Репликация
Физическая репликация

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

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

Темы

Задачи репликации

Схема работы физической репликации

Способы доставки журнальных записей

Особенности и ограничения использования реплики

Синхронная и асинхронная репликация

Мониторинг репликации

Возможные проблемы и способы их решения

Задачи репликации

Репликация

процесс синхронизации нескольких копий кластера баз данных на разных серверах

Базовый механизм для решения ряда задач

отказоустойчивость	при возникновении сбоя система должна сохранить работоспособность
масштабируемость	распределение нагрузки между серверами

Одиночный сервер, управляющий базами данных, может не удовлетворять требованиям, предъявляемым к системе. Есть две основных задачи, для решения которых требуется наличие нескольких серверов.

Во-первых, отказоустойчивость: один физический сервер — это возможная точка отказа. Если сервер выходит из строя, система становится недоступной.

Во-вторых, производительность. Один сервер может не справляться с нагрузкой. Зачастую желательна возможность не вертикального, а горизонтального масштабирования — распределения нагрузки на несколько серверов. Дело может быть как в стоимости аппаратуры, так и в необходимости различных настроек для разных типов нагрузки (например, для OLTP и отчетности).

В случае PostgreSQL распределенные системы строятся по принципу «shared nothing»: несколько серверов работают независимо друг от друга и не имеют ни общей оперативной памяти, ни общих дисков. Следовательно, если серверы должны работать с одними и теми же данными, то эти данные требуется синхронизировать между ними. Механизм синхронизации и называется репликацией.

Мастер → реплика

Записи WAL передаются на реплику и применяются

- поток данных только в одну сторону
- реплицируется кластер целиком, выборочная репликация невозможна

Реплика — точная копия мастера

- одна и та же основная версия сервера
- полностью совместимые архитектуры и платформы

Реплика доступна только для чтения

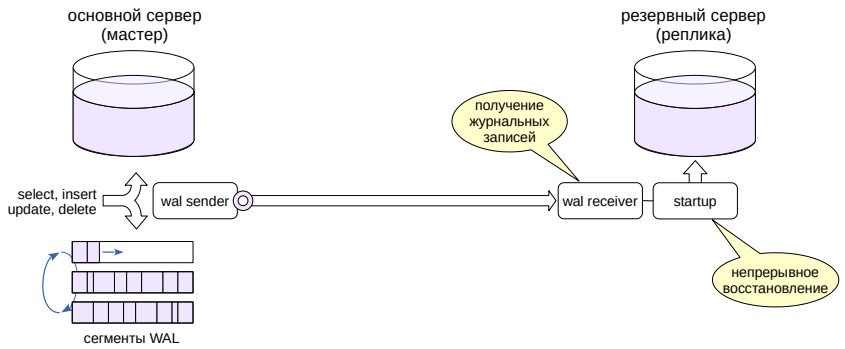
При физической репликации, рассматриваемой в этой теме, серверы имеют назначенные роли: один является ведущим («мастер») и один или несколько серверов являются ведомыми («реплики»).

Мастер передает на реплику журнальные записи, а реплика применяет эти записи к своим файлам данных. Применение происходит чисто механически, без «понимания смысла» изменений, и выполняется быстрее, чем работало бы повторное выполнение операторов SQL. Но при этом критична двоичная совместимость между серверами — они должны работать на одной и той же программно-аппаратной платформе и на них должны быть запущены одинаковые основные версии PostgreSQL.

Поскольку журнал является общим для всего кластера, то и реплицировать можно только кластер целиком, а не отдельные базы данных или таблицы. Возможность «отфильтровать» журнальные записи отсутствует.

Реплика не может генерировать собственных журнальных записей, она лишь применяет записи мастера. Поэтому при таком подходе реплика может быть доступна только для чтения — никакие операции, изменяющие данные, на реплике не допустимы.

Потоковая репликация



Есть два способа доставки журналов от мастера к реплике. Основной, который используется на практике — потоковая репликация.

В этом случае реплика подключается к мастеру по протоколу репликации и читает поток записей WAL. За счет этого при потоковой репликации отставание реплики от мастера минимально (и при необходимости может быть сведено к нулю).

На реплике за прием журнальных записей отвечает процесс wal receiver. Запустившись, он подключается к мастеру по протоколу репликации, а мастер запускает процесс wal sender, который обслуживает это соединение. Полученные записи пишутся на диск в виде сегментов WAL так же, как это происходит на мастере. Далее они (сразу или с задержкой) применяются к файлам данных. За применение отвечает процесс startup — тот же самый, который обеспечивает восстановление после сбоев, только теперь работает постоянно.

Потоковая репликация может, если нужно, использовать слот репликации.

Настройка потоковой репликации

Поскольку в нашей конфигурации не будет архива журнала предзаписи, важно на всех этапах использовать слот репликации — иначе при определенной задержке мастер может успеть удалить необходимые сегменты и весь процесс придется повторять с самого начала.

Создаем слот:

```
α=> SELECT pg_create_physical_replication_slot('replica');

 pg_create_physical_replication_slot
-------------------------------------
 (replica,)
(1 row)
```

Посмотрим на созданный слот:

```
α=> SELECT * FROM pg_replication_slots \gx

-[ RECORD 1 ]-------+---------
slot_name           | replica
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | f
active_pid          |
xmin                |
catalog_xmin        |
restart_lsn         |
confirmed_flush_lsn |
wal_status          |
safe_wal_size       |
```

Вначале слот не инициализирован (restart_lsn и wal_status пустые).

Как мы помним из модуля «Резервное копирование», все необходимые настройки есть по умолчанию:

- wal_level = replica;
- max_wal_senders;
- разрешение на подключение в pg_hba.conf.

Создадим автономную резервную копию, используя созданный слот. Копию расположим в подготовленном каталоге. С ключом -R утилита создает файлы, необходимые для будущей реплики.

```
student$ pg_basebackup --pgdata=/home/student/backup -R --slot=replica
```

Снова проверим слот:

```
α=> SELECT * FROM pg_replication_slots \gx

-[ RECORD 1 ]-------+----------
slot_name           | replica
plugin              |
slot_type           | physical
datoid              |
database            |
temporary           | f
active              | f
active_pid          |
xmin                |
catalog_xmin        |
restart_lsn         | 0/4000000
confirmed_flush_lsn |
wal_status          | reserved
safe_wal_size       |
```

После выполнения резервной копии слот инициализировался, и мастер теперь хранит все файлы журнала с начала копирования (restart_lsn, wal_status).

Сравните со строкой в backup_label:

```
student$ head -n 1 /home/student/backup/backup_label

START WAL LOCATION: 0/4000028 (file 000000010000000000000004)
```

Файл postgresql.auto.conf был подготовлен утилитой pg_basebackup, поскольку мы указали ключ -R. Он содержит информацию для подключения к мастеру (primary_conninfo) и имя слота репликации (primary_slot_name):

```
student$ cat /home/student/backup/postgresql.auto.conf

# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
primary_conninfo = 'user=student passfile=''/home/student/.pgpass'' channel_binding=prefer host=''/var/run/postgresql'' port=5432 sslmode=prefer sslcompression=0 sslsni=1 ssl_min_proto
primary_slot_name = 'replica'
```

По умолчанию реплика будет «горячей», то есть сможет выполнять запросы во время восстановления. Если такая возможность не нужна, реплику можно сделать «теплой» (hot_standby = off).

Утилита также создала сигнальный файл standby.signal, наличие которого указывает серверу войти в режим постоянного восстановления.

```
student$ ls -l /home/student/backup/standby.signal

-rw------- 1 student student 0 ноя  1 22:56 /home/student/backup/standby.signal
```

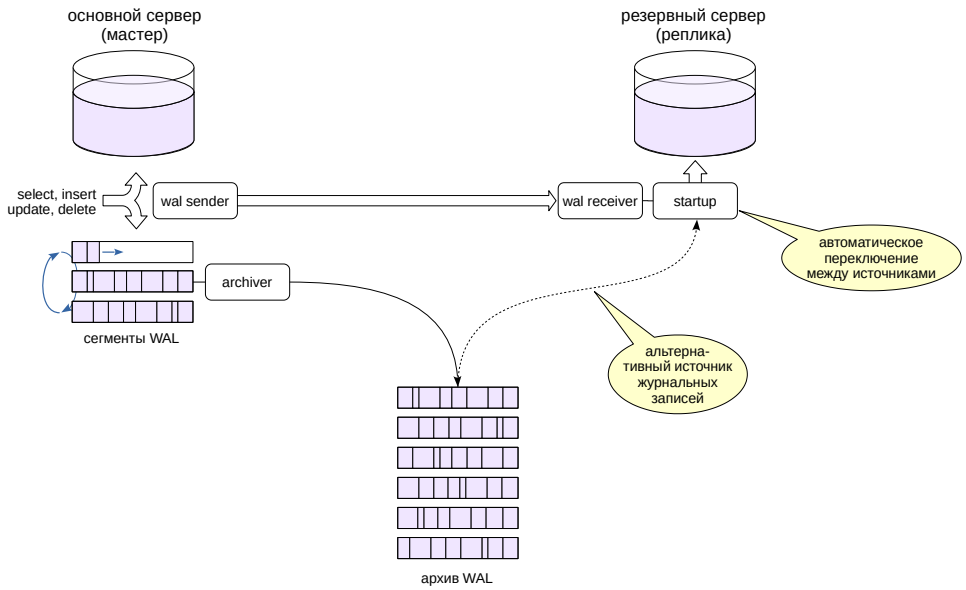
Выкладываем резервную копию в каталог данных сервера beta.

```
student$ sudo pg_ctlcluster 13 beta status

Error: /var/lib/postgresql/13/beta is not accessible or does not exist

student$ sudo rm -rf /var/lib/postgresql/13/beta

student$ sudo mv /home/student/backup /var/lib/postgresql/13/beta

student$ sudo chown -R postgres:postgres /var/lib/postgresql/13/beta
```

Журнальные записи, необходимые для восстановления согласованности, реплика получит от мастера по протоколу репликации. Далее она войдет в режим непрерывного восстановления и продолжит получать и проигрывать поток записей.

```
student$ sudo pg_ctlcluster 13 beta start
```

Репликация с архивом



Другой способ доставки журнальных записей состоит в том, чтобы предоставить реплике доступ к архиву журналов — точно так же, как при обычном восстановлении из архива (этот механизм рассмотрен в модуле «Резервное копирование»).

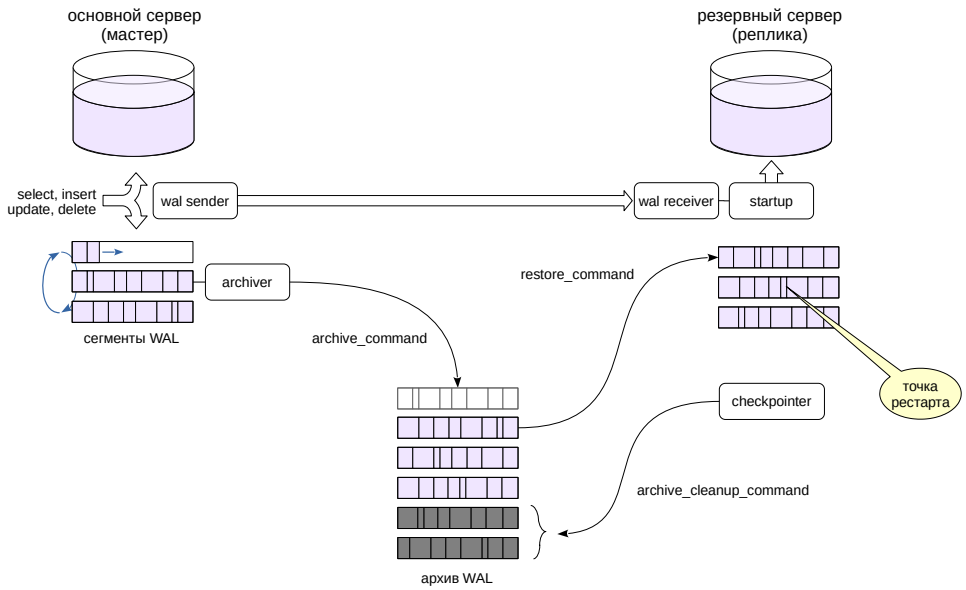
Если реплика по каким-то причинам не сможет получить очередную журнальную запись по протоколу репликации (например, из-за обрыва связи), она попробует прочитать ее из архива с помощью команды, заданной в параметре *restore_command*. При восстановлении связи реплика снова автоматически переключится на использование потоковой репликации.

В принципе, репликация может работать и с одним только архивом, без потоковой репликации. Но в этом случае:

- реплика вынужденно отстает от мастера на время заполнения сегмента;

- мастер ничего не знает о существовании реплики, что в некоторых случаях может привести к проблемам.

Очистка архива

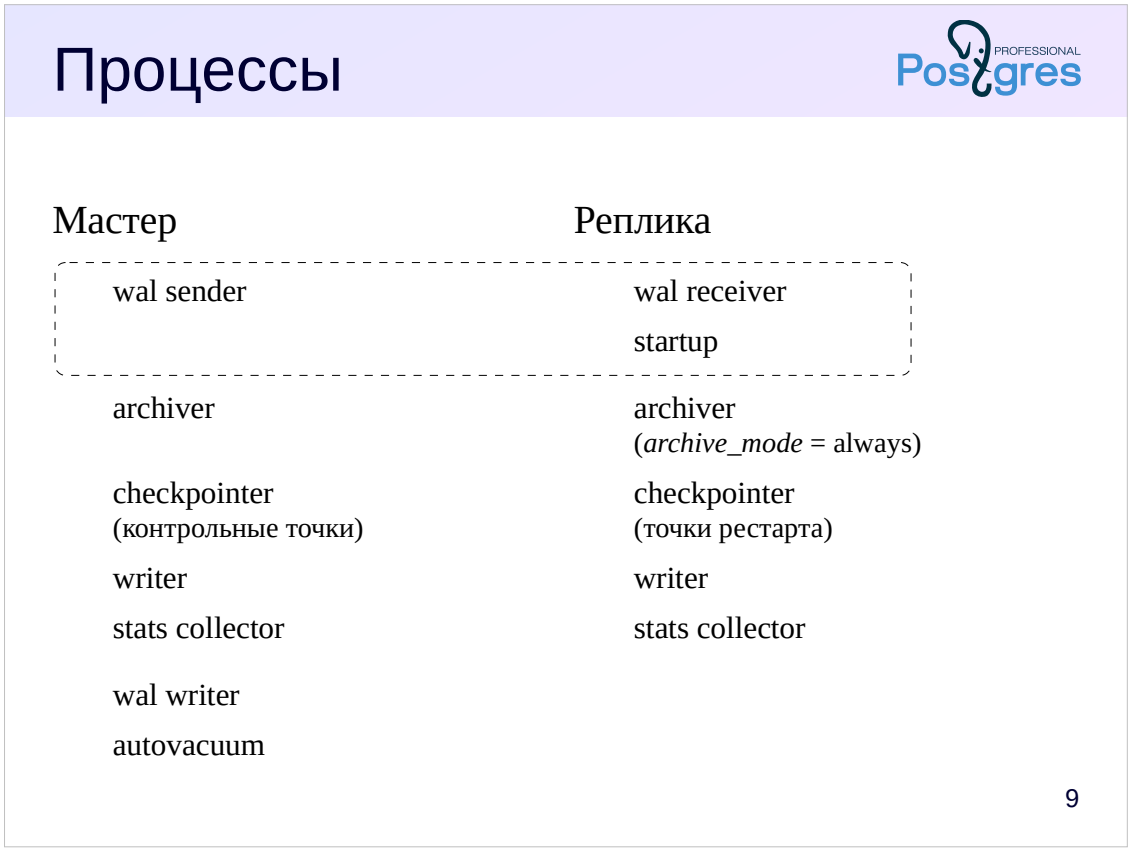


Если архив используется только для синхронизации одной реплики, можно настроить его автоматическую очистку, для этого в параметре *archive_cleanup_command* задается команда, которая будет выполняться по окончании каждой точки рестарта подобно тому, как после каждой контрольной точки удаляются файлы журнала. В команде удобно вызывать утилиту pg_archivecleanup:

```
archive_cleanup_command = 'pg_archivecleanup
/path/to/archive %r'
```

https://postgrespro.ru/docs/postgresql/13/runtime-config-wal#RUNTIME-CONFIG-WAL-ARCHIVE-RECOVERY

https://postgrespro.ru/docs/postgresql/13/pgarchivecleanup



На мастере и на реплике выполняется разный набор процессов. Кроме уже рассмотренных wal sender (на мастере) и wal receiver и startup (на реплике), работают также другие процессы.

Общие процессы:

- **background writer** — записывает грязные страницы из буферного кеша на диск (на реплике, для того, чтобы применить журнальную запись, страницы точно так же читаются в буферный кеш и затем записываются в фоновом режиме или при вытеснении).

- **stats collector** — статистика не передается в журнале, а собирается на реплике отдельно.

- **checkpointer** — на реплике при получении журнальной записи о контрольной точке выполняется похожая процедура — точка рестарта. Если в процессе восстановления случится сбой, реплика сможет продолжить с последней точки рестарта, а не с самого начала.

Процессы только на мастере:

- **wal writer** — реплика не создает журналы упреждающей записи, а только получает их с мастера.

- **autovacuum launcher/worker** — реплика не выполняет очистку, ее выполнение на мастере передается реплике через журнал.

Есть также процесс, наличие которого зависит от настройки:

- **archiver** — при значении *archive_mode* = on выполняется только на мастере, а при *archive_mode* = always также и на реплике (подробнее рассматривается в теме «Переключение на реплику»).

Процессы реплики

Посмотрим на процессы реплики.

```
student$ ps -o pid,command --ppid `sudo head -n 1 /var/lib/postgresql/13/beta/postmaster.pid`
```

```
    PID COMMAND
  79945 postgres: 13/beta: startup recovering 000000010000000000000005
  79946 postgres: 13/beta: checkpointer
  79947 postgres: 13/beta: background writer
  79948 postgres: 13/beta: stats collector
  79949 postgres: 13/beta: walreceiver streaming 0/5000060
```

Процесс wal receiver принимает поток журнальных записей, процесс startup применяет изменения.

Процессы wal writer и autovacuum launcher отсутствуют.

И сравним с процессами мастера.

```
student$ ps -o pid,command --ppid `sudo head -n 1 /var/lib/postgresql/13/alpha/postmaster.pid`
```

```
    PID COMMAND
  79517 postgres: 13/alpha: checkpointer
  79518 postgres: 13/alpha: background writer
  79519 postgres: 13/alpha: walwriter
  79520 postgres: 13/alpha: autovacuum launcher
  79521 postgres: 13/alpha: stats collector
  79522 postgres: 13/alpha: logical replication launcher
  79557 postgres: 13/alpha: student student [local] idle
  79950 postgres: 13/alpha: walsender student [local] streaming 0/5000060
```

Здесь добавился процесс wal sender, обслуживающий подключение по протоколу репликации.

Использование реплики

Допускаются

- запросы на чтение данных (select, copy to, курсоры)
- установка параметров сервера (set, reset)
- управление транзакциями (begin, commit, rollback...)
- создание резервной копии (pg_basebackup)

Не допускаются

- любые изменения (insert, update, delete, truncate, nextval...)
- блокировки, предполагающие изменение (select for update...)
- команды DDL (create, drop...), в том числе создание временных таблиц
- команды сопровождения (vacuum, analyze, reindex...)
- управление доступом (grant, revoke...)

Как мы уже говорили, на реплике не допускается любое изменение данных. Более точно, не допускаются:

- изменения таблиц, материализованных представлений, последовательностей;

- блокировки (так как для этого требуется изменение страниц данных);

- команды DDL, включая создание временных таблиц;

- такие команды, как vacuum и analyze;

- команды управления доступом.

При этом реплика может выполнять запросы на чтение данных, если установлен параметр *hot_standby* = on. Также будет работать установка параметров сервера и команды управления транзакциями — например, можно начать (читающую) транзакцию с нужным уровнем изоляции.

Кроме того, реплику можно использовать и для изготовления резервных копий.

Использование реплики

Выполним несколько команд на мастере:

```
α=> CREATE DATABASE replica_physical;

CREATE DATABASE

α=> \c replica_physical

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

α=> CREATE TABLE test(s text);

CREATE TABLE

α=> INSERT INTO test VALUES ('Привет, мир!');

INSERT 0 1
```

Проверим реплику:

```
student$ psql -p 5433 -d replica_physical

  β=> SELECT * FROM test;

        s
--------------
 Привет, мир!
(1 row)
```

При этом изменения на реплике не допускаются:

```
  β=> INSERT INTO test VALUES ('Replica');

  ERROR:  cannot execute INSERT in a read-only transaction
```

Вообще реплику от мастера можно отличить с помощью функции:

```
  β=> SELECT pg_is_in_recovery();

   pg_is_in_recovery
--------------------
 t
(1 row)
```

Ограничения

Изоляция и многоверсионность

- большое число точек сохранения задерживает выполнение запросов
- не поддерживается уровень изоляции serializable

Триггеры и блокировки

- триггеры и пользовательские блокировки (advisory locks) не работают

Резервное копирование с реплики

- параметр *full_page_writes* должен быть заранее включен на мастере
- задержки из-за ожидания контрольной точки и переключения сегментов

Изменение табличных пространств

- каталоги на реплике нужно создавать заранее, так как прав пользователя ОС postgres может не хватить

У репликации есть ряд ограничений и особенностей.

Если на мастере выполняется транзакция с числом вложенных транзакций > 64 (то есть активно используются точки сохранения), создание снимка на реплике будет приостановлено — это означает невозможность некоторое время выполнять запросы.

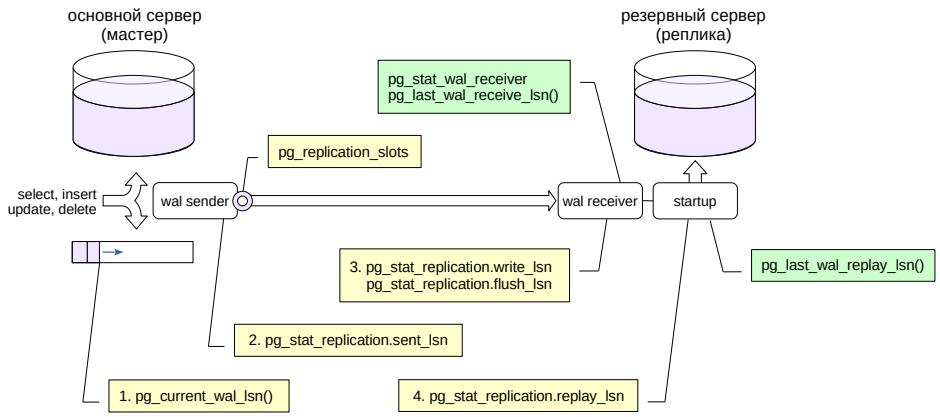
Не поддерживается уровень изоляции serializable на реплике.

Триггеры не будут срабатывать на реплике (они выполнились на мастере и реплицируется уже результат их работы). Не будут работать и рекомендательные (advisory) блокировки.

При выполнении резервного копирования с реплики нет технической возможности полноценно управлять мастером, в частности, вызывать контрольные точки и включать параметр *full_page_writes*. Поэтому параметр нужно включить на мастере заранее, а контрольную точку приходится ждать (что может существенно задержать процесс).

Создание табличных пространств реплицируется, но, если у пользователя ОС, под которым работает СУБД, недостаточно прав для создания каталога, произойдет ошибка. Лучше всего сначала создать каталоги на мастере и на всех репликах, а уже затем выполнять команду CREATE TABLESPACE.

Мониторинг



log_replication_commands = on

1 – 2 = задержка записи в поток на мастере
2 – 3 = задержка получения данных репликой
3 – 4 = задержка применения данных на реплике

14

В ходе репликации возможны проблемы, о которых должен предупредить заранее настроенный мониторинг. Основной информацией являются позиции в журнале предзаписи. Можно выделить четыре важные точки:

1. появление журнальной записи на мастере;

2. трансляция записи процессом wal sender;

3. получение записи процессом wal receiver;

4. применение полученной записи процессом startup.

Все эти точки обычно отслеживаются на мастере. Первую точку дает функция pg_current_wal_lsn(), остальные — представление pg_stat_replication. Реплика передает мастеру статус репликации при каждой записи на диск, но как минимум раз в *wal_receiver_status_interval* секунд (по умолчанию — 10 секунд).

Если используется слот репликации, то информацию о нем можно получить из представления pg_replication_slots.

Данные о ходе репликации можно получить и на реплике в представлении pg_stat_wal_receiver и с помощью функций pg_last_wal_receive_lsn() и pg_last_wal_replay_lsn().

Параметр *log_replication_commands* включает вывод в журнал сообщений информации о выполняющихся командах протокола репликации.

Мониторинг репликации

Состояние репликации можно смотреть в специальном представлении на мастере. Чтобы пользователь получил доступ к этой информации, ему должна быть выдана роль pg_read_all_stats (или он должен быть суперпользователем).

```
α=> \du student
```

```
                            List of roles
 Role name |             Attributes              |      Member of
-----------+-------------------------------------+---------------------
 student   | Create role, Create DB, Replication | {pg_read_all_stats}
```

```
α=> SELECT * FROM pg_stat_replication \gx
```

```
-[ RECORD 1 ]----+------------------------------
pid              | 79950
usesysid         | 16384
usename          | student
application_name | 13/beta
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2022-11-01 22:56:34.526874+03
backend_xmin     |
state            | streaming
sent_lsn         | 0/501BEC0
write_lsn        | 0/501BEC0
flush_lsn        | 0/501BEC0
replay_lsn       | 0/501BEC0
write_lag        | 00:00:00.000345
flush_lag        | 00:00:00.03614
replay_lag       | 00:00:01.387579
sync_priority    | 0
sync_state       | async
reply_time       | 2022-11-01 22:56:40.99707+03
```

Обратите внимание на поля *_lsn (и *_lag) — они показывают отставание реплики на разных этапах. Сейчас все позиции совпадают, отставание нулевое.

Теперь вставим в таблицу большое количество строк, чтобы увидеть репликацию в процессе работы.

```
α=> INSERT INTO test SELECT 'Just a line' FROM generate_series(1,1000000);
```

```
INSERT 0 1000000
```

```
α=> SELECT *, pg_current_wal_lsn() from pg_stat_replication \gx
```

```
-[ RECORD 1 ]------+------------------------------
pid                | 79950
usesysid           | 16384
usename            | student
application_name   | 13/beta
client_addr        |
client_hostname    |
client_port        | -1
backend_start      | 2022-11-01 22:56:34.526874+03
backend_xmin       |
state              | streaming
sent_lsn           | 0/8A00000
write_lsn          | 0/89A0000
flush_lsn          | 0/8980000
replay_lsn         | 0/893FFF8
write_lag          | 00:00:01.458638
flush_lag          | 00:00:01.463649
replay_lag         | 00:00:01.473671
sync_priority      | 0
sync_state         | async
reply_time         | 2022-11-01 22:56:49.042627+03
pg_current_wal_lsn | 0/94F9CC8
```

Видно, что возникла небольшая задержка. Однако не следует рассчитывать на точность выше чем полсекунды, так как статистика обновляется примерно с такой частотой.

Проверим реплику:

```
β=> SELECT count(*) FROM test;
```

```
  count
---------
 1000001
(1 row)
```

Все строки успешно доехали.

И еще раз проверим состояние репликации:

```
α=> SELECT *, pg_current_wal_lsn() from pg_stat_replication \gx
```

```
-[ RECORD 1 ]------+------------------------------
pid                | 79950
usesysid           | 16384
usename            | student
application_name   | 13/beta
client_addr        |
client_hostname    |
client_port        | -1
backend_start      | 2022-11-01 22:56:34.526874+03
backend_xmin       |
state              | streaming
sent_lsn           | 0/94F9CC8
write_lsn          | 0/94F9CC8
flush_lsn          | 0/94F9CC8
replay_lsn         | 0/94F9CC8
write_lag          | 00:00:00.891145
flush_lag          | 00:00:00.89933
replay_lag         | 00:00:00.931834
sync_priority      | 0
sync_state         | async
reply_time         | 2022-11-01 22:56:49.720745+03
pg_current_wal_lsn | 0/94F9CC8
```

Все позиции выровнялись.

Режимы репликации

Асинхронный

synchronous_commit = off
local

Синхронный

фиксация с ожиданием подтверждения от синхронной реплики

обеспечивает надежность, но не согласованность

synchronous_commit = remote_write
on
remote_apply

synchronous_standby_names = FIRST *N* (*список реплик*)
ANY *N* (*список реплик*)

Как известно, фиксация транзакций может выполняться в двух режимах. В более быстром асинхронном режиме есть шанс потерять уже зафиксированные изменения в случае сбоя. В синхронном режиме команда COMMIT не завершается, пока запись не дойдет до диска. При наличии нескольких серверов надежность можно еще более увеличить, дожидаясь подтверждения от реплики о получении записи, тогда данные не пропадут даже при выходе из строя носителя.

Синхронная репликация включается тем же параметром *synchronous_commit,* что и синхронная фиксация. Разные значения этого параметра позволяют управлять уровнем надежности. При значении **remote_write** мастер ожидает подтверждения о получении записи (остается шанс потерять изменения, если на реплике случится сбой и она не успеет записать данные на диск).

При значении **on** мастер ожидает подтверждения о попадании журнальной записи на диск реплики (это надежный режим; однако приложение, обратившись к реплике, может не увидеть изменений).

Наконец, значение **remote_apply** заставляет мастер дождаться применения записи на реплике. Каждый следующий режим вызывает все большие задержки.

Мастер может синхронизироваться как с одной, так и с несколькими репликами. Можно указать список реплик в порядке приоритета или организовать синхронизацию на основе *кворума*, при которой мастер дожидается подтверждения от любых *N* реплик из числа доступных.

Возможные проблемы

Мастер может удалить файл журнала, нужный реплике

если реплика не успеет его получить
(например, будет остановлена на некоторое время)

Решение 1. Слот репликации

вводит зависимость мастера от реплики, требует мониторинга

Решение 2. Архив журнала предзаписи

позволяет обойтись без слота

При репликации возможен ряд сложностей, которые можно решать разными способами.

Сервер PostgreSQL периодически удаляет файлы журнала, которые не требуются для восстановления. При этом мастер может удалить файл, который содержит данные, еще не переданные реплике.

Это не проблема, если используется архив журналов предзаписи, поскольку реплика, получив отказ по протоколу репликации, прочитает необходимые ей данные из архива, а затем, «догнав» мастер, снова переключится на получение данных из потока.

Но если архива нет, реплика не сможет продолжать восстановление и ее придется пересоздавать из новой резервной копии. Чтобы этого избежать, можно использовать слот репликации. Однако надо понимать, что создание слота ставит мастер в зависимость от реплики: если реплика не будет получать журнальные записи, они будут накапливаться на мастере и свободное пространство рано или поздно закончится. Поэтому слот — еще одна точка, которую необходимо включать в мониторинг.

Влияние слота репликации

Остановим реплику.

```
student$ sudo pg_ctlcluster 13 beta stop
```

Ограничим размер WAL двумя сегментами.

```
α=> \c - postgres
```

```
You are now connected to database "replica_physical" as user "postgres".
```

```
α=> ALTER SYSTEM SET min_wal_size='32MB';
```

```
ALTER SYSTEM
```

```
α=> ALTER SYSTEM SET max_wal_size='32MB';
```

```
ALTER SYSTEM
```

```
α=> SELECT pg_reload_conf();
```

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

Слот неактивен, но помнит номер последней записи, полученной репликой:

```
α=> SELECT active, restart_lsn, wal_status FROM pg_replication_slots \gx
```

```
-[ RECORD 1 ]----------
active      | f
restart_lsn | 0/94F9CC8
wal_status  | reserved
```

Вставим еще строки в таблицу.

```
α=> INSERT INTO test SELECT 'Just a line' FROM generate_series(1,1000000);
```

```
INSERT 0 1000000
```

Номер LSN в слоте не изменился:

```
α=> SELECT active, restart_lsn, wal_status FROM pg_replication_slots \gx
```

```
-[ RECORD 1 ]----------
active      | f
restart_lsn | 0/94F9CC8
wal_status  | extended
```

Выполним контрольную точку, которая должна очистить журнал, поскольку его размер превышает допустимый.

```
α=> CHECKPOINT;
```

```
CHECKPOINT
```

Каков теперь размер журнала?

```
α=> SELECT pg_size_pretty(sum(size)) FROM pg_ls_waldir();
```

```
 pg_size_pretty
----------------
 80 MB
(1 row)
```

Контрольная точка не удалила сегменты — их удерживает слот, несмотря на превышение лимита. Если оставить слот без присмотра, дисковое пространство может быть исчерпано.

Скорее всего, бесперебойная работа основного сервера важнее, чем синхронизация реплики. Чтобы предотвратить разрастание журнала, можно ограничить объем, удерживаемый слотом:

```
α=> ALTER SYSTEM SET max_slot_wal_keep_size='16MB';
```

```
ALTER SYSTEM
```

```
α=> SELECT pg_reload_conf();
```

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

```
α=> CHECKPOINT;
```

```
CHECKPOINT
```

Теперь журнал очищается контрольной точкой:

```
α=> SELECT pg_size_pretty(sum(size)) FROM pg_ls_waldir();
```

```
 pg_size_pretty
----------------
 32 MB
(1 row)
```

Но слот уже не обеспечивает наличие записей WAL (wal_status=lost):

```
α=> SELECT active, restart_lsn, wal_status FROM pg_replication_slots \gx
```

```
-[ RECORD 1 ]-----
active      | f
restart_lsn |
wal_status  | lost
```

А реплика не может синхронизироваться:

```
student$ sudo pg_ctlcluster 13 beta start
```

```
student$ sudo tail -n 2 /var/log/postgresql/postgresql-13-beta.log
```

```
2022-11-01 22:57:01.802 MSK [81062] LOG:  started streaming WAL from primary at 0/9000000 on timeline 1
2022-11-01 22:57:01.802 MSK [81062] FATAL:  could not receive data from WAL stream: ERROR:  requested WAL segment 000000010000000000000009 has already been removed
```

Для синхронизации реплики в таких случаях можно воспользоваться архивом, задав параметр restore_command. Если это невозможно, придется заново настроить репликацию, повторив формирование базовой копии.

Возможные проблемы

Конфликты: записи WAL, несовместимые с запросами на реплике

- очистка удаляет версии строк, нужные запросам на реплике
- исключительные блокировки объектов
- исключительные блокировки буферов
- реже: взаимоблокировки, DROP TABLESPACE, DROP DATABASE – запрос сразу отменяется

Мониторинг

- pg_stat_database_conflicts

Вторая проблема возникает, если реплика используется для выполнения запросов. Мастер может выполнить действие, несовместимое с запросом на реплике:

- очистка может удалить версию строки, которая используется запросом, или потребовать блокировку, несовместимую с запросом;

- может прийти запись об эксклюзивной блокировке объекта (например, при удалении таблицы);

Возникновение таких конфликтов можно отслеживать с помощью представления pg_stat_database_conflicts.

Конфликты с запросами на реплике, вызванные редкими причинами, такими как взаимоблокировки, удаление базы данных или табличного пространства, разрешаются путем немедленной отмены запроса.

Возможные проблемы

Решение 0: избегать конфликтов

не выполнять команды, вызывающие блокировки

отменить усечение файла в конце очистки: параметр хранения *vacuum_truncate* = off

Решение 1: откладывать применение конфликтующих записей

max_standby_streaming_delay

max_standby_archive_delay

Один из путей борьбы с конфликтами — избегать действий на мастере, приводящих к исключительным блокировкам. Например, можно не выполнять команды DROP TABLE, TRUNCATE, LOCK, DROP INDEX, DROP TRIGGER и т. п. или выполнять их в то время, когда на реплике заведомо нет выполняющихся запросов.

Некоторых неявных блокировок можно избежать, отключая соответствующую функциональность на мастере. Например, если в результате очистки освободились страницы в конце файла, при уменьшении его размера устанавливается исключительная блокировка таблицы. Но можно отменить усечение файла, задав параметр хранения таблицы *vacuum_truncate* = off. Однако отменять или надолго откладывать очистку нельзя, поэтому избежать конфликта в случае очистки нужных запросу версий строк таким образом не получится.

Другой способ — откладывать применение полученных несовместимых журнальных записей на реплике. Этот способ работает и при потоковой репликации, и при использовании архива; он откладывает записи, относящие и к очистке, и к эксклюзивным блокировкам. В параметрах устанавливается максимальная задержка применения (от времени получения записи). Если к этому моменту конфликтующий запрос не успеет завершиться, он будет прерван.

Возможные проблемы

Решение 2: обратная связь по протоколу репликации

- снимок на реплике предотвращает удаление версий строк так же, как и локальный
- *hot_standby_feedback* = on
- только для конфликта очистки с запросом на реплике

Мониторинг

- pg_stat_replication.backend_xmin
- pg_replication_slots.xmin

Еще один путь решения — откладывать несовместимые действия на мастере. Для этого включается *обратная связь.*

Этот способ работает только для очистки (эксклюзивную блокировку отложить не получится) и только в случае потоковой репликации (если реплика переключается на получение записей из архива, она фактически перестает существовать для мастера).

Каждый снимок данных содержит так называемый горизонт событий — это значение счетчика транзакций, по которому проходит граница между ненужными и нужными версиями строк. Минимальный горизонт по всем снимкам реплики определяет возможность очистки. Подробности обсуждаются в теме «Снимки данных» курса DBA2.

Благодаря обратной связи, мастер понимает, какие версии строк нужны реплике. В таком случае (с точки зрения очистки) запрос на реплике ничем не отличается от запроса на самом мастере.

Обратная связь позволяет мониторить горизонт событий на стороне мастера. Если слот репликации не используется, горизонт виден в поле pg_stat_replication.backend_xmin, а при использовании слота — в pg_replication_slots.xmin. Нужно учитывать, что при включенной обратной связи слот, даже неактивный, будет задерживать очистку, что может привести к увеличению размеров файлов данных. Этот эффект будет исследован в практике.

Итоги

Механизм репликации основан на передаче журнальных записей на реплику и их применении

- основной режим: потоковая репликация
- может быть поддержана архивом журнальных записей

Реплика — точная копия мастера

- не генерирует собственные записи WAL, доступна только для чтения

Сложный механизм, требующий настройки и мониторинга

- настройка существенно зависит от решаемых задач

Практика

1. Настройте физическую потоковую репликацию между двумя серверами в синхронном режиме, без обратной связи.
2. Проверьте работу репликации. Убедитесь, что при остановленной реплике фиксация не завершается.
3. Воспроизведите ситуацию, при которой запрос на реплике прерывается из-за очистки версий строк на мастере.
4. Запретите откладывать применение конфликтующих изменений; проверьте, что запрос отменяется сразу.
5. Включите обратную связь и убедитесь, что очистка на мастере не прерывает выполнение запроса.
6. Остановите реплику и убедитесь, что слот препятствует очистке. Удалите слот, чтобы очистка возобновилась.

1. Для этого на мастере установите параметры:

- *synchronous_commit* = on,

- *synchronous_standby_names* = 'beta',

а на реплике

- *cluster_name* = beta.

3. Для этого на реплике можно начать транзакцию с уровнем изоляции repeatable read и периодически выполнять запросы к таблице, а на мастере изменить эту таблицу и выполнить очистку.

Учтите, что параметр *max_standby_streaming_delay* по умолчанию имеет значение 30 секунд.

4. Установите параметр *max_standby_streaming_delay* в ноль.

5. Установите на реплике параметр *hot_standby_feedback* = on. Максимальный интервал оповещений устанавливается в параметре *wal_receiver_status_interval* (по умолчанию 10 секунд).

6. Не забудьте отменить *synchronous_commit*, иначе при остановленной реплике не удастся зафиксировать транзакцию.

Физическая потоковая репликация в синхронном режиме

```
α=> CREATE DATABASE replica_physical;
```

```
CREATE DATABASE
```

Создаем и запускаем реплику точно так же, как в демонстрации.

```
α=> \c - postgres
```

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

Резервная копия:

```
student$ pg_basebackup --pgdata=/home/student/backup -R --slot=replica --create-slot
```

В postgresql.auto.conf добавляем параметр cluster_name, чтобы основной сервер мог идентифицировать реплику:

```
student$ echo 'cluster_name=beta' | sudo tee -a /home/student/backup/postgresql.auto.conf
```

```
cluster_name=beta
```

```
student$ sudo cat /home/student/backup/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
primary_conninfo = 'user=student passfile=''/home/student/.pgpass'' channel_binding=prefer host=''/var/run/postgresql'' port=5432 sslmode=prefer sslcompression=0 sslsni=1 ssl_min_proto
primary_slot_name = 'replica'
cluster_name=beta
```

Другой вариант - добавить атрибут application_name в строку соединения:

primary_conninfo='user=student port=5432 application_name=beta'

Выкладываем в PGDATA и запускаем:

```
student$ sudo pg_ctlcluster 13 beta status
```

```
Error: /var/lib/postgresql/13/beta is not accessible or does not exist
```

```
student$ sudo rm -rf /var/lib/postgresql/13/beta
```

```
student$ sudo mv /home/student/backup /var/lib/postgresql/13/beta
```

```
student$ sudo chown -R postgres:postgres /var/lib/postgresql/13/beta
```

```
student$ sudo pg_ctlcluster 13 beta start
```

Теперь включаем на мастере режим синхронной репликации. Сразу переключимся на суперпользовательскую роль.

```
student$ psql -d replica_physical -U postgres
```

```
α=> ALTER SYSTEM SET synchronous_commit = on;
```

```
ALTER SYSTEM
```

```
α=> ALTER SYSTEM SET synchronous_standby_names = beta;
```

```
ALTER SYSTEM
```

Если все реплики синхронные, можно задавать synchronous_standby_names = *

```
student$ sudo pg_ctlcluster 13 alpha reload
```

Проверка физической репликации

```
α=> SELECT * FROM pg_stat_replication \gx
```

```
-[ RECORD 1 ]----+------------------------------
pid              | 92364
usesysid         | 16384
usename          | student
application_name | beta
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2022-11-01 23:03:54.353219+03
backend_xmin     |
state            | streaming
sent_lsn         | 0/5000640
write_lsn        | 0/5000640
flush_lsn        | 0/5000640
replay_lsn       | 0/5000640
write_lag        | 00:00:00.000368
flush_lag        | 00:00:00.006711
replay_lag       | 00:00:00.006793
sync_priority    | 1
sync_state       | sync
reply_time       | 2022-11-01 23:03:56.631926+03
```

sync_state: sync говорит о том, что репликация работает в синхронном режиме.

application_name: beta - имя, под которым мастер знает реплику.

Остановим реплику.

```
student$ sudo pg_ctlcluster 13 beta stop
```

```
α=> \c - student
```

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

```
α=> BEGIN;
```

```
BEGIN
```

```
α=> CREATE TABLE test(id integer);
```

```
CREATE TABLE
```

```
α=> COMMIT;
```

Фиксация ждет появления синхронной реплики...

```
student$ sudo pg_ctlcluster 13 beta start
```

После старта реплики фиксация завершается.

```
COMMIT
```

Отмена запроса из-за очистки на мастере

```
α=> INSERT INTO test VALUES (1);
```

```
INSERT 0 1
```

```
student$ psql -p 5433 -d replica_physical
```

```
β=> SHOW max_standby_streaming_delay;
```

```
 max_standby_streaming_delay
-----------------------------
 30s
(1 row)
```

Начинаем транзакцию с уровнем изоляции repeatable read: первый оператор создаст снимок данных, который будет использоваться всеми последующими операторами этой транзакции.

```
β=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
β=> SELECT * FROM test;
```

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

На мастере изменяем единственную строку и выполняем очистку. При этом первая версия строки будет удалена.

```
α=> UPDATE test SET id = 2;
```

```
UPDATE 1
```

```
α=> VACUUM test;
```

```
VACUUM
```

Через 20 секунд запрос на реплике еще сработает — реплика задерживает применение конфликтующей журнальной записи:

```
β=> SELECT * FROM test;
```

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

Через 30 секунд такой же запрос уже будет аварийно прерван, поскольку версия строки, входящая в снимок, больше не существует.

```
β=> SELECT * FROM test;
```

```
FATAL:  terminating connection due to conflict with recovery
DETAIL:  User query might have needed to see row versions that must be removed.
HINT:  In a moment you should be able to reconnect to the database and repeat your command.
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
connection to server was lost
```

Запрет откладывания применения конфликтующей записи

Выставим задержку применения конфликтующих изменений в ноль.

```
student$ psql -p 5433 -d replica_physical -U postgres
```

```
β=> ALTER SYSTEM SET max_standby_streaming_delay = 0;
```

```
ALTER SYSTEM
```

```
β=> \c - student
```

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

```
student$ sudo pg_ctlcluster 13 beta reload
```

Снова начинаем транзакцию...

```
β=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
β=> SELECT * FROM test;
```

```
 id
----
  2
(1 row)
```

На мастере изменяем строку и выполняем очистку...

```
α=> UPDATE test SET id = 3;
```

```
UPDATE 1
```

```
α=> VACUUM test;
```

```
VACUUM
```

Запрос на реплике прерывается тут же:

```
β=> SELECT * FROM test;
```

```
FATAL:  terminating connection due to conflict with recovery
DETAIL:  User query might have needed to see row versions that must be removed.
HINT:  In a moment you should be able to reconnect to the database and repeat your command.
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
connection to server was lost
```

Обратная связь

Установим обратную связь и, чтобы не ждать долго, небольшой интервал оповещений.

```
student$ psql -p 5433 -d replica_physical -U postgres
```

```
β=> ALTER SYSTEM SET hot_standby_feedback = on;
```

```
ALTER SYSTEM
```

```
β=> ALTER SYSTEM SET wal_receiver_status_interval = '1s';
```

```
ALTER SYSTEM
```

```
β=> \c - student
```

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

```
student$ sudo pg_ctlcluster 13 beta reload
```

Снова начинаем транзакцию...

```
β=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
β=> SELECT * FROM test;
```

```
 id
----
  3
(1 row)
```

На мастере изменяем строку и выполняем очистку...

```
α=> UPDATE test SET id = 4;
```

```
UPDATE 1
```

```
α=> VACUUM VERBOSE test;
```

```
INFO:  vacuuming "public.test"
INFO:  "test": found 0 removable, 2 nonremovable row versions in 1 out of 1 pages
DETAIL:  1 dead row versions cannot be removed yet, oldest xmin: 492
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Благодаря обратной связи, очистка не может удалить старую версию строки (found 0 removable, 2 nonremovable row versions).

```
β=> SELECT * FROM test;
```

```
 id
----
  3
(1 row)
```

```
β=> COMMIT;
```

```
COMMIT
```

```
α=> VACUUM VERBOSE test;
```

```
INFO:  vacuuming "public.test"
INFO:  "test": found 1 removable, 1 nonremovable row versions in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 493
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

После завершения транзакции на реплике мастер может выполнить очистку (found 1 removable...). Журнальные записи реплицируются, но ничего не поломают.

Теперь отменим синхронизацию и остановим реплику:

```
α=> \c - postgres
```

```
You are now connected to database "replica_physical" as user "postgres".
```

```
α=> ALTER SYSTEM RESET synchronous_standby_names;
```

```
ALTER SYSTEM
```

```
α=> \c - student
```

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

```
student$ sudo pg_ctlcluster 13 alpha reload
```

```
student$ sudo pg_ctlcluster 13 beta stop
```

Еще раз изменим строку таблицы:

```
α=> UPDATE test SET id = 5;
```

```
UPDATE 1
```

Слот неактивен, но помнит минимальный xmin снимков реплики:

```
α=> SELECT active, xmin FROM pg_replication_slots;
```

```
 active | xmin
--------+------
 f      |  493
(1 row)
```

Это значение xmin меньше, чем минимальный xmin локальных снимков:

```
α=> SELECT min(backend_xmin::text::numeric) FROM pg_stat_activity;
```

```
 min
-----
 494
(1 row)
```

Поэтому слот будет задерживать очистку (1 dead row versions cannot be removed yet, oldest xmin: ...).

```
α=> VACUUM VERBOSE test;
```

```
INFO:  vacuuming "public.test"
INFO:  "test": found 0 removable, 2 nonremovable row versions in 1 out of 1 pages
DETAIL:  1 dead row versions cannot be removed yet, oldest xmin: 493
There were 0 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Удалим слот, теперь очистка сработает.

```
α=> SELECT pg_drop_replication_slot('replica');
```

```
 pg_drop_replication_slot
--------------------------

(1 row)
```

```
α=> VACUUM VERBOSE test;
```

```
INFO:  vacuuming "public.test"
INFO:  "test": found 1 removable, 1 nonremovable row versions in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 494
There were 1 unused item identifiers.
Skipped 0 pages due to buffer pins, 0 frozen pages.
0 pages are entirely empty.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```