

# Репликация

## Обзор логической репликации



16

### 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](mailto: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.

Логическая репликация

Публикации и подписчики

Уровни журнала

Обнаружение и разрешение конфликтов

Особенности работы триггеров

Задачи, решаемые с помощью логической репликации

Ограничения

## 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-подписка: поток данных возможен в обе стороны  
требуется совместимость на уровне протокола  
возможна выборочная репликация отдельных таблиц

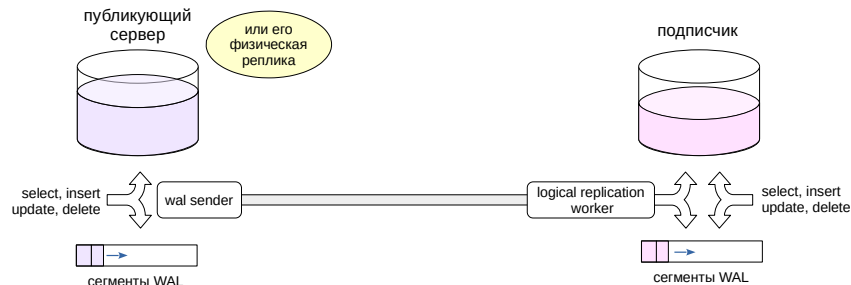
При физической репликации только один сервер (мастер) выполняет изменения и генерирует журнальные записи. Остальные серверы (реплики) только читают журнальные записи мастера и применяют их.

При логической репликации все серверы работают в обычном режиме, могут изменять данные и генерировать собственные журнальные записи. Любой 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.

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

<https://postgrespro.ru/docs/postgresql/16/logical-replication>

# Логическая репликация



На публикующем сервере процесс `wal_sender` формирует журнальные записи, отражающие изменения в публикуемых данных, а фоновый процесс `logical replication worker` на сервере-подписчике получает информацию от публикующего сервера и применяет ее.

Подписчик может получать изменения не только напрямую с публикующего сервера, но и с его физической реплики. В этом случае отправку изменений подписчику выполняет процесс `wal_sender` физической реплики. Такая схема работы полезна для уменьшения нагрузки на публикующий сервер.

<https://postgrespro.ru/docs/postgresql/16/logical-replication-publication>

<https://postgrespro.ru/docs/postgresql/16/logical-replication-subscription>

## Публикация

- включает одну или несколько таблиц базы данных
- можно указать столбцы и фильтр для строк
- обрабатывает команды INSERT, UPDATE, DELETE, TRUNCATE
- выдает изменения построчно после фиксации транзакции
- использует слот логической репликации

## Подписка

- получает и применяет изменения
- возможна начальная синхронизация
- без разбора, переписывания и планирования — сразу выполнение
- возможны конфликты с локальными данными

Логическая репликация использует модель «публикация-подписка».

На одном сервере создается *публикация*, которая может включать ряд таблиц одной базы данных. Начиная с версии 15 можно опубликовать только часть данных таблицы: указать набор столбцов и задать условие фильтрации для строк. Другие серверы могут *подписываться* на эту публикацию: получать и применять изменения.

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

После фиксации транзакции информация о строках, которые она изменила, извлекается из записей WAL на сервере публикации — эта процедура называется *логическим декодированием*. Сформированные сообщения пересылаются подписке по протоколу репликации в формате, независимом от платформы и версии сервера.

Применение изменений происходит без выполнения команд SQL и связанных с этим накладных расходов на разбор и планирование. С другой стороны, результат выполнения одной команды SQL может превратиться в множество однострочных изменений.

<https://postgrespro.ru/docs/postgresql/16/logical-replication>

Параметр *wal\_level*

<b>minimal</b>	<	<b>replica</b>	<	<b>logical</b>
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 сформировать сообщения об изменениях на уровне табличных строк, в журнал необходимо помещать дополнительную информацию, в частности идентификаторы затрагиваемых при репликации строк и сообщения о фактах изменения определений таблиц и типов данных. Это позволяет подписчику в любой момент знать структуру объектов, участвующих в репликации.

Такой расширенный уровень журнала называется **logical**. Поскольку по умолчанию используется уровень *replica*, для логической репликации потребуется изменить значение параметра *wal\_level* на публикующем сервере.

<https://postgrespro.ru/docs/postgresql/16/protocol-logical-replication#PROTOCOL-LOGICAL-MESSAGES-FLOW>

## Логическая репликация

Пусть на первом сервере имеется таблица:

```
=> CREATE DATABASE replica_overview_logical_dev;
```

```
CREATE DATABASE
```

```
=> \c replica_overview_logical_dev
```

You are now connected to database "replica\_overview\_logical\_dev" as user "student".

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

```
CREATE TABLE
```

Склонируем кластер с помощью резервной копии, как мы делали в теме «Обзор физической репликации», но команде pg\_basebackup не будем передавать ключ -R, поскольку нам потребуется независимый сервер, а не реплика.

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

Если второй сервер работает, остановим его

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

Cluster is not running.

Перемещаем резервную копию в каталог данных второго сервера, поменяв владельца файлов:

```
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 второй сервер:

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

Получили два независимых сервера, на каждом из них есть пустая таблица test. Добавим в таблицу на первом сервере пару строк:

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

```
INSERT 0 2
```

Теперь мы хотим настроить между серверами логическую репликацию. Для этого понадобится дополнительная информация в журнале первого сервера:

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

```
ALTER SYSTEM
```

```
student$ sudo pg_ctlcluster 16 main restart
```

На первом сервере создаем публикацию:

```
student$ psql -d replica_overview_logical_dev
```

```
=> CREATE PUBLICATION test_pub FOR TABLE test;
```

```
CREATE PUBLICATION
```

```
=> \dRp+
```

```

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

На втором сервере подписываемся на эту публикацию:

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

```
=> CREATE SUBSCRIPTION test_sub
CONNECTION 'port=5432 user=student dbname=replica_overview_logical_dev'
PUBLICATION test_pub;
```

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

При создании подписки на сервере публикации автоматически создался слот репликации — объект, который помнит, какие изменения получил подписчик, и в случае разрыва соединения позволяет возобновить передачу с нужного места.

```
=> \dRs
```

```

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

Проверяем репликацию:

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

```
INSERT 0 1
```

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

```

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

Состояние подписки можно посмотреть в представлении:

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

```

-[ RECORD 1 ]-----+-----
subid          | 24752
subname        | test_sub
pid            | 170660
leader_pid     |
relid         |
received_lsn   | 0/7006680
last_msg_send_time | 2024-08-14 11:15:09.685458+03
last_msg_receipt_time | 2024-08-14 11:15:09.686422+03
latest_end_lsn  | 0/7006680
latest_end_time | 2024-08-14 11:15:09.685458+03
```

К процессам сервера-подписчика добавился logical replication worker (его номер указан в pg\_stat\_subscription.pid):

```
student$ ps -o pid,command --ppid 170363
```

```

PID COMMAND
170364 postgres: 16/replica: checkpointer
170365 postgres: 16/replica: background writer
170370 postgres: 16/replica: walwriter
170371 postgres: 16/replica: autovacuum launcher
170372 postgres: 16/replica: logical replication launcher
170627 postgres: 16/replica: student replica_overview_logical_dev [local] idle
170660 postgres: 16/replica: logical replication apply worker for subscription 24752
```



## 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

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 (replica identity): использовать уникальный индекс или использовать все столбцы. Можно вообще отказаться от поддержки репликации для некоторых таблиц (по умолчанию не реплицируются таблицы системного каталога).

Поскольку таблицы на сервере публикации и сервере подписки могут изменяться независимо друг от друга, при вставке новых версий строк возможны конфликты — нарушения ограничений целостности. В этом случае процесс применения записей приостанавливается до тех пор, пока конфликт не будет разрешен вручную.

<https://postgrespro.ru/docs/postgresql/16/sql-altertable>

<https://postgrespro.ru/docs/postgresql/16/logical-replication-conflicts>

<https://postgrespro.ru/docs/postgresql/16/sql-altersubscription>

## Конфликты

Локальные изменения на подписчике не запрещаются, вставим строку в таблицу на втором сервере.

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

```
INSERT 0 1
```

Если теперь строку с таким же значением первичного ключа вставить на первом сервере, при ее применении на стороне подписки произойдет конфликт.

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

```
INSERT 0 1
```

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

```
INSERT 0 1
```

Подписка не может применить изменение, репликация остановилась.

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

```
-[ RECORD 1 ]-----+-----
subid          | 24752
subname        | test_sub
pid            |
leader_pid     |
relid          |
received_lsn   |
last_msg_send_time |
last_msg_receipt_time |
latest_end_lsn |
latest_end_time |
```

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

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

Чтобы разрешить конфликт, удалим строку на втором сервере и немного подождем...

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

```
DELETE 1
```

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

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

Репликация возобновилась.

## 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)

Срабатывание триггеров на сервере-подписчике имеет некоторые особенности.

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.

Если нужно, чтобы триггер все-таки срабатывал, его необходимо определить как репликационный (`ALTER TABLE ... ENABLE REPLICA TRIGGER`) или универсальный (`ENABLE ALWAYS TRIGGER`).

Изменения реплицируются построчно, поэтому работают только триггеры таблиц на уровне строк (`FOR EACH ROW`), но не на уровне оператора (`FOR EACH STATEMENT`). Исключение составляет начальная синхронизация — при ней срабатывают триггеры и 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 сеанса *session\_replication\_role*. По умолчанию параметр имеет значение origin, а процесс logical replication worker задает ему значение replica.

<https://postgrespro.ru/docs/postgresql/16/sql-altertable>

## Триггеры на подписчике

Попробуем создать триггер на сервере-подписчике.

```
=> CREATE FUNCTION change_descr() RETURNS trigger AS $$
BEGIN
    NEW.descr := NEW.descr || ' - триггер сработал';
    RETURN NEW;
END;
$$ 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 (6, 'Шесть');
```

INSERT 0 1

Что получится?

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

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

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

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

INSERT 0 1

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

```
 id |          descr
-----+-----
   7 | Семь - триггер сработал
(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 (8, 'Восемь');
```

INSERT 0 1

```
=> INSERT INTO test VALUES (9, 'Девять');
```

INSERT 0 1

Проверим:

```
=> SELECT * FROM test WHERE id >= 8;
```

```
 id |          descr
-----+-----
   8 | Восемь - триггер сработал
   9 | Девять
(2 rows)
```

Теперь триггер работает только при применении реплицированных изменений.

Если необходимо добиться срабатывания триггера и при репликации, и при локальных изменениях, нужно определить его как ALWAYS TRIGGER. А чтобы различить эти изменения, используем параметр session\_replication\_role:

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

ALTER TABLE

=> CREATE OR REPLACE FUNCTION change_descr() RETURNS trigger AS $$
BEGIN
    NEW.descr := NEW.descr || ' (' || current_setting('session_replication_role') || ')';
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION
```

Вставим еще по одной строке на каждом сервере и проверим, что получилось:

```
=> INSERT INTO test VALUES (10, 'Десять');
```

```
INSERT 0 1
```

```
=> INSERT INTO test VALUES (11, 'Одиннадцать');
```

```
INSERT 0 1
```

```
=> SELECT * FROM test WHERE id >= 10;
```

```
 id |      descr
----+-----
 10 | Десять (replica)
 11 | Одиннадцать (origin)
(2 rows)
```

Заметим, что проверки внешних ключей в PostgreSQL реализованы посредством триггеров, поэтому присвоение параметру `session_replication_role` значения `replica` отключает эти проверки, а это может привести к нарушению согласованности данных.

## Удаление подписки

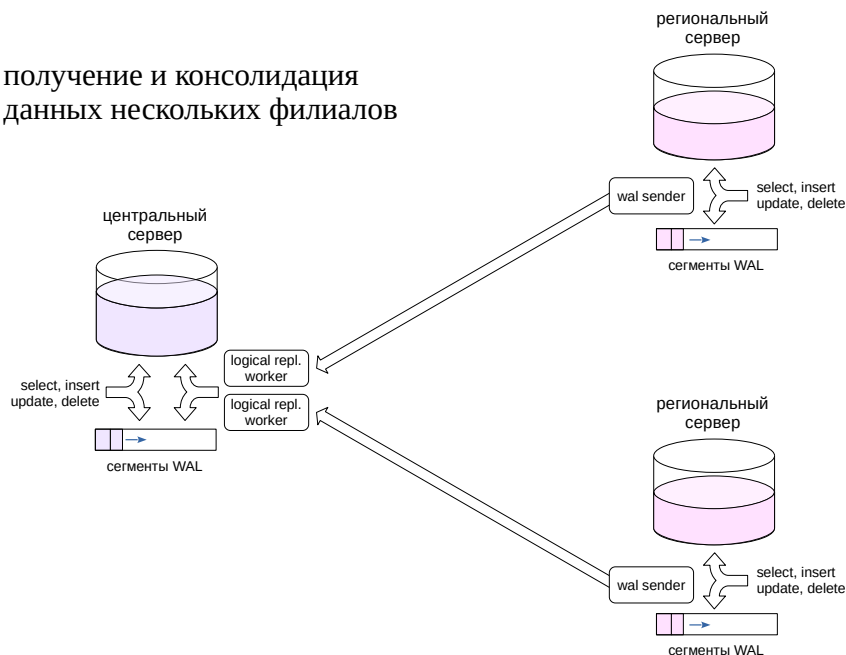
Если репликация больше не нужна, надо аккуратно удалить подписку — иначе на публикующем сервере останется открытым репликационный слот.

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

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

# Задачи: консолидация

получение и консолидация  
данных нескольких филиалов



12

Рассмотрим несколько задач, которые можно решить с помощью логической репликации.

Пусть имеются несколько региональных филиалов, каждый из которых работает на собственном сервере PostgreSQL. Задача состоит в консолидации части данных на центральном сервере.

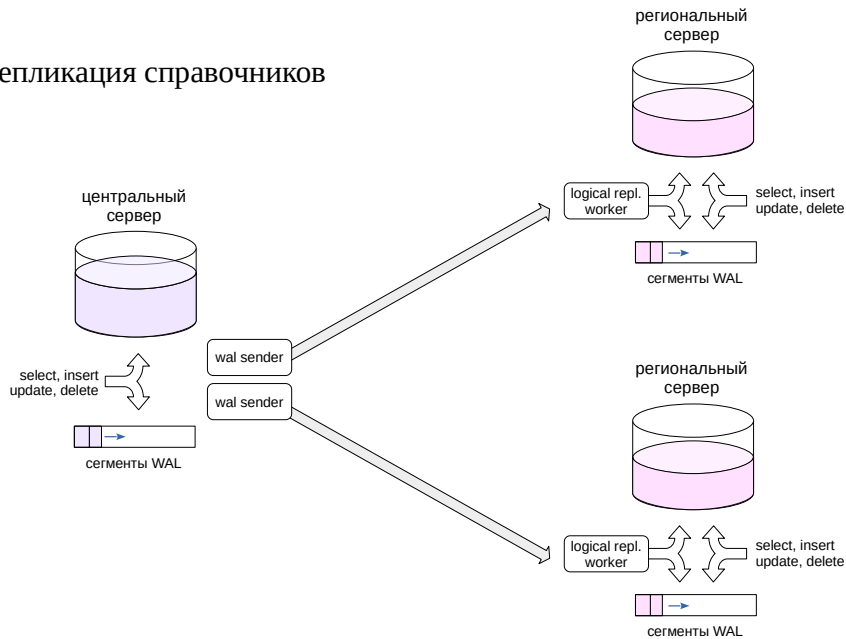
Для решения задачи на региональных серверах создаются публикации необходимых данных. Центральный сервер подписывается на эти публикации. Полученные данные можно обрабатывать с помощью триггеров на стороне центрального сервера (например, приводя данные к единому виду).

С точки зрения бизнес-логики есть множество особенностей, требующих внимания. В каких-то случаях может быть проще периодически передавать данные в пакетном режиме.

На иллюстрации: на центральном сервере работают два процесса приема журналов, по одному на каждую подписку.

# Задачи: справочники

## репликация справочников



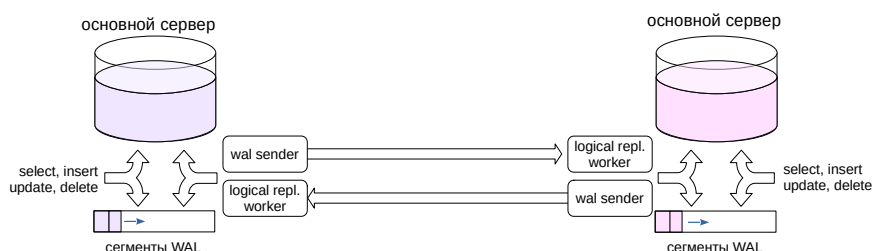
13

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.

# Задачи: мастер-мастер

кластер, в котором данные могут изменять несколько серверов



14

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.

Двунаправленная репликация появилась в PostgreSQL 16. Система умеет различать источники репликации (origin), и при создании подписки можно указать, чтобы с публикующего сервера не отправлялись изменения, помеченные определенным источником.

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, о согласованности данных между серверами придется позаботиться отдельно.



## Не реплицируются

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

## Могут вызвать проблемы

- массовые изменения данных

## Не поддерживаются

- автоматическое разрешение конфликтов

Логическая репликация имеет довольно много ограничений.

Не реплицируются команды DDL — все изменения схемы данных надо переносить вручную.

Реплицироваться могут только обычные базовые таблицы и секционированные таблицы. То есть нельзя реплицировать отношения другого рода, такие как представления, материализованные представления и внешние таблицы.

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).

Изменения реплицируются построчно, поэтому выполнение на публикаторе команды SQL, затрагивающей много строк, приведет к повышенной нагрузке на подписчик.

И, как уже говорилось, нет возможности автоматического разрешения конфликтов.

Эти ограничения уменьшают применимость логической репликации.

<https://postgrespro.ru/docs/postgresql/16/logical-replication-restrictions>

Логическая репликация передает изменения строк  
отдельных таблиц

- разнонаправленная

- совместимость на уровне протокола

Модель публикация — подписчик

Следует учитывать имеющиеся ограничения



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.

## 1. Развертывание второго магазина

Для развертывания сервера выполняем те же команды, что и в демонстрации:

```
student$ pg_basebackup --pgdata=/home/student/backup --checkpoint=fast
student$ sudo pg_ctlcluster 16 replica stop
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
=> ALTER SYSTEM SET wal_level = logical;
ALTER SYSTEM
=> \q
student$ sudo pg_ctlcluster 16 main restart
student$ psql -d bookstore2
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
|-----+-----
|      32 |         48
|      13 |         38
|      90 |         26
|      35 |         44
|      94 |         62
| (5 rows)
```

```
| => UPDATE books SET onhand_qty = 0;
|
| UPDATE 100
```

## 2. Репликация справочников книг и авторов

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

```
=> 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
```

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 в обратном направлении. Проверьте работу двунаправленной репликации.

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. Чтобы публикация не отправляла реплицированные изменения, задайте подписке параметр `origin = none`.

## 1. Консолидация

На первом (публикующем) сервере установим необходимый уровень журнала:

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

ALTER SYSTEM

```
student$ sudo pg_ctlcluster 16 main restart
```

Создадим таблицу транзакций:

```
student$ psql
```

```
=> CREATE DATABASE replica_overview_logical_dev;
```

CREATE DATABASE

```
=> \c replica_overview_logical_dev
```

You are now connected to database "replica\_overview\_logical\_dev" 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 server text NOT NULL DEFAULT current_setting('cluster_name');
```

ALTER TABLE

---

Чтобы не допустить конфликта значений первичного ключа, выделим каждому из серверов свое множество значений: 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 возможный вариант — использовать для первичного ключа универсальные уникальные идентификаторы, представленные типом данных uuid. Для этого можно использовать встроенную функцию `get_random_uuid` или средства из расширений `pgcrypto` и `uuid-oss`:

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

```
      gen_random_uuid
-----
5cdbbf7c-8c67-449a-862b-954ae30db7c7
(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, server)
```

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

Мы используем первый вариант: последовательности, генерирующие разные значения.

---

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

```
student$ pg_basebackup --pgdata=/home/student/backup
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
student$ psql -p 5433 -d replica_overview_logical_dev
```

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	server
1	0	4	78645.62	16/main
3	2	3	75562.02	16/main
5	0	3	84302.40	16/main
7	2	5	65728.81	16/main
9	0	5	16223.62	16/main

(5 rows)

На втором сервере не забудем заменить настройки последовательности:

```
=> 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	server
2	5	1	47190.68	16/replica
4	4	0	95548.63	16/replica
6	3	1	24615.71	16/replica
8	4	1	50371.74	16/replica
10	3	0	64115.22	16/replica

(5 rows)

Настроим репликацию с первого сервера на второй.

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

CREATE PUBLICATION

```
=> CREATE SUBSCRIPTION trx_sub
    CONNECTION 'dbname=replica_overview_logical_dev'
    PUBLICATION trx_pub;
```

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

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

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



server	count
16/main	10000
16/replica	10000

(2 rows)

## 2. Двухнаправленная репликация

Настройка уровня журнала уже скопирована на второй сервер:

```
=> SHOW wal_level;
```

```
wal_level
-----
logical
(1 row)
```

Теперь очистим таблицы и сбросим последовательности

```
=> TRUNCATE TABLE transactions;
```

```
TRUNCATE TABLE
```

```
=> SELECT setval('transactions_trx_id_seq',1);
```

```
setval
-----
1
(1 row)
```

```
=> TRUNCATE TABLE transactions;
```

```
TRUNCATE TABLE
```

```
=> SELECT setval('transactions_trx_id_seq',2);
```

```
setval
-----
2
(1 row)
```

Чтобы публикация не отправляла подписке реплицированные изменения, нужно задать для подписки параметр origin:

```
=> ALTER SUBSCRIPTION trx_sub SET (origin = none);
```

```
ALTER SUBSCRIPTION
```

Теперь настроим репликацию со второго сервера на первый.

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

```
CREATE PUBLICATION
```

```
=> CREATE SUBSCRIPTION trx_sub
CONNECTION 'port=5433 dbname=replica_overview_logical_dev'
PUBLICATION trx_pub
WITH (origin = none, copy_data = false);
```

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

Проверим:

```
=> 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(1,5000);
```

```
INSERT 0 5000
```

```
=> 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(1,5000);
```

```
INSERT 0 5000
```

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

server	count
16/main	5000
16/replica	5000

(2 rows)

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

server	count
16/main	5000
16/replica	5000

(2 rows)

Удалим треть строк на каждом сервере:

```
=> DELETE FROM transactions WHERE trx_id % 3 = 1;
```

DELETE 3333

```
=> DELETE FROM transactions WHERE trx_id % 3 = 2;
```

DELETE 3333

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

server	count
16/main	1667
16/replica	1667

(2 rows)

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

server	count
16/main	1667
16/replica	1667

(2 rows)

Репликация работает.

Удалим подписки, чтобы не оставлять активные слоты репликации.

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

NOTICE: dropped replication slot "trx\_sub" on publisher  
DROP SUBSCRIPTION

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

NOTICE: dropped replication slot "trx\_sub" on publisher  
DROP SUBSCRIPTION