

Репликация

Обзор физической репликации



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

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.

Задачи и виды репликации

Физическая репликация

Уровни журнала

Варианты использования реплики

Переключение на реплику

Синхронизация данных на разных серверах

Основные задачи

отказоустойчивость, высокая доступность
масштабируемость

Физическая репликация

синхронизация на уровне страниц и версий строк

Логическая репликация

синхронизация на уровне строк таблиц

Одиночный сервер, управляющий базами данных, может не удовлетворять предъявляемым требованиям.

Один сервер — возможная точка отказа. Два (или больше) серверов позволяют сохранить доступность системы в случае сбоя (отказоустойчивость) или — более широко — в любом случае, например, при проведении плановых работ (высокая доступность).

Один сервер может не справляться с нагрузкой. Нарастивать ресурсы сервера может оказаться невыгодно или вообще невозможно.

Но можно распределить нагрузку на несколько серверов (масштабирование).

Информационные системы могут использовать общие данные.

Таким образом, речь идет о том, чтобы иметь несколько серверов, работающих с одними и теми же данными. Под репликацией понимается процесс синхронизации этих данных.

В зависимости от уровня, на котором происходит синхронизация, различают *физическую* и *логическую* репликацию: при физической репликации синхронизируются изменения, произошедшие в страницах данных и статусах транзакций, а при логической — изменения строк таблиц.

Механизм

один сервер транслирует журнальные записи на другой сервер,
и тот проигрывает полученные записи

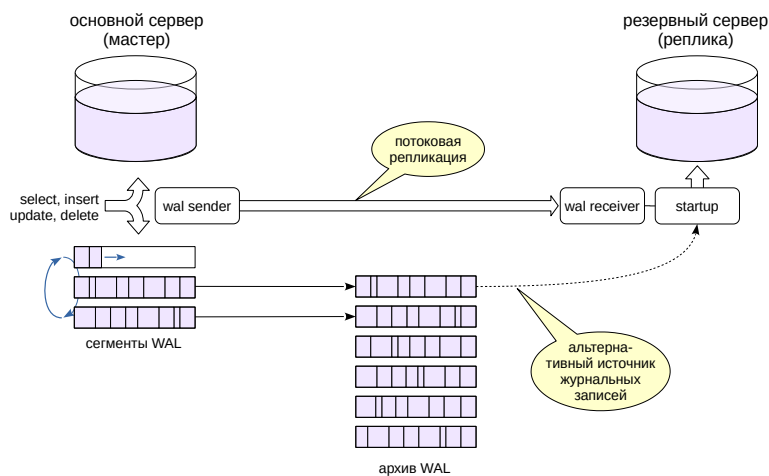
Особенности

мастер-реплика: поток данных только в одну сторону
требуется двоичная совместимость серверов
возможна репликация только всего кластера

Механизм физической репликации состоит в том, что один сервер передает журнальные записи на другой сервер, а тот их проигрывает — как при восстановлении после сбоя.

При физической репликации серверы имеют назначенные роли: мастер и реплика. Мастер передает на реплику свои журнальные записи в виде файлов или потока записей; реплика применяет эти записи к своим файлам данных. Применение происходит чисто механически, без «понимания смысла» изменений, поэтому важна двоичная совместимость между серверами (одинаковые платформы и основные версии PostgreSQL). Поскольку журнал общий для всего кластера, то и реплицировать можно только весь кластер целиком.

Физическая репликация



5

Чтобы организовать репликацию между двумя серверами, мы создаем реплику из физической резервной копии основного сервера. При обычном восстановлении из такой резервной копии получается новый независимый сервер. А если включить репликацию, ведомый сервер начинает работать в режиме *постоянного восстановления*: он все время применяет новые журнальные записи, приходящие с основного сервера (этим занимается процесс startup). Таким образом, реплика постоянно поддерживается в почти актуальном состоянии.

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

В этом случае реплика (процесс walreceiver) подключается к мастеру (процесс walsender) по протоколу репликации и получает поток записей WAL. За счет этого при потоковой репликации отставание реплики сведено к минимуму или даже к нулю при синхронном режиме.

Если в системе настроено непрерывное архивирование, то возможна *файловая репликация*. В этом случае реплика будет заметно отставать, поскольку файловый архив пополняется только при переключении сегмента WAL.

На практике файловую репликацию используют как дополнение к потоковой. Если реплика не может получить очередную журнальную запись по протоколу репликации, она будет пробовать прочитать соответствующий файл из архива сегментов WAL.

<https://postgrespro.ru/docs/postgresql/16/high-availability>

Параметр *wal_level*

minimal	<	replica
восстановление после сбоя		восстановление после сбоя восстановление из резервной копии, репликация

Поскольку на реплику попадает только та информация, которая содержится в журнале, в журнал должны записываться все необходимые для синхронизации данные.

Состав информации, попадающей в журнал, регулируется параметром *wal_level*.

До версии PostgreSQL 10 уровнем по умолчанию был **minimal**, гарантирующий только восстановление после сбоя. На таком уровне репликация работать не может, поскольку для обеспечения надежности в этом случае часть изменений сразу записывается на энерго-независимый носитель и не попадает в журнал.

В версиях 10+ уровень по умолчанию — **replica**. На этом уровне в журнал записываются *все* изменения данных, что позволяет восстанавливать систему из горячих резервных копий, сделанных утилитой *pg_basebackup*, а также использовать физическую репликацию.

Поскольку резервное копирование и репликация — востребованные задачи, уровень по умолчанию и был изменен в пользу *replica*.

Настройка физической репликации

Посмотрим, как организовать потоковую репликацию между двумя серверами. Мы ограничимся самым простым вариантом; детально репликация рассматривается в курсе для администраторов DBA3.

Минимальный набор параметров, которые нужно проверить:

```
=> SELECT name, setting FROM pg_settings
WHERE name in ('wal_level', 'max_wal_senders');

name          | setting
-----+-----
max_wal_senders | 10
wal_level      | replica
(2 rows)
```

Начиная с версии PostgreSQL 10, параметры по умолчанию уже имеют подходящие значения.

Разрешение на подключение по протоколу репликации в pg_hba.conf:

```
=> SELECT type, user_name, address, auth_method FROM pg_hba_file_rules
WHERE 'replication' = ANY(database);

type | user_name | address | auth_method
-----+-----+-----+-----
local | {all}     |         | trust
host  | {all}     | 127.0.0.1 | scram-sha-256
host  | {all}     | ::1      | scram-sha-256
(3 rows)
```

Разрешение тоже имеется.

Развернем реплику из физической резервной копии, для этого используем утилиту pg_basebackup.

Целевой каталог копии должен быть пустым или отсутствовать:

```
student$ rm -rf /home/student/tmp/backup
```

Ключ --checkpoint=fast просит утилиту выполнить контрольную точку как можно быстрее (без пауз), а ключ -R — добавить настройки для реплики:

```
student$ pg_basebackup --pgdata=/home/student/tmp/backup -R --checkpoint=fast
```

Утилита создала заготовку конфигурационного файла...

```
student$ cat /home/student/tmp/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 sslnegotiation=postgres sslcompression=
```

... и сигнальный файл, который даст реплике указание перейти в режим постоянного восстановления:

```
student$ ls -l /home/student/tmp/backup/*.signal
-rw----- 1 student student 0 июн 24 05:20 /home/student/tmp/backup/standby.signal
```

Кластер, в котором мы собираемся развернуть реплику, уже предварительно инициализирован. Если сервер работает, его необходимо остановить:

```
student$ sudo pg_ctlcluster 16 replica stop
```

Cluster is not running.

Копия была размещена в домашнем каталоге пользователя student, а теперь переносим ее в каталог данных кластера и делаем владельцем файлов пользователя postgres:

```
student$ sudo rm -rf /var/lib/postgresql/16/replica
```

```
student$ sudo mv /home/student/tmp/backup /var/lib/postgresql/16/replica
```

```
student$ sudo chown -R postgres: /var/lib/postgresql/16/replica
```

Можно запускать сервер:

```
student$ sudo pg_ctlcluster 16 replica start
```

Посмотрим на процессы реплики.

```
student$ sudo head -n 1 '/var/lib/postgresql/16/replica/postmaster.pid'
```

42921

```
student$ ps -o pid,command --ppid 42921
```

```
    PID COMMAND
42922 postgres: 16/replica: checkpointer
42923 postgres: 16/replica: background writer
42924 postgres: 16/replica: startup recovering 00000001000000000000000000000007
42926 postgres: 16/replica: walreceiver streaming 0/7000060
```

Процесс walreceiver принимает поток журнальных записей, а процесс startup применяет изменения.

Сравним с процессами мастера.

```
student$ sudo head -n 1 '/var/lib/postgresql/16/main/postmaster.pid'
```

42500

```
student$ ps -o pid,command --ppid 42500
```

```
    PID COMMAND
42501 postgres: 16/main: checkpointer
42502 postgres: 16/main: background writer
42504 postgres: 16/main: walwriter
42505 postgres: 16/main: autovacuum launcher
42506 postgres: 16/main: logical replication launcher
42556 postgres: 16/main: student student [local] idle
42927 postgres: 16/main: walsender student [local] streaming 0/7000060
```

Процесс walsender отправляет реплике журнальные записи.

Состояние репликации можно смотреть в специальном представлении на мастере:

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

```

-[ RECORD 1 ]-----+-----
pid           | 42927
usesysid      | 16384
username      | student
application_name | 16/replica
client_addr   |
client_hostname |
client_port   | -1
backend_start | 2025-06-24 05:20:40.187427+03
backend_xmin  |
state         | streaming
sent_lsn      | 0/7000060
write_lsn     | 0/7000060
flush_lsn     | 0/7000060
replay_lsn    | 0/7000060
write_lag     | 00:00:00.060174
flush_lag     | 00:00:00.074944
replay_lag    | 00:00:00.075077
sync_priority | 0
sync_state    | async
reply_time    | 2025-06-24 05:20:40.187476+03

```

- значения `*_lsn` — показывают, какие журнальные записи отправлены на реплику, получены ею, записаны на диск и применены;
- `sync_state` — синхронная или асинхронная репликация (об этом подробнее позже).

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 т. ч. создание временных таблиц
- команды сопровождения (VACUUM, ANALYZE, REINDEX...)
- управление доступом (GRANT, REVOKE...)
- не срабатывают триггеры и рекомендательные блокировки

По умолчанию реплика работает в режиме *горячего резерва*, в этом случае разрешены клиентские подключения, но только для чтения данных. 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 (конечно, принимая во внимание возможное отставание от мастера).

В режиме горячего резерва на реплике не допускаются никакие изменения данных (включая последовательности), многие виды блокировок, команды DDL, а также такие команды, как `vacuum`, `analyze`, `reindex` и команды управления доступом — словом, все, что так или иначе изменяет данные.

При необходимости реплику можно запустить в режиме *теплого резерва*, задав параметр `hot_standby = off`, тогда подключения будут вообще невозможны.

<https://postgrespro.ru/docs/postgresql/16/hot-standby>

Использование реплики

Выполним несколько команд на основном сервере:

```
=> CREATE DATABASE replica_overview_physical;
```

CREATE DATABASE

```
=> \c replica_overview_physical
```

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

```
=> CREATE TABLE test(id integer PRIMARY KEY, descr text);
```

CREATE TABLE

Проверим реплику:

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

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

id	descr
-----+-----	
(0 rows)	

Вставим строку в таблицу на основном сервере:

```
=> INSERT INTO test VALUES (1, 'Раз');
```

INSERT 0 1

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

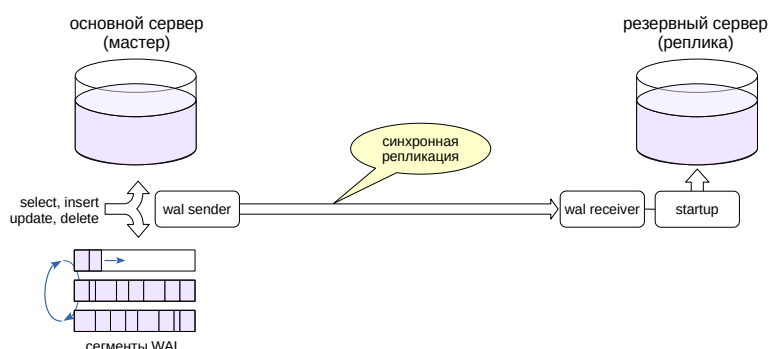
id	descr
-----+-----	
1	Раз
(1 row)	

Итак, репликация работает и запросы на реплике выполняются. При этом изменения на реплике не допускаются:

```
=> INSERT INTO test VALUES (2, 'Два');
```

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

надежность хранения данных



10

Механизм репликации позволяет построить систему так, чтобы она отвечала предъявляемым к ней требованиям. Рассмотрим несколько типичных задач и средства их решения.

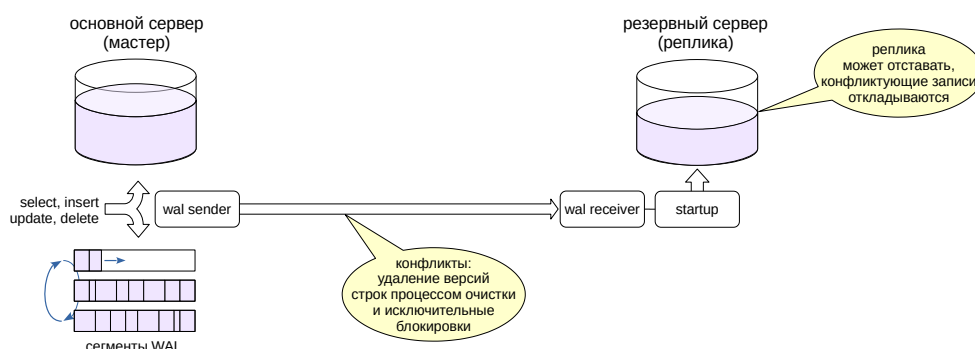
Одна из возможных задач — обеспечить надежность хранения данных.

Напомним, что фиксация транзакций может работать в синхронном и асинхронном режимах. В первом случае фиксация не завершается до тех пор, пока данные не будут надежно записаны на энерго-независимый носитель. Во втором — есть риск потерять часть зафиксированных данных, но фиксация не ждет записи на диск, и система работает быстрее.

Аналогичная картина и с репликацией. При синхронном режиме (*synchronous_commit = on*) и наличии реплики фиксация ждет не только записи WAL на диск, но и подтверждения приема журнальных записей от синхронной реплики. Это еще больше увеличивает надежность (данные не пропадут, даже если основной сервер выйдет из строя), но и еще больше замедляет систему.

Существуют и промежуточные варианты настройки, которые не дают полной гарантии надежности, но все-таки снижают вероятность потери данных.

выполнение долгих аналитических запросов (отчетов)



11

Как уже говорилось, долгие запросы удерживают горизонт очистки, из-за чего не удастся удалять ненужные версии строк. Если какие-то таблицы в это время активно изменяются, они могут сильно вырасти в размере. Поэтому для выполнения долгих аналитических запросов можно использовать реплику.

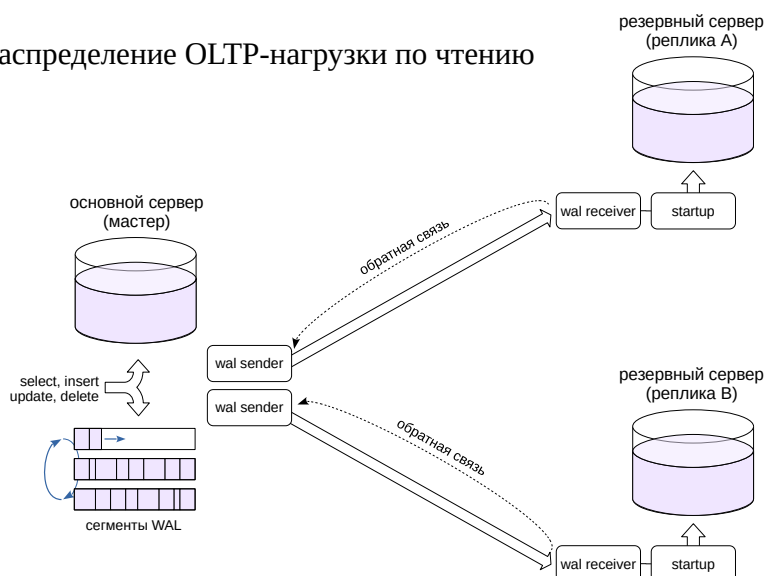
Тонкий момент состоит в том, что с основного сервера могут приходить журнальные записи, конфликтующие с выполняющимся запросом. Есть два источника таких записей.

1. Очистка удаляет версии строк, уже не нужные основному серверу, но еще нужные для выполнения запроса на реплике.
2. Исключительные блокировки, происходящие на основном сервере, несовместимые с запросом на реплике.

Поэтому «отчетную» реплику настраивают так, чтобы она принимала WAL-записи, но откладывала их применение, если они конфликтуют с запросами. Это приводит к тому, что данные на реплике могут «отставать» от основного сервера, но, как правило, для долгих аналитических запросов это не существенно.

Несколько реплик

распределение OLTP-нагрузки по чтению



12

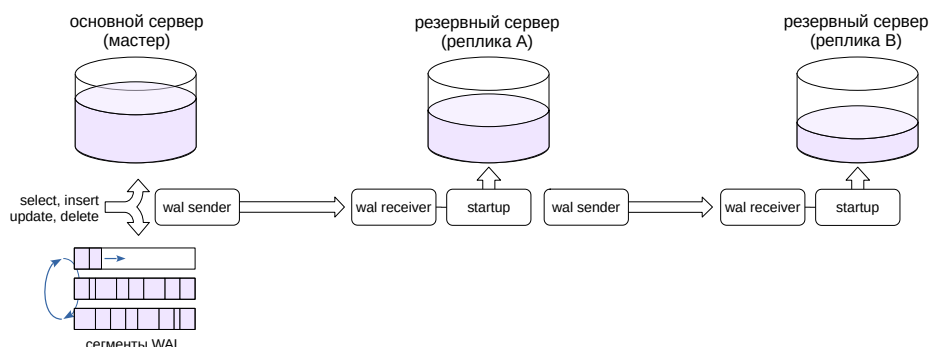
К основному серверу можно подключить несколько реплик, например, для распределения OLTP-нагрузки по чтению.

OLTP-запросы не должны быть долгими. Это позволяет использовать *обратную связь* между репликой и основным сервером по протоколу репликации. В этом случае основной сервер знает, какой горизонт транзакций нужен запросам на реплике, и очистка не будет удалять соответствующие версии строк. Иными словами, обратная связь дает такой же эффект, как если бы все запросы выполнялись непосредственно на основном сервере.

Однако репликация обеспечивает только базовый механизм. Для автоматического распределения нагрузки необходимы внешние средства (балансировщики). Следует также иметь в виду, что между основным сервером и репликами не гарантируется согласованность данных, даже в случае синхронной репликации. Если приложение читает данные только с одного из серверов, оно, разумеется, будет получать согласованные данные. Но это не выполняется, если приложение читает данные с нескольких серверов. С реплики можно прочитать как устаревшие данные, так и данные, которых еще не видно на основном сервере. Подробнее эти вопросы обсуждаются в курсе DBA3 «Резервное копирование и репликация».

Каскадная репликация

уменьшение нагрузки на мастер и перераспределение сетевого трафика



13

Несколько реплик, подключенных к одному основному серверу, будут создавать на него определенную нагрузку. Кроме того, надо учитывать нагрузку на сеть при пересылке нескольких копий потока журнальных записей.

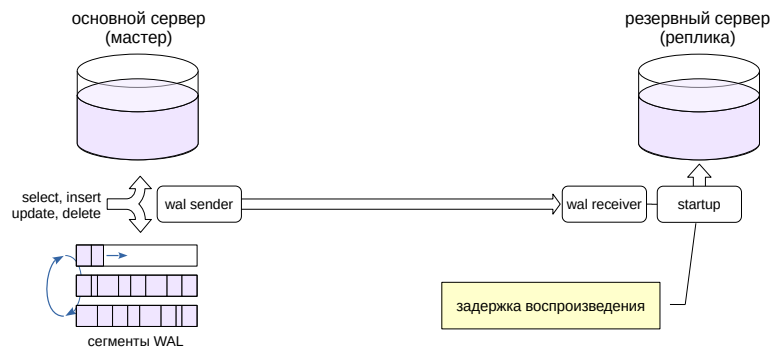
Для снижения нагрузки реплики можно соединять каскадом; при этом серверы передают журнальные записи друг другу по цепочке. Чем дальше от мастера, тем большее запаздывание может накопиться в изменяемых данных.

Заметим, что каскадная синхронная репликация не поддерживается: основной сервер может быть синхронизирован только с непосредственно подключенной к нему репликой. А вот обратная связь поступает к основному серверу от всех реплик.

Отложенная репликация

«машина времени»

и возможность восстановления на определенный момент без архива



14

Полезной может оказаться возможность просматривать данные на некоторый момент в прошлом и, при необходимости, восстановить сервер на этот момент. Это позволяет, в частности, справиться с ошибкой пользователя, совершившего неправильные действия, требующие отмены.

Обычный механизм восстановления из архива на момент времени (point-in-time recovery) в принципе позволяет решить эту задачу, но требует большой подготовительной работы и занимает много времени. А способа построить снимок данных по состоянию на произвольный момент в прошлом в PostgreSQL нет.

Задача решается созданием реплики, которая применяет записи WAL не сразу, а через установленный интервал времени.

В этом курсе мы не обсуждаем необходимые настройки для каждого из показанных вариантов. Для подробной информации обратитесь к курсу DBA3 «Резервное копирование и репликация».

Плановое переключение

останов основного сервера для технических работ без прерывания обслуживания
ручной режим

Аварийное переключение

переход на реплику из-за сбоя основного сервера
ручной режим,
но в принципе можно автоматизировать с помощью дополнительного кластерного ПО

Имеющуюся реплику можно использовать и для того, чтобы переключить на нее приложение с основного сервера.

Причины перехода на резервный сервер бывают разные. Это может быть необходимость технических работ на основном сервере — тогда переход выполняется в удобное время в штатном режиме. Возможен и сбой основного сервера, в таком случае переходить на резервный сервер нужно как можно быстрее, чтобы не прерывать обслуживание пользователей.

Даже в случае сбоя переход осуществляется вручную, если не используется специальное кластерное программное обеспечение, которое следит за состоянием серверов и может инициировать переход автоматически.

Переключение на реплику

Чтобы перевести реплику из режима восстановления в обычный режим, нужно дать ей соответствующую команду

```
=> SELECT pg_is_in_recovery(); -- является репликой?
```

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

```
student$ sudo pg_ctlcluster 16 replica promote
```

Начиная с версии 12 это же можно сделать, используя SQL-функцию `pg_promote`.

```
=> SELECT pg_is_in_recovery(); -- снова проверим: это реплика?
```

```
pg_is_in_recovery
-----
f
(1 row)
```

Таким образом мы получили два самостоятельных, никак не связанных друг с другом сервера.

```
=> INSERT INTO test VALUES (2, 'Два');
```

```
INSERT 0 1
```

Очень важно иметь гарантии того, что приложение работает только с одним из серверов, иначе возникает ситуация, называемая *split brain*: часть данных оказывается на одном сервере, часть — на другом, и собрать их воедино практически невозможно.

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

трансляция потока записей или файлов WAL

Физическая репликация создает точную копию всего кластера

однонаправленная, требует двоичной совместимости
базовый механизм для решения целого ряда задач



1. Разверните реплику так, как показано в демонстрации.
Проверьте, как работает приложение, если переключить его на использование реплики.
2. Добавьте к механизму фоновых заданий возможность выполнять задания на реплике с помощью расширения dblink.
Убедитесь, что долгое задание не приведет к появлению такой же долгой транзакции на основном сервере.

1. Переключатель сервера находится в верхней части информационной панели приложения.

Обратите внимание на то, какие операции по-прежнему работают, а какие — вызывают ошибку.

2. Приложение позволяет при отправке задания на выполнение указать удаленный сервер, и записывает имя узла и порт в таблицу tasks.

Добавьте в процедуру process_tasks проверку: если указаны узел и порт, вызывайте функцию run не локально, а с помощью расширения dblink на указанном сервере. (Расширение рассматривалось в теме «Фоновые процессы»)

Чтобы долгое задание не приводило к появлению такой же долгой транзакции на основном сервере, используйте dblink в асинхронном режиме, и периодически опрашивайте готовность запроса в отдельных, коротких, транзакциях.

1. Развертывание реплики

Выполняем те же команды, что и в демонстрации.

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

```
student$ sudo pg_ctlcluster 16 replica stop
```

Cluster is not running.

```
student$ sudo rm -rf /var/lib/postgresql/16/replica
```

```
student$ sudo mv /home/student/backup /var/lib/postgresql/16/replica
```

```
student$ sudo chown -R postgres: /var/lib/postgresql/16/replica
```

```
student$ sudo pg_ctlcluster 16 replica start
```

При переключении приложения на реплику:

- Поиск книг и другие операции, не требующие изменения данных, работают;
- Вход, покупка книг и другие операции, изменяющие данные, выдают ошибку.

2. Фоновые задания в удаленном режиме

Добавим в процедуру запуска задания обработку узла и порта. Если они указаны, будем выполнять функцию `gup` на указанном сервере с помощью расширения `dblink`. За основу берем вариант процедуры, написанной в практике к теме «Асинхронная обработка».

```
=> CREATE EXTENSION IF NOT EXISTS dblink;
```

```
CREATE EXTENSION
```

```

=> CREATE OR REPLACE PROCEDURE process_tasks() AS $$
DECLARE
    task tasks;
    result text;
    ctx text;
BEGIN
    SET application_name = 'process_tasks';
    <<forever>>
    LOOP
        COMMIT;
        PERFORM pg_sleep(1);
        SELECT * INTO task FROM take_task();
        COMMIT;
        CONTINUE forever WHEN task.task_id IS NULL;

        IF task.host IS NULL THEN -- запускаем локально
            BEGIN
                result := empapi.run(task);
                PERFORM complete_task(task, 'finished', result);
            EXCEPTION
                WHEN others THEN
                    GET STACKED DIAGNOSTICS
                        result := MESSAGE_TEXT,
                        ctx := PG_EXCEPTION_CONTEXT;
                    PERFORM complete_task(
                        task, 'error', result || E'\n' || ctx
                    );
            END;
        ELSE -- запускаем удаленно в асинхронном режиме
            BEGIN
                PERFORM dblink_connect(
                    'remote',
                    format(
                        'host=%s port=%s dbname=%s user=%s password=%s',
                        task.host, task.port, 'bookstore2', 'student', 'student'
                    )
                );
            EXCEPTION
                WHEN others THEN
                    GET STACKED DIAGNOSTICS
                        result := MESSAGE_TEXT,
                        ctx := PG_EXCEPTION_CONTEXT;
                    PERFORM complete_task(
                        task, 'error', result || E'\n' || ctx
                    );
                CONTINUE forever;
            END;
            PERFORM dblink_send_query(
                'remote',
                format('SELECT * FROM empapi.run(%L)', task)
            );
            -- ожидание результата
            LOOP
                PERFORM pg_sleep(1);
                EXIT WHEN (SELECT dblink_is_busy('remote')) = 0;
                COMMIT;
            END LOOP;
            -- получение результата
            BEGIN
                SELECT s INTO result
                FROM dblink_get_result('remote') AS (s text);
                PERFORM complete_task(task, 'finished', result);
            EXCEPTION
                WHEN others THEN
                    GET STACKED DIAGNOSTICS
                        result := MESSAGE_TEXT,
                        ctx := PG_EXCEPTION_CONTEXT;
                    PERFORM complete_task(
                        task, 'error', result || E'\n' || ctx
                    );
            END;
            PERFORM dblink_disconnect('remote');
        END IF;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

CREATE PROCEDURE

```

Обратите внимание на следующее:

- Для краткости не обрабатывается статус вызова функции `dblink_send_query`, что, конечно же, необходимо делать.
- В цикле ожидания результатов удаленного запуска выполняется фиксация. В противном случае получалась бы долгая транзакция, удерживающая горизонт базы данных.
- Дублируется код для обработки исключительных ситуаций. Это связано с тем, что процедура не может выполнять фиксацию внутри блока с секцией `EXCEPTION`.
- Команда `COMMIT` перенесена из конца цикла `forever` в начало. Это сделано для удобства прерывания обработки в случае ошибки (см. обработчик `dblink_connect`).

Работающий в системе фоновый процесс, выполняющий процедуру `process_tasks`, будет продолжать выполнять старую версию процедуры. Его необходимо прервать и перезапустить:

```
=> SELECT pg_terminate_backend(pid)
      FROM pg_stat_activity
      WHERE application_name = 'process_tasks';

pg_terminate_backend
-----
(0 rows)

=> SELECT * FROM pg_background_detach(
      pg_background_launch('CALL process_tasks()')
);

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

1. Настройте физическую потоковую репликацию между двумя серверами в синхронном режиме. Проверьте работу репликации. Убедитесь, что при остановленной реплике фиксация не завершается.
2. По умолчанию применение конфликтующих записей на реплике откладывается максимум на 30 секунд. Отключите откладывание применения и убедитесь, что долгий запрос, выполняющийся на реплике, будет прерван, если необходимые ему версии строк удаляются и очищаются на мастере.
Включите обратную связь и убедитесь, что теперь запрос не прерывается из-за того, что на мастере откладывается очистка.

20

1. Для этого на мастере установите с помощью ALTER SYSTEM следующие параметры:

- `synchronous_commit = on`
- `synchronous_standby_names = "16/replica"`

2. За откладывание применения конфликтующих записей на реплике отвечает параметр `max_standby_streaming_delay`. Установите его в 0. Обратная связь включается параметром `hot_standby_feedback = on`. Оба параметра устанавливаются через ALTER SYSTEM с последующим перечитыванием настроек.

Чтобы запрос выполнялся долго на небольшом объеме данных, можно добавить в него вызов функции `pg_sleep`.

1. Синхронная репликация

Разворачиваем реплику, как было показано в демонстрации:

```
student$ pg_basebackup --pgdata=/home/student/tmp/backup -R --checkpoint=fast
```

```
student$ sudo pg_ctlcluster 16 replica stop
```

Cluster is not running.

```
student$ sudo rm -rf /var/lib/postgresql/16/replica
```

```
student$ sudo mv /home/student/tmp/backup /var/lib/postgresql/16/replica
```

```
student$ sudo chown -R postgres: /var/lib/postgresql/16/replica
```

Запускаем реплику.

```
student$ sudo pg_ctlcluster 16 replica start
```

Настроим синхронную репликацию на мастере. По умолчанию синхронный режим включен, но записи о фиксации транзакций синхронизируются только с локальной файловой системой:

```
=> SHOW synchronous_commit;
```

```
synchronous_commit
-----
on
(1 row)
```

А синхронизация с репликой не настроена:

```
=> SHOW synchronous_standby_names;
```

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

Реплик может быть несколько, и мастер должен знать, с какой из них синхронизироваться. Реплика представляется именем, заданным в ее параметре cluster_name:

```
student$ psql -p 5433
```

```
| => SHOW cluster_name;
```

```
| cluster_name
|-----
| 16/replica
| (1 row)
```

```
=> ALTER SYSTEM SET synchronous_standby_names = '"16/replica"';
```

```
ALTER SYSTEM
```

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

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

```
=> SELECT sync_state FROM pg_stat_replication;
```

```
sync_state
-----
sync
(1 row)
```

Репликация стала синхронной.

```
=> CREATE DATABASE replica_overview_physical;
```

```
CREATE DATABASE
```

```
=> \c replica_overview_physical
```

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

Теперь остановим реплику...

```
student$ sudo pg_ctlcluster 16 replica stop
```

...и попробуем выполнить какую-либо транзакцию:

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

Управление возвратится только когда реплика будет снова запущена и репликация восстановится:

```
student$ sudo pg_ctlcluster 16 replica start
```

```
CREATE TABLE
```

2. Конфликтующие записи

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

Отключаем откладывание применения конфликтующих записей:

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

```
ALTER SYSTEM
```

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

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

Добавляем строки в таблицу:

```
=> INSERT INTO test(n) SELECT id FROM generate_series(1,10) AS id;
```

```
INSERT 0 10
```

Выполняем на реплике долгий запрос...

```
=> SELECT pg_sleep(5), count(*) FROM test;
```

...а в это время на мастере удаляем строки из таблицы и выполняем очистку:

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

```
DELETE 10
```

```
=> VACUUM VERBOSE test;
```

```
INFO:  vacuuming "replica_overview_physical.public.test"
INFO:  table "test": truncated 1 to 0 pages
INFO:  finished vacuuming "replica_overview_physical.public.test": index scans: 0
pages: 1 removed, 0 remain, 1 scanned (100.00% of total)
tuples: 10 removed, 0 remain, 0 are dead but not yet removable
removable cutoff: 819, which was 1 XIDs old when operation ended
new relfrozenxid: 819, which is 3 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 1 pages from table (100.00% of total) had 10 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 3.017 MB/s
buffer usage: 10 hits, 0 misses, 4 dirtied
WAL usage: 6 records, 1 full page images, 8675 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s
VACUUM
```

Очистка стерла все версии строк (10 removed). В итоге запрос на реплике завершается ошибкой:

```
ERROR:  canceling statement due to conflict with recovery
DETAIL:  User query might have needed to see row versions that must be removed.
```

Повторим эксперимент со включенной обратной связью.

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

```
ALTER SYSTEM
```

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

```

pg_reload_conf
-----
t
(1 row)

```

```
=> INSERT INTO test(n) SELECT id FROM generate_series(1,10) AS id;
```

```
INSERT 0 10
```

```
=> SELECT pg_sleep(5), count(*) FROM test;
```

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

```
DELETE 10
```

```
=> VACUUM VERBOSE test;
```

```

INFO:  vacuuming "replica_overview_physical.public.test"
INFO:  finished vacuuming "replica_overview_physical.public.test": index scans: 0
pages: 0 removed, 1 remain, 1 scanned (100.00% of total)
tuples: 0 removed, 10 remain, 10 are dead but not yet removable
removable cutoff: 820, which was 2 XIDs old when operation ended
new relfrozenxid: 820, which is 1 XIDs ahead of previous value
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers
removed
avg read rate: 0.000 MB/s, avg write rate: 63.004 MB/s
buffer usage: 8 hits, 0 misses, 1 dirtied
WAL usage: 1 records, 0 full page images, 188 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM

```

Теперь очистка не удаляет версии строк, поскольку знает о запросе, выполняющемся на реплике (10 are dead but not yet removable) и запрос обрабатывает:

```

pg_sleep | count
-----+-----
          |    10
(1 row)

```

Итак:

- В первом случае (max_standby_streaming_delay) откладывается воспроизведение журнальных записей на реплике.
- Во втором случае (hot_standby_feedback) откладывается очистка на мастере.

Отключим синхронную репликацию.

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

```
ALTER SYSTEM
```

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

```

pg_reload_conf
-----
t
(1 row)

```