Расширяемость

Логическая репликация

Авторы: Егор Рогов, Павел Лузанов

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

Темы

Логическая репликация

Публикации и подписчики

Обнаружение и разрешение конфликтов

Особенности работы триггеров

Варианты использования логической репликации

Логическая репликация

Механизм

публикующий сервер читает собственные журнальные записи, декодирует их в изменения строк данных и отправляет подписчикам

сервер-подписчик применяет изменения к своим таблицам

Особенности

публикация-подписчик: поток данных возможен в обе стороны

требуется совместимость на уровне протокола

возможна выборочная репликация отдельных таблиц

При физической репликации только один сервер (мастер) выполняет изменения и генерирует журнальные записи. Остальные серверы (реплики) только читают журнальные записи мастера и применяют их.

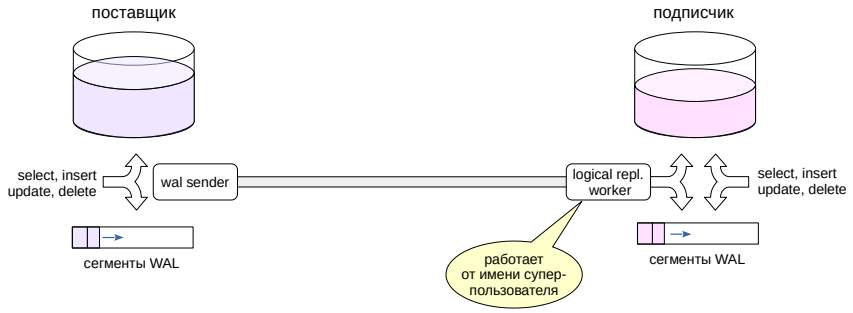
При логической репликации все серверы работают в обычном режиме, могут изменять данные и генерировать собственные журнальные записи. Один сервер может *публиковать* свои изменения, а другие — *подписываться* на них. При этом один и тот же сервер может как публиковать изменения, так и подписываться на другие. Это позволяет организовать произвольные потоки данных между серверами.

Сервер-публикатор читает собственные журнальные записи, но не пересылает их подписчикам «как есть» (как при физической репликации), а предварительно декодирует их в «логический», независимый от платформы и версии PostgreSQL вид. Поэтому для логической репликации не нужна двоичная совместимость, реплика должна лишь понимать протокол репликации. Также можно получать и применять не все изменения, а только отдельных таблиц.

Логическая репликация доступна, начиная с версии 10; более ранние версии должны были использовать расширение pg_logical (https://www.2ndquadrant.com/en/resources/pglogical/), либо организовывать репликацию с помощью триггеров.

https://postgrespro.ru/docs/postgresql/12/logical-replication

Логическая репликация



На рисунке: фоновый процесс logical replication worker на сервере-подписчике получает информацию от публикующего сервера и применяет ее. Этот процесс работает от имени суперпользователя, чтобы иметь доступ ко всем таблицам, участвующим в публикации.

В это же время сервер-подписчик работает обычным образом и принимает запросы и на чтение, и на запись.

Уровни журнала

Параметр *wal_level*

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

Чтобы публикующий сервер смог сопоставить собственные журнальные записи с изменениями табличных строк, в журнале необходима дополнительная информации. В основном она касается возможности узнать схему данных любой таблицы в любой момент времени.

Такой уровень журнала называется **logical**. На этом уровне в журнал записывается та же информация, что и на уровне replica, и добавляется возможность логической репликации.

Поскольку по умолчанию используется уровень replica, значение параметра wal_level потребуется изменить на публикующем сервере.

Публикации и подписчики

Публикующий сервер

- публикация включает несколько таблиц одной базы данных
- изменения данных выдаются построчно в порядке фиксации транзакций (реплицируются строки, а не команды SQL)

Подписчики

- получают и применяют изменения
- возможна начальная синхронизация данных
- без разбора, трансформаций и планирования — сразу выполнение
- возможны конфликты с локальными данными

Логическая репликация использует модель «публикация — подписчик». На одном сервере создается *публикация,* которая может включать ряд таблиц одной базы данных. Другие серверы могут *подписываться* на эту публикацию и получать и применять изменения.

Реплицируются измененные строки таблиц (а не команды SQL). Команды DDL не передаются, то есть таблицы-приемники на стороне подписчика надо создавать вручную. Но есть возможность автоматической начальной синхронизации содержимого таблиц при создании подписки.

Технически информация об измененных строках извлекается и декодируется из имеющегося журнала WAL на публикующем сервере, а затем пересылается процессом wal sender подписчику по протоколу репликации.

Процесс logical replication worker на подписчике принимает информацию и применяет изменения.

Применение изменений происходит без выполнения команд SQL и связанных с этим накладных расходов на разбор и планирование, что уменьшает нагрузку на подписчика.

https://postgrespro.ru/docs/postgresql/12/sql-createpublication

https://postgrespro.ru/docs/postgresql/12/sql-createsubscription

Логическая репликация

Пусть на первом сервере имеется таблица:

```
=> CREATE DATABASE ext_repl_logical;

CREATE DATABASE

=> \c ext_repl_logical;

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

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

CREATE TABLE
```

Развернем реплику из резервной копии, как мы делали в теме «Физическая репликация». Но команде pg_basebackup не будем передавать ключ -R, поскольку нам требуется отдельный сервер, а не реплика.

```
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

student$ sudo pg_ctlcluster 12 replica start
```

Добавим в таблицу на первом сервере несколько строк:

```
=> INSERT INTO test VALUES (1, 'Раз'), (2, 'Два');

INSERT 0 2
```

На втором сервере этих данных, естественно, нет, но сама таблица была перенесена из резервной копии:

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

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

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

Мы хотим настроить между серверами логическую репликацию. Для этого нам понадобится дополнительная информация в журнале (на публикующем сервере).

```
=> ALTER SYSTEM SET wal_level = logical;

ALTER SYSTEM

student$ sudo pg_ctlcluster 12 main restart
```

На первом сервере создаем публикацию:

```
student$ psql -d ext_repl_logical

=> CREATE PUBLICATION test_pub FOR TABLE test;

CREATE PUBLICATION

=> \dRp+
```

```
                     Publication test_pub
  Owner  | All tables | Inserts | Updates | Deletes | Truncates
---------+------------+---------+---------+---------+-----------
 student | f          | t       | t       | t       | t
Tables:
    "public.test"
```

На втором сервере подписываемся на эту публикацию:

```
=> CREATE SUBSCRIPTION test_sub
CONNECTION 'host=localhost port=5432 user=postgres password=postgres dbname=ext_repl_logical'
PUBLICATION test_pub;
```

```
NOTICE:  created replication slot "test_sub" on publisher
CREATE SUBSCRIPTION
```

Обратите внимание на сообщение о создании репликационного слота, причем на публикующем (!) сервере. При

использовании репликации есть опасность, что после очередной контрольной точки публикующий сервер удалит ненужный (для восстановления) сегмент WAL, данные из которого еще не переданы на сервер-подписчик. Чтобы этого не происходило, и используется слот. Это специальный объект базы данных, который связан с потоком журнальных записей и не позволяет серверу удалять записи, еще не переданные клиенту через этот поток.

При физической репликации тоже можно использовать слот, но это не обязательно.

```
=> \dRs
```

```
            List of subscriptions
   Name   |  Owner  | Enabled | Publication
----------+---------+---------+-------------
 test_sub | student | t       | {test_pub}
(1 row)
```

При создании подписки все данные из таблицы test были переданы на сервер-подписчик:

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

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

Если это не требуется, можно добавить к команде CREATE SUBSCRIPTION предложение

```
WITH (copy_data = false)
```

Проверим работу репликации:

```
=> INSERT INTO test VALUES (3, 'Три');
```

```
INSERT 0 1
```

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

```
 id | descr
----+-------
  1 | Раз
  2 | Два
  3 | Три
(3 rows)
```

При этом сервер-подписчик тоже может изменять данные в таблице (но эти изменения не будут реплицированы на публикующий сервер):

```
=> INSERT INTO test VALUES (4, 'Четыре');
```

```
INSERT 0 1
```

Состояние подписки можно посмотреть в представлении:

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

```
-[ RECORD 1 ]---------+------------------------------
subid                 | 346020
subname               | test_sub
pid                   | 35300
relid                 |
received_lsn          | 0/AD008A68
last_msg_send_time    | 2022-12-09 23:48:51.562872+03
last_msg_receipt_time | 2022-12-09 23:48:51.563146+03
latest_end_lsn        | 0/AD008A68
latest_end_time       | 2022-12-09 23:48:51.562872+03
```

К процессам сервера добавился logical replication worker (его номер указан в pg_stat_subscription.pid):

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

```
    PID COMMAND
  35010 postgres: 12/replica: checkpointer
  35011 postgres: 12/replica: background writer
  35012 postgres: 12/replica: walwriter
  35013 postgres: 12/replica: autovacuum launcher
  35014 postgres: 12/replica: stats collector
  35015 postgres: 12/replica: logical replication launcher
  35096 postgres: 12/replica: student ext_repl_logical [local] idle
  35300 postgres: 12/replica: logical replication worker for subscription 346020
```

Конфликты

Режимы идентификации для изменения и удаления

- столбцы первичного ключа (по умолчанию)
- столбцы указанного уникального индекса с ограничением NOT NULL
- все столбцы
- без идентификации (по умолчанию для системного каталога)

Конфликты — нарушение ограничений целостности

- репликация приостанавливается до устранения конфликта вручную
- либо исправление данных,
- либо пропуск конфликтующей транзакции

Вставка новых строк происходит достаточно просто. Интереснее обстоит дело при изменениях и удалениях — в этом случае надо как-то идентифицировать старую версию строки. По умолчанию для этого используются столбцы первичного ключа, но при определении таблицы можно указать и другие способы:

- использовать уникальный индекс (ALTER TABLE … REPLICA IDENTITY USING INDEX …);

- использовать все столбцы (REPLICA IDENTITY FULL).

- вообще отказаться от поддержки репликации для некоторых таблиц (REPLICA IDENTITY NOTHING, такой режим по умолчанию установлен для таблиц системного каталога).

Поскольку таблицы на поставщике и на подписчике могут изменяться независимо друг от друга, при вставке новых версий строк возможно возникновение конфликта — нарушение ограничения целостности.
В этом случае процесс применения записей приостанавливается до тех пор, пока конфликт не будет разрешен вручную. Можно либо исправить данные на подписчике так, чтобы конфликт не происходил, либо отменить применение части записей.

https://postgrespro.ru/docs/postgresql/12/logical-replication-conflicts

https://postgrespro.ru/docs/postgresql/12/sql-altertable

Конфликты

Сейчас на двух серверах в таблице test находятся разные строки:

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

```
 id | descr
----+-------
  1 | Раз
  2 | Два
  3 | Три
(3 rows)
```

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

```
 id | descr
----+--------
  1 | Раз
  2 | Два
  3 | Три
  4 | Четыре
(4 rows)
```

Что произойдет, если публикующий сервер добавит строку, которая уже есть на сервере-подписчике?

```
=> INSERT INTO test VALUES (4, 'Четыре-бис');
```

```
INSERT 0 1
```

Столбец id — первичный ключ, поэтому дублирующее значение не может быть вставлено и логическая репликация остановится. Фактически, процесс logical replication worker будет периодически перезапускаться, проверяя, не устранен ли конфликт. Поэтому информация в pg_stat_subscription пропадает:

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

```
-[ RECORD 1 ]---------+---------
subid                 | 346020
subname               | test_sub
pid                   |
relid                 |
received_lsn          |
last_msg_send_time    |
last_msg_receipt_time |
latest_end_lsn        |
latest_end_time       |
```

Чтобы разрешить этот конфликт, надо удалить конфликтующую строку из таблицы на сервере-подписчике:

```
=> DELETE FROM test WHERE id = 4;
```

```
DELETE 1
```

...и немного подождать.

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

```
 id |   descr
----+------------
  1 | Раз
  2 | Два
  3 | Три
  4 | Четыре-бис
(4 rows)
```

Данные появились, репликация восстановлена.

Триггеры на подписчике

При обычной работе

в обслуживающих процессах (*session_replication_role* = origin или local)

ENABLE TRIGGER

ENABLE ALWAYS TRIGGER

При применении реплицированных изменений

в процессе logical replication worker (*session_replication_role* = replica)

ENABLE REPLICA TRIGGER

ENABLE ALWAYS TRIGGER

только на уровне строк (за исключением начальной синхронизации)

Есть особенности в срабатывании триггеров на сервере-подписчике.

При применении изменений, полученных из публикации, обычные триггеры работать не будут. Это удобно, если на обоих серверах созданы одинаковые таблицы с одинаковым набором триггеров: в таком случае триггер уже отработал на публикующем сервере, и его не надо выполнять на подписчике.

Чтобы триггер все-таки срабатывал, его необходимо разрешить как репликационный (ALTER TABLE … ENABLE REPLICA TRIGGER) или универсальный (ENABLE ALWAYS TRIGGER).

Изменения реплицируются построчно, поэтому работают только триггеры на таблицах на уровне строк (FOR EACH ROW), но не на уровне оператора (FOR EACH STATEMENT). Исключение составляет начальная синхронизация — при ней срабатывают и триггеры на уровне строк, и на уровне оператора.

При обычной работе срабатывают «обычные» триггеры (просто созданные командой CREATE TRIGGER, или явно разрешенные командой ALTER TABLE … ENABLE TRIGGER) и универсальные триггеры (ENABLE ALWAYS TRIGGER).

Сервер отличает обычный процесс от процесса, применяющего реплицированные изменения, с помощью параметра *session_replication_role*. По умолчанию параметр равен origin, но процесс logical replication worker устанавливает его в replica.

https://postgrespro.ru/docs/postgresql/12/sql-altertable

Триггеры на подписчике

Попробуем создать триггер на сервере-подписчике.

```
=> CREATE FUNCTION change_descr() RETURNS trigger AS $$
BEGIN
  NEW.descr := NEW.descr || ' (получено из публикации)';
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER test_before_row
BEFORE INSERT OR UPDATE ON test
FOR EACH ROW
EXECUTE FUNCTION change_descr();
```

```
CREATE TRIGGER
```

Добавляем строку на публикующем сервере:

```
=> INSERT INTO test VALUES (5, 'Пять');
```

```
INSERT 0 1
```

Что получится?

```
=> SELECT * FROM test WHERE id = 5;
```

```
 id | descr
----+-------
  5 | Пять
(1 row)
```

Обычный триггер не срабатывает при применении изменений из публикации, зато он будет срабатывать при операциях на подписчике:

```
=> INSERT INTO test VALUES (6, 'Шесть');
```

```
INSERT 0 1
```

```
=> SELECT * FROM test WHERE id = 6;
```

```
 id |             descr
----+--------------------------------
  6 | Шесть (получено из публикации)
(1 row)
```

Допустим, нам нужно, чтобы триггер срабатывал только для изменений, происходящих при репликации. Для этого явно включим его как репликационный:

```
=> ALTER TABLE test ENABLE REPLICA TRIGGER test_before_row;
```

```
ALTER TABLE
```

```
=> INSERT INTO test VALUES (7, 'Семь');
```

```
INSERT 0 1
```

```
=> INSERT INTO test VALUES (8, 'Восемь');
```

```
INSERT 0 1
```

Проверим:

```
=> SELECT * FROM test WHERE id >= 7;
```

```
 id |             descr
----+-------------------------------
  7 | Семь (получено из публикации)
  8 | Восемь
(2 rows)
```

Теперь триггер работает только при применении реплицированных изменений. Чтобы триггер срабатывал и при локальных изменениях, нужно установить режим ALWAYS TRIGGER.

Удаление подписки

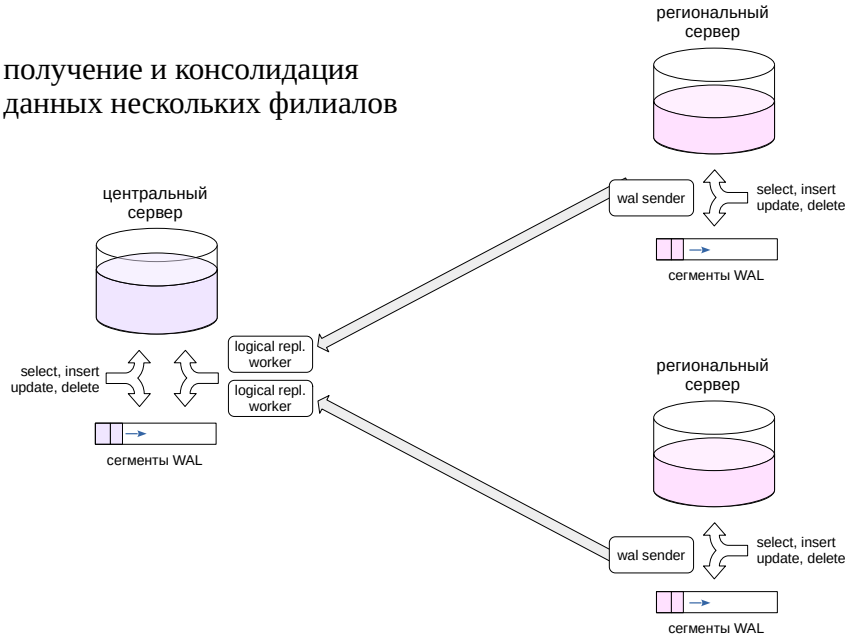
Если репликация больше не нужна, важно удалить подписку. Иначе на публикующем сервере останется открытым репликационный слот, а это приведет к тому, что публикующий сервер не сможет удалять сегменты WAL — через некоторое время место на диске закончится и сервер аварийно остановится.

```
=> DROP SUBSCRIPTION test_sub;
```

```
NOTICE:  dropped replication slot "test_sub" on publisher
DROP SUBSCRIPTION
```

Консолидация

получение и консолидация данных нескольких филиалов



Логическая репликация, как и физическая, позволяет решать разные задачи.

Например, пусть имеются несколько региональных филиалов, каждый из которых работает на собственном сервере PostgreSQL, и необходимо консолидировать часть данных на центральном сервере.

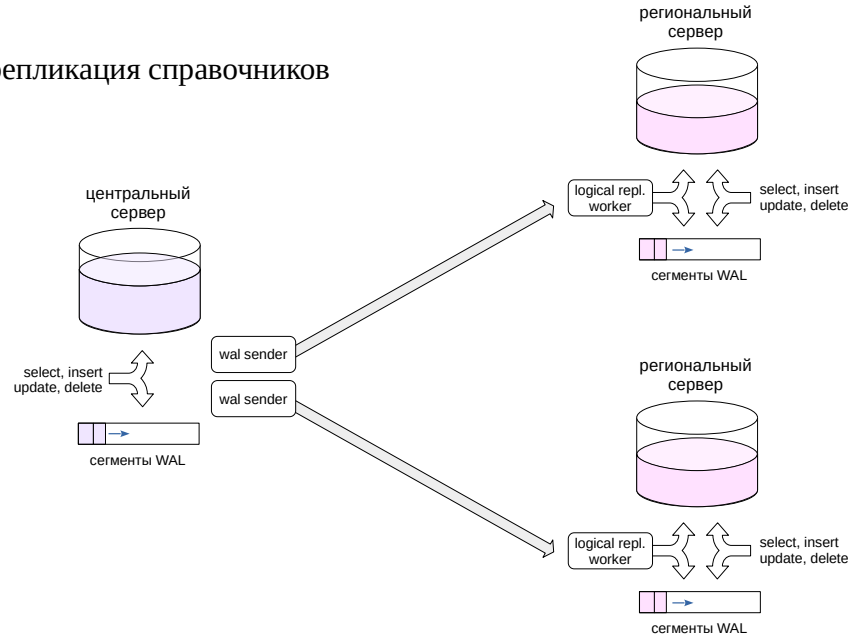
Для решения на региональных серверах создаются публикации необходимых данных. Центральный сервер подписывается на эти публикации. Полученные данные можно обрабатывать (например, приводить к единому виду) с помощью триггеров на стороне центрального сервера.

Поскольку репликация основана на передаче данных через слот, между серверами необходимо более или менее постоянное соединение, так как во время разрыва соединения региональные серверы будут вынуждены сохранять файлы журнала.

Есть множество особенностей такого процесса и с точки зрения бизнес-логики, требующих всестороннего изучения. В ряде случаев может оказаться проще передавать данные пакетно раз в определенный интервал времени.

Общие справочники

репликация справочников

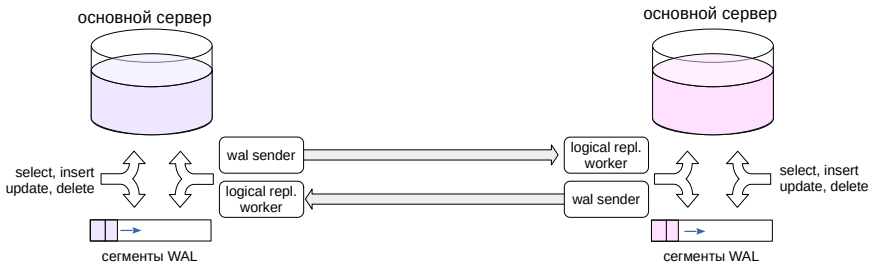


Другая задача: на центральном сервере поддерживаются справочники, актуальные версии которых должны быть доступны на региональных серверах.

В этом случае схему надо развернуть наоборот: центральный сервер публикует изменения, а региональные серверы подписываются на эти обновления.

Мастер-мастер

кластер, в котором данные могут изменять несколько серверов
(дело будущего)



Задача: обеспечить надежное хранение данных на нескольких серверах с возможностью записи на любом сервере (что полезно, например, для геораспределенной системы).

Для решения можно использовать двунаправленную репликацию, передавая изменения в одних и тех же таблицах от одного сервера к другому и обратно.

Сразу заметим, что в настоящее время средства, доступные в PostgreSQL 12, не позволяют этого реализовать, но со временем эта возможность скорее всего появится в ядре. В настоящее время более развитые средства предоставляют расширения pg_logical (https://www.2ndquadrant.com/en/resources/pglogical/) и BDR (https://www.2ndquadrant.com/en/resources/bdr/).

Конечно, прикладная система должна быть построена таким образом, чтобы избегать конфликтов при изменении данных в одних и тех же таблицах. Например, использовать глобальные уникальные идентификаторы или гарантировать, что разные серверы работают с разными диапазонами ключей.

Надо учитывать, что система мастер-мастер, построенная на логической репликации, не обеспечивает сама по себе выполнение глобальных распределенных транзакций и, следовательно, согласованности данных между серверами.

Ограничения

Postgres PROFESSIONAL

Не реплицируются

- команды DDL
- значения последовательностей
- большие объекты (lo)
- изменения материализованных представлений, сторонних таблиц, секционированных таблиц

Могут вызвать проблемы

- массовые изменения данных
- изменения, сделанные долгими транзакциями

Не поддерживаются

- автоматическое разрешение конфликтов
- двунаправленная репликация одной и той же таблицы

Логическая репликация имеет довольно много ограничений.

Не реплицируются команды DDL — все изменения схемы данных надо переносить вручную. До версии 11 не реплицировалась также команда TRUNCATE. Не реплицируются большие объекты (large objects).

Реплицироваться могут только обычные базовые таблицы. Т. е. нельзя реплицировать материализованные представления, сторонние и секционированные таблицы (но можно — отдельные секции).

Не реплицируются значения последовательностей. Это означает, что если сервер-подписчик добавляет строки в таблицу с суррогатным уникальным ключом, для которой настроена репликация, возможны конфликты. Конфликтов можно избежать, выделяя двум серверам разные диапазоны значений последовательностей, или используя вместо них универсальные уникальные идентификаторы (UUID).

При выполнении на публикующем сервере одной команды SQL, затрагивающей много строк, изменения все равно реплицируются построчно, что приводит к повышенной нагрузке на подписчик.

Текущая реализация не очень хорошо справляется с долгими транзакциями. Они увеличивают нагрузку на публикующий сервер.

И, как уже говорилось, нет возможности автоматического разрешения конфликтов и двунаправленной репликации одних и тех же таблиц.

Эти ограничения уменьшают применимость логической репликации.

https://postgrespro.ru/docs/postgresql/12/logical-replication-restrictions

Итоги

Логическая репликация передает изменения строк отдельных таблиц

- разнонаправленная
- совместимость на уровне протокола

Следует учитывать имеющиеся ограничения

Практика

Мы открываем второй магазин. Он работает независимо от основного (свой склад, свои закупки и продажи), но имеет тот же ассортимент.

1. Разверните второй сервер из резервной копии, как показано в демонстрации. Очистите таблицу операций.
2. Организуйте репликацию справочников книг и авторов из основного магазина во второй. Предполагается, что эти таблицы могут меняться только на основном сервере.

   Убедитесь, что изменения названий книг и т. п. передаются на второй сервер, но изменения наличного количества происходят на двух серверах независимо.

```
student$ psql bookstore2
```

1. Развертывание второго магазина

Для развертывания сервера 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
```

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

```
=> ALTER SYSTEM SET wal_level = logical;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 12 main restart
```

```
student$ psql -p 5433 -d bookstore2
```

Очистим таблицу операций, поскольку второй магазин ведет свою деятельность независимо от основного:

```
=> TRUNCATE TABLE operations;
```

```
TRUNCATE TABLE
```

При выполнении команды TRUNCATE не сработает триггер update_onhand_qty_trigger, поэтому очистим наличное количество вручную:

```
=> SELECT book_id, onhand_qty FROM books LIMIT 5;
```

```
 book_id | onhand_qty
---------+------------
      38 |        100
      57 |         94
      79 |         46
      76 |         60
      66 |         22
(5 rows)
```

```
=> UPDATE books SET onhand_qty = 0;
```

```
UPDATE 96
```

2. Репликация справочников книг и авторов

Все три связанные таблицы должны входить в публикацию. Иначе на второй сервер реплицируются данные с нарушением внешних ключей:

```
student$ psql bookstore2
```

```
=> CREATE PUBLICATION books_pub FOR TABLE books, authors, authorships;
```

```
CREATE PUBLICATION
```

Как только мы настроим репликацию справочников, на второй сервер в том числе будут передаваться и изменения наличного количества (books.onhand_qty). Очевидно, что это неудачное решение: такие изменения не нужны второму серверу, наличное количество вычисляется на нем независимо от основного сервера.

Следовало бы поместить наличное количество в отдельную таблицу, изменив соответствующим образом триггер update_onhand_qty_trigger и интерфейсную функцию get_catalog.

Другой способ — отменять нежелательные обновления с помощью триггера. Он не требует изменения остального кода системы, но не устраняет бесполезную пересылку данных по сети.

```
=> CREATE FUNCTION public.keep_onhand_qty() RETURNS trigger
AS $$
BEGIN
    NEW.onhand_qty := OLD.onhand_qty;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER keep_onhand_qty_trigger
BEFORE UPDATE ON public.books
FOR EACH ROW
WHEN (NEW.onhand_qty IS DISTINCT FROM OLD.onhand_qty)
EXECUTE FUNCTION public.keep_onhand_qty();
```

```
CREATE TRIGGER
```

Созданный триггер будет срабатывать только при репликации и не будет мешать обновлять наличное количество при нормальной работе:

```
=> ALTER TABLE books ENABLE REPLICA TRIGGER keep_onhand_qty_trigger;
```

```
ALTER TABLE
```

Создаем подписку. Поскольку текущее состояние справочников уже попало на второй сервер из резервной копии, отключаем начальную синхронизацию данных:

```
=> CREATE SUBSCRIPTION books_sub
CONNECTION 'host=localhost port=5432 user=postgres password=postgres dbname=bookstore2'
PUBLICATION books_pub WITH (copy_data = false);
```

```
NOTICE:  created replication slot "books_sub" on publisher
CREATE SUBSCRIPTION
```

Практика

1. На двух серверах ведутся однотипные таблицы бухгалтерских транзакций. Требуется консолидировать все записи в ту же таблицу на одном из серверов (например, для целей построения общей отчетности).
   Предложите способ различить «свои» и «чужие» записи и не допустить конфликты при логической репликации.
   Реализуйте предложенную схему.
2. Настройте логическую репликацию произвольной таблицы в другую на том же самом сервере.

1. Исходите из следующего вида таблицы транзакций:

```
CREATE TABLE transactions (
  trx_id     bigint PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  debit_acc  integer NOT NULL,
  credit_acc integer NOT NULL,
  amount     numeric(15,2) NOT NULL
);
```

2. Если попробовать выполнить обычные действия, команда CREATE SUBSCRIPTION «повиснет». Внимательно изучите документацию: https://postgrespro.ru/docs/postgresql/12/sql-createsubscription

1. Консолидация

На основном сервере установим необходимый уровень журнала:

```
=> ALTER SYSTEM SET wal_level = logical;
```

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 12 main restart
```

Создадим таблицу транзакций:

```
student$ psql
```

```
=> CREATE DATABASE ext_repl_logical;
```

```
CREATE DATABASE
```

```
=> \c ext_repl_logical
```

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

```
=> CREATE TABLE transactions (
    trx_id     bigint PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    debit_acc  integer NOT NULL,
    credit_acc integer NOT NULL,
    amount     numeric(15,2) NOT NULL
);
```

```
CREATE TABLE
```

Чтобы отличать строки разных серверов, понадобится дополнительный столбец. Например:

```
=> ALTER TABLE transactions ADD origin text NOT NULL DEFAULT 'A';
```

```
ALTER TABLE
```

Чтобы не допустить конфликта значений первичного ключа, можно выделить каждому из серверов свой диапазон значений. Например, первому серверу — нечетные, второму — четные. Имя последовательности, которая используется для генерации уникальных номеров, можно узнать с помощью функции:

```
=> SELECT pg_get_serial_sequence('transactions','trx_id');
```

```
     pg_get_serial_sequence
-------------------------------
 public.transactions_trx_id_seq
(1 row)
```

```
=> ALTER SEQUENCE transactions_trx_id_seq
    START WITH 1 INCREMENT BY 2;
```

```
ALTER SEQUENCE
```

Недостатком такой схемы является то, что при появлении третьего сервера ее придется перестраивать.

Другой вариант — использовать для первичного ключа универсальные уникальные идентификаторы, представленные типом данных uuid. Для этого можно использовать расширение pgcrypto или uuid-ossp (а в версии 13 появится встроенная функция get_random_uuid):

```
=> CREATE EXTENSION pgcrypto;
```

```
CREATE EXTENSION
```

```
=> SELECT gen_random_uuid();
```

```
           gen_random_uuid
--------------------------------------
 559f3f3d-769e-4f80-baba-83debf11239e
(1 row)
```

Однако тип uuid занимает 16 байт (bigint — только 8) и генерация нового значения происходит относительно долго.

Еще один вариант — создать составной первичный ключ:

```
PRIMARY KEY (trx_id, origin)
```

В этом случае конфликтов гарантированно не будет, но индекс получится больше.

Теперь развернем второй сервер из резервной копии, как показано в демонстрации.

```
student$ sudo pg_ctlcluster 12 replica stop

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

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

student$ sudo pg_ctlcluster 12 replica start

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

Заполним таблицу транзакций тестовыми данными — разными на разных серверах.

```
=> INSERT INTO transactions(debit_acc, credit_acc, amount)
    SELECT trunc(random()*3),     -- 0..2
           trunc(random()*3) + 3, -- 3..5
           random()*100000
    FROM generate_series(1,10000);

INSERT 0 10000

=> SELECT * FROM transactions LIMIT 5;

 trx_id | debit_acc | credit_acc |  amount  | origin
--------+-----------+------------+----------+--------
      1 |         0 |          5 | 24720.87 | A
      3 |         0 |          5 | 96963.51 | A
      5 |         0 |          5 | 39171.60 | A
      7 |         1 |          3 | 88000.10 | A
      9 |         1 |          5 | 98721.96 | A
(5 rows)
```

На втором сервере не забудем предварительно заменить значение по умолчанию для столбца origin и настройки последовательности:

```
=> ALTER TABLE transactions ALTER origin SET DEFAULT 'B';

ALTER TABLE

=> ALTER SEQUENCE transactions_trx_id_seq
    START WITH 2 INCREMENT BY 2 RESTART;

ALTER SEQUENCE

=> INSERT INTO transactions(debit_acc, credit_acc, amount)
    SELECT trunc(random()*3) + 3, -- 3..5
           trunc(random()*3),     -- 0..2
           random()*100000
    FROM generate_series(1,10000);

INSERT 0 10000

=> SELECT * FROM transactions LIMIT 5;

 trx_id | debit_acc | credit_acc |  amount  | origin
--------+-----------+------------+----------+--------
      2 |         5 |          2 |  8951.34 | B
      4 |         5 |          1 | 23762.16 | B
      6 |         4 |          2 | 15216.83 | B
      8 |         5 |          0 | 36265.56 | B
     10 |         3 |          2 | 18941.39 | B
(5 rows)
```

Настроим репликацию.

```
=> CREATE PUBLICATION trx_pub FOR TABLE transactions;

CREATE PUBLICATION

=> CREATE SUBSCRIPTION trx_sub
CONNECTION 'host=localhost port=5433 user=postgres password=postgres dbname=ext_repl_logical'
PUBLICATION trx_pub;

NOTICE:  created replication slot "trx_sub" on publisher
CREATE SUBSCRIPTION
```

После небольшой паузы убедимся, что данные второго сервера успешно реплицированы:

```
=> SELECT origin, count(*) FROM transactions GROUP BY origin;
```

```
 origin | count
--------+-------
 B      | 10000
 A      | 10000
(2 rows)
```

Удалим подписку, чтобы завершить задание:

```
=> DROP SUBSCRIPTION trx_sub;
```

```
NOTICE:  dropped replication slot "trx_sub" on publisher
DROP SUBSCRIPTION
```

```
=> \q
```

2. Репликация на одном сервере

Создадим публикацию:

```
=> CREATE PUBLICATION trx_pub FOR TABLE transactions;
```

```
CREATE PUBLICATION
```

Создадим вторую базу данных на том же сервере, и таблицу в ней:

```
=> CREATE DATABASE ext_repl_logical2;
```

```
CREATE DATABASE
```

```
student$ psql ext_repl_logical2
```

```
=> CREATE TABLE transactions (
    trx_id     bigint PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    debit_acc  integer NOT NULL,
    credit_acc integer NOT NULL,
    amount     numeric(15,2) NOT NULL,
    origin     text NOT NULL
);
```

```
CREATE TABLE
```

Команда CREATE SUBSCRIPTION по умолчанию создает слот репликации, а для этого требуется дождаться, пока завершатся все транзакции, активные на момент начала создания слота. Одной из таких транзакция является та, в которой выполняется команда CREATE SUBSCRIPTION, что приводит к бесконечному ожиданию.

Решением является создание слота вручную и указание его имени при создании подписки:

```
=> SELECT pg_create_logical_replication_slot('trx_slot','pgoutput');
```

```
 pg_create_logical_replication_slot
------------------------------------
 (trx_slot,1/D52CF938)
(1 row)
```

```
=> CREATE SUBSCRIPTION trx_sub
CONNECTION 'host=localhost port=5432 user=postgres password=postgres dbname=ext_repl_logical'
PUBLICATION trx_pub WITH (slot_name = trx_slot, create_slot = false);
```

```
CREATE SUBSCRIPTION
```

Проверим:

```
=> SELECT count(*) FROM transactions;
```

```
  count
-------
 20000
(1 row)
```

Репликация работает.

В завершение удалим подписку (слот будет удален автоматически):

```
=> DROP SUBSCRIPTION trx_sub;
```

```
NOTICE:  dropped replication slot "trx_slot" on publisher
DROP SUBSCRIPTION
```