Расширяемость

Физическая репликация

Авторы: Егор Рогов, Павел Лузанов

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

Темы

Физическая репликация

Уровни журнала

Варианты использования реплики

Переключение на реплику

Физическая репликация

Синхронизация копий кластера на нескольких серверах

Основные задачи

- высокая доступность, отказоустойчивость
- масштабируемость

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

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).

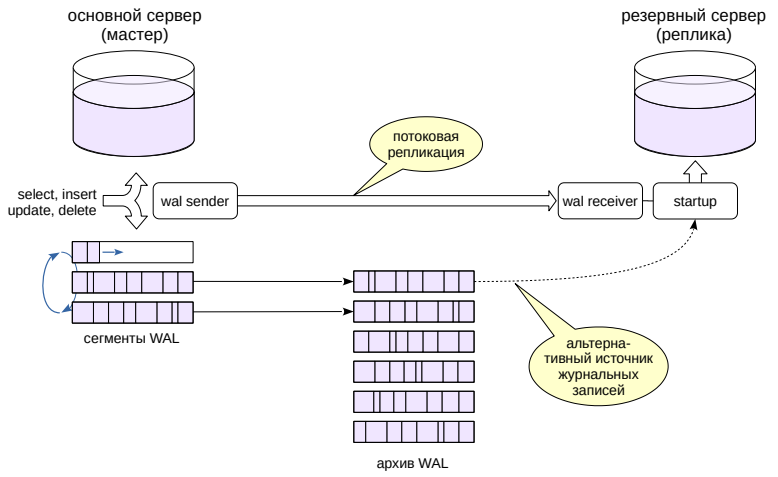
Один сервер может не справляться с нагрузкой. Наращивать ресурсы сервера может оказаться невыгодно или вообще невозможно. Но можно распределить нагрузку на несколько серверов (масштабирование).

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

Механизм репликации состоит в том, что один сервер передает журнальные записи на другой сервер, и 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



Чтобы организовать репликацию между двумя серверами, мы создаем реплику из резервной копии основного сервера. Но при обычном восстановлении из резервной копии мы получаем новый независимый сервер. А в данном случае реплика начинает работать в режиме *постоянного восстановления*: она все время применят новые журнальные записи, приходящие с основного сервера (этим занимается процесс startup). Таким образом, реплика постоянно поддерживается в почти актуальном состоянии.

Если реплика не допускает подключений, она называется «теплым резервом». Однако можно сделать и «горячий резерв» — реплика будет допускать подключения для чтения данных.

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

В этом случае реплика (процесс wal_receiver) подключается к мастеру (процесс wal_sender) по протоколу репликации и читает поток записей WAL. За счет этого при потоковой репликации отставание реплики сведено к минимуму, и даже к нулю при синхронном режиме.

Кроме этого, может использоваться архив сегментов WAL, если в системе настроено непрерывное архивирование. Если реплика не сможет получить очередную журнальную запись по протоколу репликации, она будет пробовать прочитать соответствующий файл из архива.

https://postgrespro.ru/docs/postgresql/12/high-availability

Уровни журнала

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 данные.

Объем информации, попадающий 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 часть данных записывается сразу на энергонезависимый носитель и не записывается в журнал.

Сейчас уровень по умолчанию — **replica**. Этот уровень дополнительно 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 установить 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 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 database = ARRAY['replication'];
```

```
 type  | user_name |  address  | auth_method
-------+-----------+-----------+-------------
 local | {all}     |           | peer
 host  | {all}     | 127.0.0.1 | md5
 host  | {all}     | ::1       | md5
(3 rows)
```

Разрешение тоже имеется.

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

```
student$ sudo pg_ctlcluster 12 replica stop
```

```
Cluster is not running.
```

Развернем реплику из резервной копии, которую сейчас и создадим. В качестве каталога для сохранения копии используем каталог PGDATA реплики, который мы предварительно очистим:

```
postgres$ rm -rf /var/lib/postgresql/12/replica/*
```

Ключ -R создаст указание серверу перейти в режим постоянного восстановления (что нам и требуется), а также поместит в файл postgresql.auto.conf настройки для подключения к основному серверу.

```
postgres$ pg_basebackup --pgdata=/var/lib/postgresql/12/replica -R
```

Вот какую заготовку создала утилита pg_basebackup:

```
postgres$ cat /var/lib/postgresql/12/replica/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
primary_conninfo = 'user=postgres passfile=''/var/lib/postgresql/.pgpass'' channel_binding=prefer host=''/var/run/postgresql'' port=5432 sslmode=prefer sslcompression=0 ssl_min_protoco
```

И можно запускать сервер:

```
student$ sudo pg_ctlcluster 12 replica start
```

Посмотрим на процессы реплики.

```
postgres$ ps -o pid,command --ppid `head -n 1 /var/lib/postgresql/12/replica/postmaster.pid`
```

```
    PID COMMAND
  34014 postgres: 12/replica: startup   recovering 000000010000000000000AB
  34016 postgres: 12/replica: checkpointer
  34017 postgres: 12/replica: background writer
  34018 postgres: 12/replica: stats collector
  34019 postgres: 12/replica: walreceiver   streaming 0/AB024350
```

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

И сравним с процессами мастера.

```
postgres$ ps -o pid,command --ppid `head -n 1 /var/lib/postgresql/12/main/postmaster.pid`
```

```
    PID COMMAND
  17504 postgres: 12/main: checkpointer
  17505 postgres: 12/main: background writer
  17506 postgres: 12/main: walwriter
  17507 postgres: 12/main: autovacuum launcher
  17508 postgres: 12/main: stats collector
  17509 postgres: 12/main: logical replication launcher
  17537 postgres: 12/main: student student 127.0.0.1(59856) idle
  17816 postgres: 12/main: student student 127.0.0.1(59862) idle
  33750 postgres: 12/main: student student [local] idle
  34020 postgres: 12/main: walsender postgres [local] streaming 0/AB024350
```

Здесь добавился процесс wal sender.

Состояние репликации можно смотреть в специальном представлении на мастере:

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

```
-[ RECORD 1 ]----+------------------------------
pid              | 34020
usesysid         | 10
usename          | postgres
application_name | 12/replica
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2022-12-09 23:47:49.266754+03
backend_xmin     |
state            | streaming
sent_lsn         | 0/AB024350
write_lsn        | 0/AB024350
flush_lsn        | 0/AB024350
replay_lsn       | 0/AB024350
write_lag        | 00:00:00.358517
flush_lag        | 00:00:00.374622
replay_lag       | 00:00:00.421183
sync_priority    | 0
sync_state       | async
reply_time       | 2022-12-09 23:47:49.689946+03
```

- *_lsn — показывают, какие журнальные записи отправлены на реплику и дошли до нее;
- sync_state — синхронная или асинхронная репликация (об этом подробнее позже).

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

В режиме горячего резерва на реплике не допускаются любые изменения данных (включая последовательности), блокировки, команды DDL, такие команды, как vacuum и analyze, команды управления доступом — словом, все, что так или иначе изменяет данные.

При этом реплика может выполнять запросы на чтение 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).

https://postgrespro.ru/docs/postgresql/12/hot-standby

Использование реплики

Выполним несколько команд на основном сервере:

```
=> CREATE DATABASE ext_repl_physical;
```

```
CREATE DATABASE
```

```
=> \c ext_repl_physical;
```

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

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

```
CREATE TABLE
```

Проверим реплику:

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

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

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

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

```
INSERT 0 1
```

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

```
 id | descr
----+-------
  1 | Раз
(1 row)
```

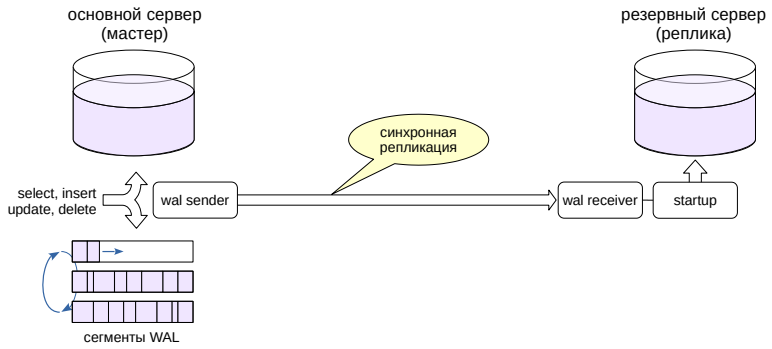
Итак, репликация работает и запросы на реплике выполняются. При этом изменения на реплике не допускаются:

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

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

Использование реплики

надежность хранения данных



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 — обеспечение надежности хранения данных.

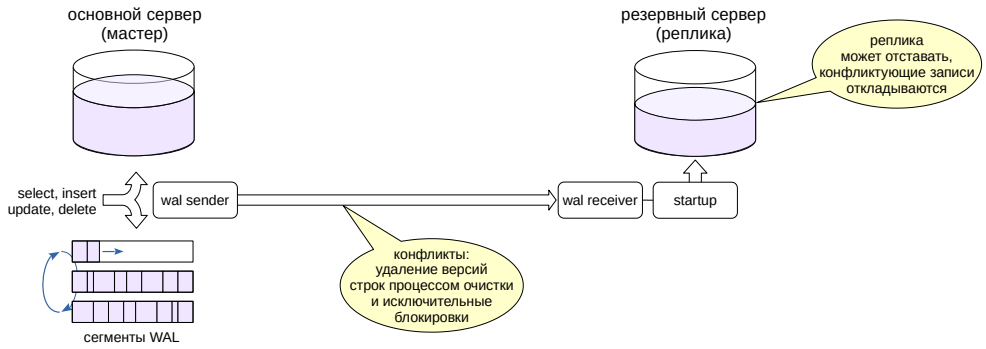
Напомним, что фиксация транзакций может работать в синхронном и асинхронном режимах. В первом случае фиксация не завершается до тех пор, пока данные не будут надежно записаны на энерго-независимый носитель. Во втором — есть риск потерять часть зафиксированных данных, но фиксация не должна ждать записи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

Использование реплики

выполнение долгих аналитических запросов (отчетов)



Как уже говорилось, длинные запросы удерживают горизонт транзакций, из-за чего очистка не может 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.

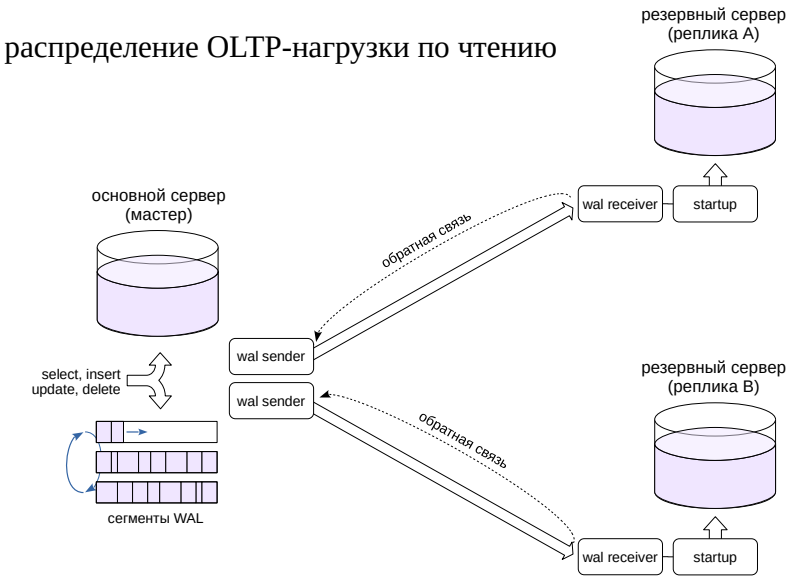
Тонкий момент состоит в том, что с основного сервера могут приходить журнальные записи, конфликтующие с выполняющимся запросом. Есть два источника таких записей.

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 запросов это не существенно.

Несколько реплик



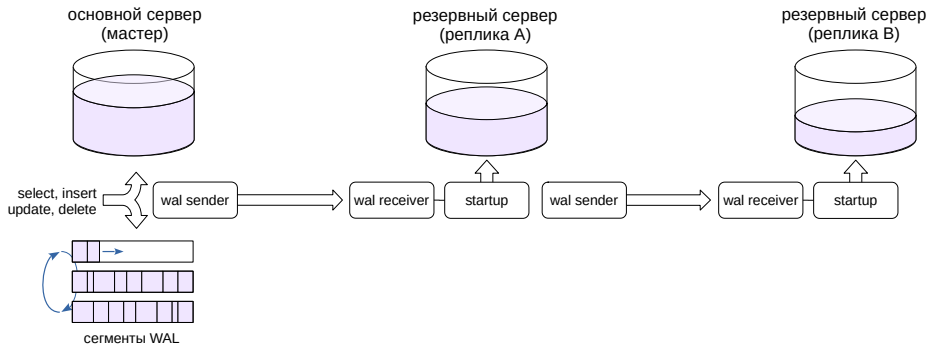
К основному серверу можно подключить несколько реплик, например, для распределения OLTP-нагрузки по чтению.

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 для реплики, 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



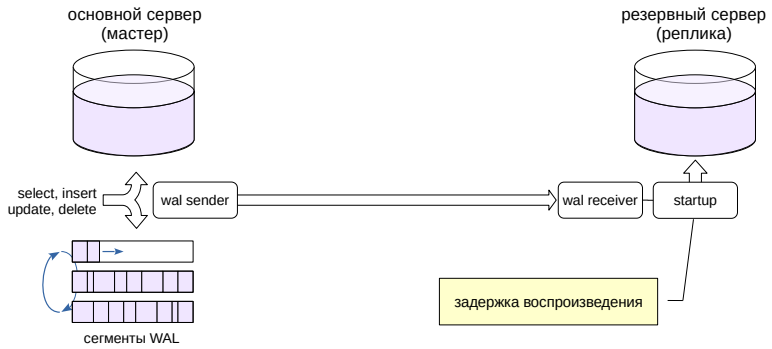
Несколько реплик, подключенных к одному основному серверу, будут создавать на него определенную нагрузку. Кроме того, надо учитывать нагрузку на сеть для пересылки нескольких копий потока журнальных записей.

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.

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

Отложенная репликация

«машина времени»
и возможность восстановления на определенный момент без архива



Полезной является 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.

Проблема в том, что обычный механизм восстановления из архива на момент времени (point-in-time recovery) в принципе позволяет решить 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».

Переключение на реплику

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 проведения технических работ на основном сервере — тогда переход выполняется в удобное время в штатном режиме. А может быть сбой основного сервера, и 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 команду.

```
=> SELECT pg_is_in_recovery();
```

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

```
student$ sudo pg_ctlcluster 12 replica promote
```

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

```
=> SELECT pg_is_in_recovery();
```

```
 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.

Итоги

Механизм 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

Практика

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.

```
student$ psql bookstore2
```

1. Развертывание реплики

Выполняем те же команды, что и в демонстрации.

```
student$ sudo pg_ctlcluster 12 replica stop
```

```
Cluster is not running.
```

```
postgres$ rm -rf /var/lib/postgresql/12/replica/*
```

```
postgres$ pg_basebackup --pgdata=/var/lib/postgresql/12/replica -R
```

```
postgres$ echo 'hot_standby = on' >> /var/lib/postgresql/12/replica/postgresql.auto.conf
```

```
student$ sudo pg_ctlcluster 12 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 run 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', 'emp', 'emp'
                    )
                );
            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 r(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
----------------------
 t
(1 row)


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

 pg_background_detach
----------------------

(1 row)
```

Практика

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.

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

- *synchronous_commit* = on,

- *synchronous_standby_names* = 'replica',

и перечитайте настройки, а на реплике добавьте в параметр *primary_conninfo* (в сгенерированном утилитой pg_basebackup файле postgresql.auto.conf) указание «application_name=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 ALTER SYSTEM с последующим перечитыванием настроек.

Чтобы запрос выполнялся долго на небольшом объеме данных, можно, например, создать таблицу, содержащую числа от 1 до N, и вычислять факториал каждого числа:

SELECT count(*) FROM t WHERE n! > 0;

В таблице должно быть около 1000 строк (подберите подходящее число в зависимости от мощности процессора):

1. Синхронная репликация

Установим необходимые для синхронной репликации параметры:

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

```
ALTER SYSTEM
```

```
=> ALTER SYSTEM SET synchronous_standby_names = 'replica';
```

```
ALTER SYSTEM
```

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

```
ALTER SYSTEM
```

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

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

Разворачиваем реплику, как было показано в демонстрации:

```
student$ sudo pg_ctlcluster 12 replica stop
```

```
Cluster is not running.
```

```
postgres$ rm -rf /var/lib/postgresql/12/replica/*
```

```
postgres$ pg_basebackup --pgdata=/var/lib/postgresql/12/replica -R
```

Заменим параметр primary_conninfo, указав значение application_name (оно должно соответствовать значению, записанному в параметр synchronous_standby_names):

```
postgres$ sed -i "s/primary_conninfo.*/primary_conninfo = 'port=5432 user=postgres application_name=replica'/" /var/lib/postgresql/12/replica/postgresql.auto.conf
```

Добавим еще параметр горячего резерва:

```
postgres$ echo 'hot_standby = on' >> /var/lib/postgresql/12/replica/postgresql.auto.conf
```

Вот что получилось:

```
postgres$ cat /var/lib/postgresql/12/replica/postgresql.auto.conf
```

```
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
synchronous_commit = 'on'
synchronous_standby_names = 'replica'
max_standby_streaming_delay = '0'
primary_conninfo = 'port=5432 user=postgres application_name=replica'
hot_standby = on
```

Запускаем реплику.

```
student$ sudo pg_ctlcluster 12 replica start
```

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

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

Состояние репликации — синхронное.

```
=> CREATE DATABASE ext_repl_physical;
```

```
CREATE DATABASE
```

```
=> \c ext_repl_physical
```

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

Теперь остановим реплику...

```
student$ sudo pg_ctlcluster 12 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 12 replica start
```

```
CREATE TABLE
```

2. Конфликтующие записи

```
student$ psql -p 5433 -d ext_repl_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 1000 FROM generate_series(1,1000) AS id;
```

```
INSERT 0 1000
```

Выполняем на реплике долгий запрос...

```
=> SELECT count(*) FROM test WHERE n! > 0;
```

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

```
=> DELETE FROM test;

DELETE 1000

=> VACUUM VERBOSE test;

INFO:  vacuuming "public.test"
INFO:  "test": removed 1000 row versions in 5 pages
INFO:  "test": found 1000 removable, 0 nonremovable row versions in 5 out of 5 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 156027
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.
INFO:  "test": truncated 5 to 0 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
VACUUM
```

Очистка стерла все версии строк. В итоге запрос на реплике завершается ошибкой:

```
 count
-------
  1000
(1 row)
```

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 1000 FROM generate_series(1,1000) AS id;

INSERT 0 1000

=> SELECT count(*) FROM test WHERE n! > 0;

=> DELETE FROM test;

DELETE 1000

=> VACUUM VERBOSE test;

INFO:  vacuuming "public.test"
INFO:  "test": found 0 removable, 1000 nonremovable row versions in 5 out of 5 pages
DETAIL:  1000 dead row versions cannot be removed yet, oldest xmin: 156028
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
```

Теперь очистка не удаляет версии строк, поскольку знает о запросе, выполняющемся на реплике. Запрос отрабатывает:

```
 count
-------
  1000
(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.