужно записать больший номер, сервер увеличивает базовое значение xid. Если при этом версии строк с небольшими номерами xid препятствуют такому увеличению, сервер замораживает эти версии «на лету».

Поскольку 64-битные номера транзакций не используются повторно, своевременная заморозка не критична для работы многоверсионности: параметр *autovacuum_freeze_max_age* имеет значение по умолчанию 10 млрд вместо 200 млн в обычном PostgreSQL, и значение *autovacuum_multixact_freeze_max_age* также увеличено. Однако заморозка все же выполняется как часть процедуры очистки, чтобы уменьшать размер CLOG (подкаталога pg_xact) и статусов мультитранзакций (pg_multixact).

Ограничением этого решения является то, что очень долгие транзакции (возраст которой превышает 2^{32}), удерживающие снимок, будут мешать замораживать версии строк. Поэтому, хотя необходимость в регулярной заморозке отпадает, долгих транзакций по-прежнему стоит избегать.

https://postgrespro.ru/docs/enterprise/13/routine-vacuuming#VACUUM-FOR-WRAPAROUND

Структура страниц

Чтобы заглянуть в страницы, воспользуемся расширением pageinspect.

```
=> CREATE DATABASE transactions;
```

```
CREATE DATABASE
```

```
=> \c transactions
```

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

```
=> CREATE EXTENSION pageinspect;
```

```
CREATE EXTENSION
```

В тестовой таблице будет две страницы по две версии строки в каждой.

```
=> CREATE TABLE test(s CHAR(300)) WITH (fillfactor=10);
```

```
CREATE TABLE
```

```
=> INSERT INTO test SELECT g.n FROM generate_series(1,4) AS g(n);
```

```
INSERT 0 4
```

Значения xmin и xmax в заголовках версий строк:

```
=> SELECT ctid, xmin, xmax FROM test;
```

```
 ctid  | xmin | xmax
-------+------+------
 (0,1) |  742 |    0
 (0,2) |  742 |    0
 (1,1) |  742 |    0
 (1,2) |  742 |    0
(4 rows)
```

В конце каждой страницы данных находится специальная область размером 24 байта:

```
=> SELECT p.page, h.lower, h.upper, h.special, h.pagesize
FROM
  (VALUES(0),(1)) p(page),
  page_header(get_raw_page('test', p.page)) h
;
```

```
 page | lower | upper | special | pagesize
------+-------+-------+---------+----------
    0 |    28 |  7512 |    8168 |     8192
    1 |    28 |  7512 |    8168 |     8192
(2 rows)
```

В специальной области хранится базовое значение xid, а в заголовках версий строк — 32-битные смещения:

```
=> SELECT p.page, substr(f,length(f)-23,8) xid_base
FROM
  (VALUES(0),(1)) p(page),
  get_raw_page('test', p.page) f;
```

```
 page |      xid_base
------+--------------------
    0 | \xe302000000000000
    1 | \xe302000000000000
(2 rows)
```

Текущее значение счетчика транзакций:

```
SELECT pg_current_xact_id();
```

```
 pg_current_xact_id
--------------------
                743
(1 row)
```

Пусть теперь сервер выполнит 2^32 транзакций. Выполнение такого количества транзакций работающим сервером заняло бы очень много времени, поэтому остановим его и воспользуемся утилитой pg_resetwal, чтобы поменять текущее значение счетчика.

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

postgres$ /opt/pgpro/ent-13/bin/pg_resetwal -x 4294968039 -D /var/lib/pgpro/ent-13

Write-ahead log reset

student$ sudo -u postgres truncate /var/lib/pgpro/ent-13/pg_xact/000000000040 --size=16M

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

Иногда после изменения счетчика сервер не запускается из-за отсутствия файла статуса транзакции. В этом случае в журнале появляется сообщение с именем отсутствующего файла, нужно создать пустой файл размером 16 Мбайт в каталоге /var/lib/pgpro/ent-13/xact и повторить попытку запуска.

```
student$ psql transactions

=> SELECT pg_current_xact_id();

 pg_current_xact_id
--------------------
         4294968039
(1 row)
```

Внимание! Не рекомендуется применять утилиту pg_resetwal в производственной среде, поскольку она изменяет внутренние структуры данных и есть опасность потери информации.

Теперь изменим одну строку таблицы.

```
=> UPDATE test SET s = '10' WHERE s = '1';

UPDATE 1
```

При этом сервер запишет номер текущей транзакции в поле xmax старой версии строки и в поле xmin новой версии, которая будет размещена в нулевой странице.

Поскольку номер текущей транзакции отличается от значений xmin в существующих версиях более чем на 2^32, сервер увеличит базовый xid и выполнит локальную заморозку в нулевой странице: выставит в заголовках версий соответствующий флаг, а в поле xmin запишет специальное значение 2. В странице 1 заморозка не делается:

```
=> SELECT (page,lp) ctid,t_xmin,t_xmax,f.combined_flags
FROM
  (VALUES(0),(1)) p(page),
  heap_page_items(get_raw_page('test', p.page)),
  heap_tuple_infomask_flags(t_infomask, t_infomask2) f
;

 ctid  |   t_xmin   |   t_xmax   |   combined_flags
-------+------------+------------+--------------------
 (0,1) |          2 | 4294968040 | {HEAP_XMIN_FROZEN}
 (0,2) |          2 |          0 | {HEAP_XMIN_FROZEN}
 (0,3) | 4294968040 |          0 | {}
 (1,1) |        742 |          0 | {}
 (1,2) |        742 |          0 | {}
(5 rows)
```

Базовый номер xid изменился только в нулевой странице:

```
=> SELECT p.page, substr(f,length(f)-23,8) xid_base
FROM
  (VALUES(0),(1)) p(page),
  get_raw_page('test', p.page) f;

 page |      xid_base
------+--------------------
    0 | \xe60200ff00000000
    1 | \xe302000000000000
(2 rows)
```

Автономные транзакции

Устройство автономных транзакций

Видимость данных в автономных транзакциях

Автономные транзакции в SQL

Автономные транзакции в PL/pgSQL и PL/Python

Ограничения автономных транзакций

Автономные транзакции

Автономная транзакция — это независимая транзакция, которая запускается внутри родительской, а фиксируется (или отменяется) до завершения родительской.

```
BEGIN;
  команды ...
  BEGIN AUTONOMOUS [ISOLATION LEVEL ...];
    команды ...
  COMMIT [AUTONOMOUS];  -- или ROLLBACK
  команды ...
COMMIT;    -- или ROLLBACK
```

В начале автономной транзакции основная транзакция приостанавливается, и последующие изменения данных относятся к дочерней автономной транзакции. По завершении автономной, родительская транзакция продолжает работу.

В отличие от вложенной транзакции, которая возникает в обычном PostgreSQL при выполнении команды SAVEPOINT (и, возможно, ROLLBACK TO), результат автономной транзакции не зависит от исхода родительской.

Автономная транзакция может включать в себя другие автономные транзакции, однако транзакция верхнего уровня не может быть автономной.

В PL/pgSQL есть дополнительная конструкция, в этом случае операторы блока выполняются в отдельной автономной транзакции:

```
BEGIN AUTONOMOUS
  операторы ...
END;
```

По сути, такая запись — это просто сокращение для

```
BEGIN
  BEGIN AUTONOMOUS;
    операторы ...
  COMMIT;
END;
```

Ошибка в блоке с автономной транзакцией приводит к ее откату, после чего обрабатывается в секции EXCEPTION или передается в вызывающий блок.

В PL/Python есть метод, позволяющий выполнить SQL в автономной транзакции:

```
with plpy.autonomous():
  plpy.execute(...)
```

Видимость

Снимки родительской и автономных транзакций независимы

- Read Committed видит изменения
- Repeatable Read, Serializable не видят изменений

Возможны взаимоблокировки

Родительская и дочерняя автономная транзакция используют собственные снимки данных, время жизни которых зависит от уровня изоляции. Поэтому родительская транзакция с уровнем изоляции Read Committed будет видеть все изменения дочерних автономных транзакций, а родительская с уровнем изоляции Repeatable Read или Serializable не увидит никаких изменений.

Автономная транзакция может вызвать взаимоблокировку, если она попытается получить блокировку ресурса, уже заблокированного родительской транзакцией.

Ограничения

max_autonomous_transactions = 100

Вложенность до 128 уровней

Не поддерживается блокировка FOR UPDATE

Нельзя обращаться к временным схемам

Нельзя изменить уровень изоляции в PL/pgSQL и PL/Python

Не поддерживаются в расширениях

- in_memory
- online_analyze

Нельзя использовать слоты логической репликации

Во всех сеансах можно запустить не более *max_autonomous_transactions* автономных транзакций одновременно (по умолчанию 100).

Глубина вложенности автономных транзакций — не более 128.

FOR UPDATE не поддерживается, поэтому, например, нельзя заблокировать строку в родительской транзакции, а в дочерней автономной пропустить ее с помощью SKIP LOCKED.

В автономной транзакции нельзя обращаться к объектам временных схем, в частности, ко временным таблицам.

В блоке PL/pgSQL и методе PL/Python нельзя задать уровень изоляции автономной транзакции, отличающийся от текущего.

Автономные транзакции не работают с расширениями in_memory и online_analyze.

В автономной транзакции нельзя использовать слоты логической репликации.

https://postgrespro.ru/docs/enterprise/13/atx

Использование автономных транзакций

Обычно автономные транзакции используются для аудита, когда нужно сохранить информацию о событии, произошедшем в родительской транзакции, независимо от ее исхода.

Опустошим таблицу:

```
=> TRUNCATE test;
```

```
TRUNCATE TABLE
```

Мы хотим записывать в таблицу test_audit информацию обо всех изменениях таблицы test, в том числе и о тех, которые не были зафиксированы.

```
=> CREATE TABLE test_audit(
  time timestamptz,
  username text,
  operation text
);
```

```
CREATE TABLE
```

Для этого понадобятся триггер и триггерная функция, тело которой выполняется в автономной транзакции.

```
=> CREATE FUNCTION test_audit() RETURNS trigger AS $$
BEGIN AUTONOMOUS
  INSERT INTO test_audit VALUES (now(), current_user, tg_op);
  RETURN new;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER test_audit
AFTER INSERT OR UPDATE OR DELETE
ON test
FOR EACH ROW
EXECUTE FUNCTION test_audit();
```

```
CREATE TRIGGER
```

Теперь информация об изменениях будет сохраняться в таблице test_audit, даже если транзакция впоследствии обрывается.

```
=> BEGIN;
```

```
BEGIN
```

```
=> INSERT INTO test VALUES ('value1'),('value2');
```

```
INSERT 0 2
```

```
=> UPDATE test SET s = 'value3';
```

```
UPDATE 2
```

```
=> DELETE FROM test;
```

```
DELETE 2
```

Откатим транзакцию:

```
=> ROLLBACK;
```

```
ROLLBACK
```

В таблице ничего не осталось:

```
=> SELECT * FROM test;
```

```
 s
---
(0 rows)
```

```
=> SELECT * FROM test_audit;
```

```
             time              | username | operation
-------------------------------+----------+-----------
 2024-01-16 21:32:19.801061+03 | student  | INSERT
 2024-01-16 21:32:19.801061+03 | student  | INSERT
 2024-01-16 21:32:19.81384+03  | student  | UPDATE
 2024-01-16 21:32:19.81384+03  | student  | UPDATE
 2024-01-16 21:32:19.826201+03 | student  | DELETE
 2024-01-16 21:32:19.826201+03 | student  | DELETE
(6 rows)
```

Однако попытки изменить данные записаны в таблицу аудита.

Пул соединений

Проблема и решения

Встроенный пул

Подключение

Ограничения пула

Проблема и решения

Один сеанс — один обслуживающий процесс

- повышенный расход ресурсов
- увеличение конкуренции
- прогрев локальных кешей

Пул соединений в сервере приложений или драйвере

- J2EE, JDBC, ODBC, node-postgres, Psycopg2

Пул соединений как сторонний продукт

- PgBouncer, Odyssey

Встроенный пул Postgres Pro Enterprise сохраняет контекст сеанса

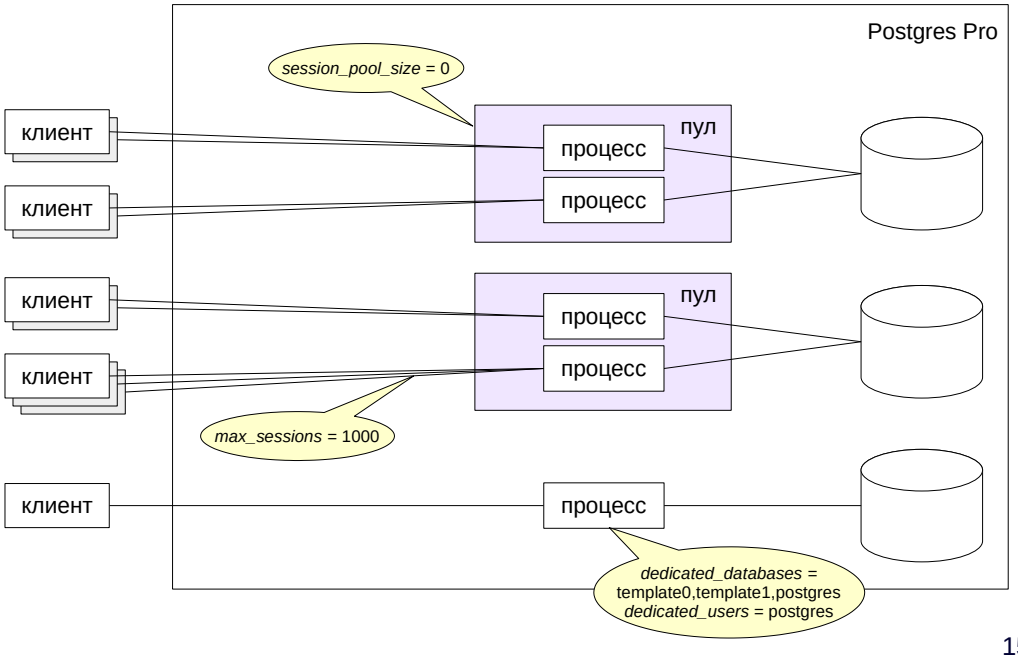
Простейшая архитектура, при которой для каждого клиентского соединения запускается отдельный обслуживающий процесс, имеет ряд недостатков, которые при большом количестве клиентов становятся критичными. Увеличивается конкуренция при обращении к ресурсам, растут накладные расходы на запуск процессов (в частности, на наполнение кешей) и освобождение ресурсов при их завершении. Кроме того, размер многих структур в общей памяти сервера пропорционален числу клиентских соединений.

По этим причинам в нагруженные многопользовательские системы добавляют пул соединений, позволяющий нескольким клиентам использовать одно соединение с сервером. Как правило, для этого либо применяют отдельное решение, такое как PgBouncer или Odyssey, либо задействуют возможности сервера приложений или драйвера PostgreSQL.

При таком подходе администратору приходится обслуживать дополнительную компоненту архитектуры, а разработчику — учитывать особенности работы приложения с пулом соединений.

Встроенный пул соединений Postgres Pro Enterprise позволяет воспользоваться преимуществами пула, сохранив при этом большинство возможностей, связанных с контекстом сеанса.

Встроенный пул



Для каждой базы данных создается отдельный пул, количество пулов не ограничено. Если количество процессов в пуле меньше значения *session_pool_size*, для нового сеанса запускается новый процесс, а если предел достигнут, сеанс привязывается к одному из существующих процессов.

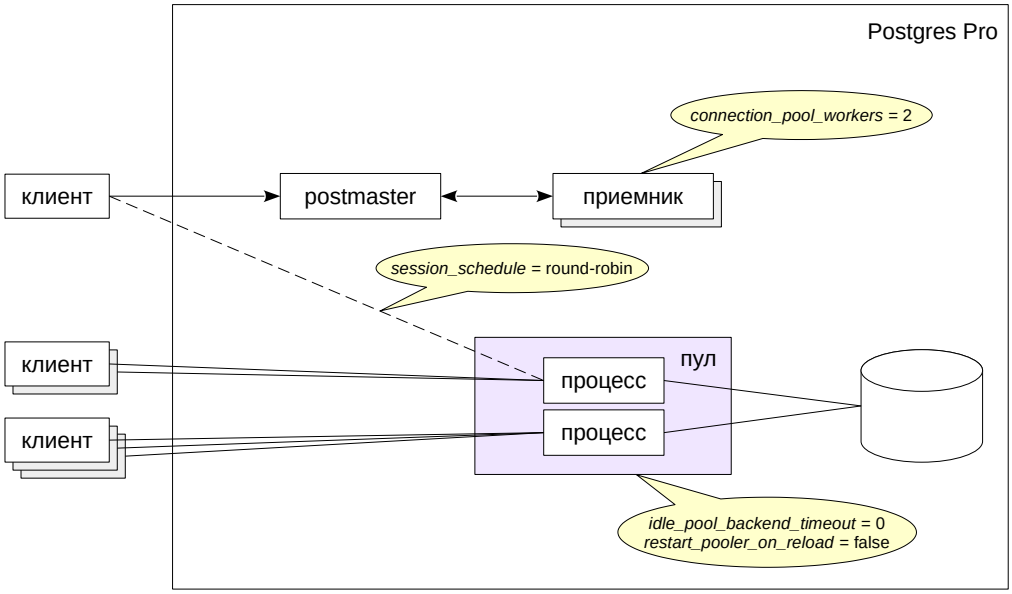
Процесс, к которому привязано несколько сеансов, поочередно обслуживает их транзакции, сохраняя при этом контекст каждого сеанса. Поэтому значение *session_pool_size* следует задавать достаточно большим, чтобы клиент не ждал слишком долго, пока завершится транзакция другого соединения.

Одновременно процесс обслуживает не больше *max_sessions* сеансов. Если число соединений с одной базой достигнет *max_sessions* × × *session_pool_size*, следующая попытка подключения вернет ошибку.

Если клиент соединяется с базой данных, входящей в список *dedicated_databases*, или подключается под ролью, входящей в *dedicated_users*, для него будет создан выделенный процесс, не входящий в пулы. По умолчанию это сеансы пользователя postgres и подключения к базам template0, template1 и postgres. Количество выделенных процессов не ограничено.

https://postgrespro.ru/docs/enterprise/13/how-connection-pooler-works

Подключение



Подключение нового клиента устроено следующим образом.

Процесс postmaster слушает входящие соединения. Подключаясь, клиент передает стартовый пакет данных, в котором указаны параметры соединения — имя базы данных, имя роли. Postmaster передает этот пакет одному из процессов-приемников для анализа. Количество процессов-приемников определяется значением параметра *connection_pool_workers* и должно быть достаточным, чтобы анализ входящих соединений не стал узким местом.

Получив от приемника информацию о базе данных и роли, postmaster направляет соединение в подходящий пул. Применяется одна из трех стратегий, которые можно указать в параметре *session_schedule*: round-robin назначает процессы по очереди, random — случайным образом, load-balancing выбирает процесс с наименьшим числом сеансов.

Когда все соединения, использующие какой-то процесс, завершаются, процесс продолжает работать. Автоматическое завершение неиспользуемого процесса по истечении определенного времени настраивается параметром *idle_pool_backend_timeout*.

Для перезагрузки рабочих процессов можно установить параметр *restart_pooler_on_reload* = true и перечитать конфигурацию.

https://postgrespro.ru/docs/enterprise/13/connection-pooler-configuration

Ограничения пула

Не поддерживаются SSL-соединения

Долгие транзакции задерживают работу других соединений

Не поддерживаются рекомендательные блокировки уровня сеанса

Нельзя перехватывать ошибку «connection to client lost»

Не работают модули, полагающиеся на состояние сеанса

Нельзя изменять параметры в других сеансах

Остановка сервера в режиме smart работает как fast

Параметр *idle_session_timeout* игнорируется

Нужно учитывать следующие ограничения при использовании пула.

- SSL-соединения в текущей реализации не поддерживаются.
- Долгая транзакция будет задерживать работу транзакций в других соединениях, использующих тот же процесс.
- Ошибка «connection to client lost» обрабатывается пулом, чтобы завершать зависшие соединения по таймауту, перехватывать ее не нужно.
- Не работают модули, использующие состояние на уровне сеанса, такие как plantuner, а также расширения multimaster, in_memory, online_analyze, pg_variables.
- Нельзя изменять параметры других сеансов.
- Не поддерживается передача сообщений между процессами (команды LISTEN, NOTIFY и т. п.).
- Остановка сервера в режиме smart работает как fast.
- Решения 1С не поддерживаются, они используют собственный пул соединений.
- Тайм-аут *idle_session_timeout* не работает.

https://postgrespro.ru/docs/enterprise/13/connection-pooler-limitations

Встроенный пул соединений

Выделим в пуле по два процесса для каждой базы данных:

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

ALTER SYSTEM

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

student$ psql
```

Если начать транзакцию в основном сеансе, ей будет выделен один из процессов:

```
=> \c transactions

You are now connected to database "transactions" as user "student".

=> BEGIN;

BEGIN

=> SELECT pg_backend_pid();

 pg_backend_pid
----------------
         254870
(1 row)
```

Сеанс может хранить свое состояние в пользовательских параметрах:

```
=> SELECT set_config('my.name', 'Я — сеанс 1', false);

 set_config
-------------
 Я — сеанс 1
(1 row)
```

Транзакции, начавшейся в другом сеансе, будет выделен другой процесс:

```
student$ psql

  => \c transactions

  You are now connected to database "transactions" as user "student".

  => BEGIN;

  BEGIN

  => SELECT pg_backend_pid();

   pg_backend_pid
  ----------------
           255034
  (1 row)


  => SELECT set_config('my.name', 'Я — сеанс 2', false);

   set_config
  -------------
   Я — сеанс 2
  (1 row)
```

По умолчанию процессы выделяются по очереди (стратегия round-robin), поэтому третьему сеансу будет назначен процесс 254870. Но этот процесс занят выполнением транзакции первого сеанса, поэтому третий сеанс будет ждать:

```
student$ psql

    => \c transactions
```

Когда транзакция первого сеанса заканчивается...

```
=> COMMIT;

COMMIT
```

...третий сеанс может воспользоваться освободившимся процессом 254870:

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

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

```
 pg_backend_pid
----------------
         254870
(1 row)
```

```
=> SELECT set_config('my.name', 'Я — сеанс 3', false);
```

```
 set_config
-------------
 Я — сеанс 3
(1 row)
```

Заметим, что состояние сеансов сохраняется, несмотря на то что процесс выполняет транзакции разных сеансов:

```
=> SELECT pg_backend_pid(), current_setting('my.name');
```

```
 pg_backend_pid | current_setting
----------------+-----------------
         254870 | Я — сеанс 1
(1 row)
```

```
=> SELECT pg_backend_pid(), current_setting('my.name');
```

```
 pg_backend_pid | current_setting
----------------+-----------------
         254870 | Я — сеанс 3
(1 row)
```

Итоги

64-битные идентификаторы транзакций упрощают обслуживание данных при высокой нагрузке

Автономные транзакции дают возможность удобно и эффективно реализовать аудит

Встроенный пул позволяет ограничить число соединений, не жертвуя контекстом сеанса

Практика

1. Воспроизведите взаимоблокировку основной транзакции и вложенной в нее автономной.
2. Реализуйте аудит изменений данных в таблице двумя способами: используя расширение dblink и автономные транзакции. Сравните производительность.
3. Убедитесь, что при использовании встроенного пула соединений сохраняется содержимое временных таблиц уровня сеанса.

1. Взаимоблокировка в автономной транзакции

Создадим базу данных:

```
=> CREATE DATABASE transactions;
```

```
CREATE DATABASE
```

```
=> \c transactions
```

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

```
=> CREATE TABLE t1 AS SELECT 1 n;
```

```
SELECT 1
```

Начинаем транзакцию и блокируем строку:

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE t1 SET n=2;
```

```
UPDATE 1
```

Пытаемся блокировать ту же строку в автономной транзакции и получаем ошибку:

```
=> BEGIN AUTONOMOUS;
```

```
BEGIN
```

```
=> UPDATE t1 SET n=3;
```

```
ERROR:  an ATX waiting for an ancestor transaction will cause a deadlock
```

```
=> END AUTONOMOUS;
```

```
ROLLBACK
```

Основную транзакцию можно зафиксировать:

```
=> COMMIT;
```

```
COMMIT
```

2. Аудит: dblink и автономные транзакции

Тестовая таблица:

```
=> CREATE TABLE test(n int);
```

```
CREATE TABLE
```

Таблица для аудита:

```
=> CREATE TABLE test_audit(
  time timestamptz,
  username text,
  operation text
);
```

```
CREATE TABLE
```

Первый вариант триггерной функции использует расширение dblink:

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

```
CREATE EXTENSION
```

```
=> CREATE OR REPLACE FUNCTION test_audit() RETURNS trigger AS $$
BEGIN
  PERFORM dblink(
    'dbname='||current_database(),
    'INSERT INTO test_audit VALUES (now(), current_user, '''||tg_op||''')'
  );
  RETURN new;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Триггер:

```
=> CREATE TRIGGER test_audit
AFTER INSERT OR UPDATE OR DELETE
ON test
FOR EACH ROW
EXECUTE FUNCTION test_audit();
```

```
CREATE TRIGGER
```

Замеряем время.

```
=> \timing on
```

```
Timing is on.
```

```
=> INSERT INTO test SELECT g.s FROM generate_series(1,500) AS g(s);
```

```
INSERT 0 500
Time: 1479,787 ms (00:01,480)
```

```
=> \timing off
```

```
Timing is off.
```

Опустошим обе таблицы и заменим триггерную функцию на вариант, использующий автономную транзакцию.

```
=> TRUNCATE test;
```

```
TRUNCATE TABLE
```

```
=> TRUNCATE test_audit;
```

```
TRUNCATE TABLE
```

```
=> CREATE OR REPLACE FUNCTION test_audit() RETURNS trigger AS $$
BEGIN AUTONOMOUS
  INSERT INTO test_audit VALUES (now(), current_user, tg_op);
  RETURN new;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Еще раз замеряем время.

```
=> \timing on
```

```
Timing is on.
```

```
=> INSERT INTO test SELECT g.s FROM generate_series(1,500) AS g(s);
```

```
INSERT 0 500
Time: 613,261 ms
```

```
=> \timing off
```

```
Timing is off.
```

Автономные транзакции работают быстрее, поскольку нет накладных расходов на установку соединения.

3. Временные таблицы и пул соединений

Включим пул соединений, выделив один процесс для каждой базы данных. Транзакции сеансов будут использовать его по очереди:

```
=> ALTER SYSTEM SET session_pool_size=1;
```

```
ALTER SYSTEM
```

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

```
student$ psql
```

```
student$ psql
```

В каждом из двух сеансов создадим временную таблицу.

```
=> CREATE TEMP TABLE temp (
  session int DEFAULT 1,
  pid int DEFAULT pg_backend_pid()
);
```

```
CREATE TABLE
```

```
  => CREATE TEMP TABLE temp (
    session int DEFAULT 2,
    pid int DEFAULT pg_backend_pid()
  );
```

```
  CREATE TABLE
```

Теперь пусть сеансы по очереди вставляют строки в таблицы.

```
=> INSERT INTO temp SELECT;
```

```
INSERT 0 1
```

```
  => INSERT INTO temp SELECT;
```

```
  INSERT 0 1
```

```
=> INSERT INTO temp SELECT;
```

```
INSERT 0 1
```

```
  => INSERT INTO temp SELECT;
```

```
  INSERT 0 1
```

Вот содержимое временных таблиц:

```
=> SELECT * FROM temp;
```

```
 session |  pid
---------+--------
       1 | 295140
       1 | 295140
(2 rows)
```

```
  => SELECT * FROM temp;
```

```
   session |  pid
  ---------+--------
         2 | 295140
         2 | 295140
  (2 rows)
```

Встроенный пул соединений Postgres Pro Enterprise сохраняет временные таблицы уровня сеанса. При использовании сторонних менеджеров пула сохранение контекста сеанса не гарантируется.