Мифы и реалии «Мультимастера» в архитектуре СУБД PostgreSQL

Привет, Хабр! Недавно мы делали доклад на конференции HighLoad 2023 — «Мифы и реалии Мультимастера в архитектуре СУБД PostgreSQL». Мы — это Павел Конотопов (@kakoka) и Михаил Жилин (@mizhka), сотрудники компании Postgres Professional. Павел занимается архитектурой построения отказоустойчивых кластеров, а Михаил — анализом производительности СУБД. У каждого за плечами более десяти лет опыта в своей области.

Павел Конотопов

Михаил Жилин

Далеко не все читатели Хабра смогли посетить конференцию или посмотреть трансляцию нашего доклада, поэтому мы решили поделиться рассказанным в статье. Порассуждаем о том, как развивалась технология «Мультимастер» в экосистеме PostgreSQL, остановимся на том, что она из себя представляет, на каких внутренних механизмах PostgreSQL основана и как её можно использовать.

Мы также поговорим о том, существует ли «Честный Мультимастер» (само понятие «Честный Мультимастер» достаточно специфично и в основном употребляется в кругу разработчиков), какие реализации у него есть и как его следует применять.

Статья будет сфокусирована на производительности «Мультимастера», ведь если СУБД работает медленней черепахи, то это уже не база данных, а черепаха 🙂

Часть 1. Что такое «Мультимастер»?

Разберёмся с терминологией: что есть что в распределенных системах, таких как кластеры РСУБД?

В мире баз данных несколько узлов СУБД (виртуальных машин или физических серверов) объединены в сущность, которую называют кластер. В кластере каждому узлу присваивается определённая роль, обычно роли бывают двух типов:

- **ведущий сервер** — «master» или «primary», генерирующий поток изменений;
- **ведомый сервер** или **«репликация»** — «slave», «standby» или «реплика», принимающий поток изменения данных от ведущего узла, и применяющий их у себя.

На всех узлах такой системы будет находиться одинаковый набор данных, и изменять данные можно только через один узел — «мастер». Само понятие «мастер» предполагает, что должна быть сущность, которую можно назвать «не-мастер». Если обе эти сущности встречаются в одном кластере, то перед нами — распределённая система.

Таким образом, термин «Мультимастер» означает, что кластер РСУБД будет состоять только из узлов одного типа — «ведущий узел», или «мастер» (например, «мастер №1», «мастер №2», «мастер №3» и т.д.). Изменять данные в таком кластере можно через любой узел, при этом изменения будут применены на всех узлах кластера, следовательно, на всех узлах будет одинаковый набор данных.

Тогда «Мультимастер» выглядит как отказоустойчивое решение для реляционной БД, которое:

- хорошо масштабируется: больше узлов — больше производительность;
- распределение нагрузки между узлами: как для чтения, так и для записи;
- надежно: выход одного из узлов не влияет на доступность БД.

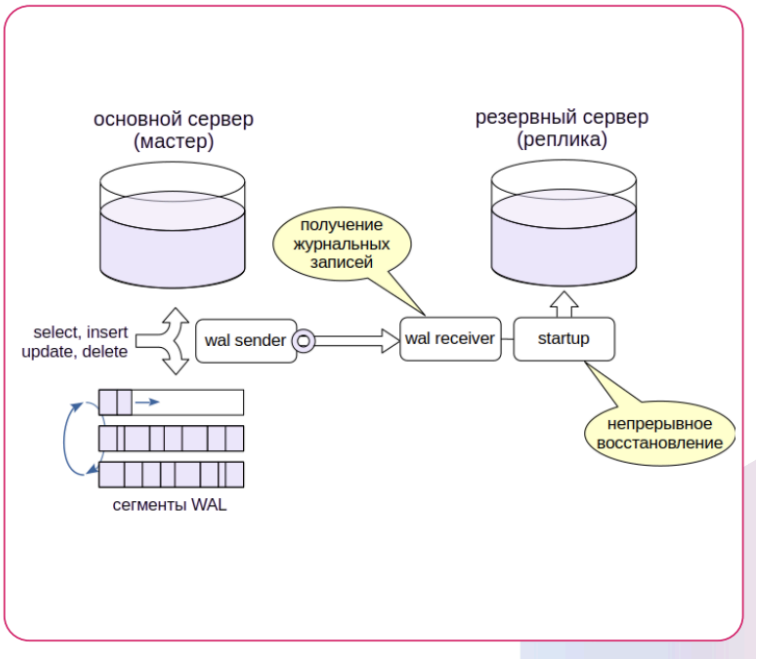
Сама по себе технология «Мультимастер» не нова: на сцене HighLoad про нее сделал хороший доклад делал Илья Космодемьянский в 2017 году.

Однако из большинства докладов можно сделать вывод, что «Мультимастер» для PostgreSQL – технология, обладающая большой сложностью, низкой производительностью, надёжностью и доступностью, но она не годится для промышленной эксплуатации. А если её всё же внедряют в промышленный контур, то у вас явно что-то не так с архитектурой информационной системы. Правда, с 2017 года прошло достаточно много времени, и приведённые выше утверждения успели либо обрасти мифами, либо потеряли свою актуальность.

Чтобы отличить миф от реальности нужно рассмотреть современное состояние «Мультимастера». Прежде всего опишем базовые механизмы PostgreSQL, на которые опирается технология «Мультимастер»[1].

Начнём с **физической** или **потоковой репликации**, которая в PostgreSQL появилась очень давно. Её суть в том, что изменение физических страниц БД ведущего узла передаётся на ведомый. Но есть нюанс: при таком типе репликации передаются **все** изменения **всех** БД экземпляра PostgreSQL.

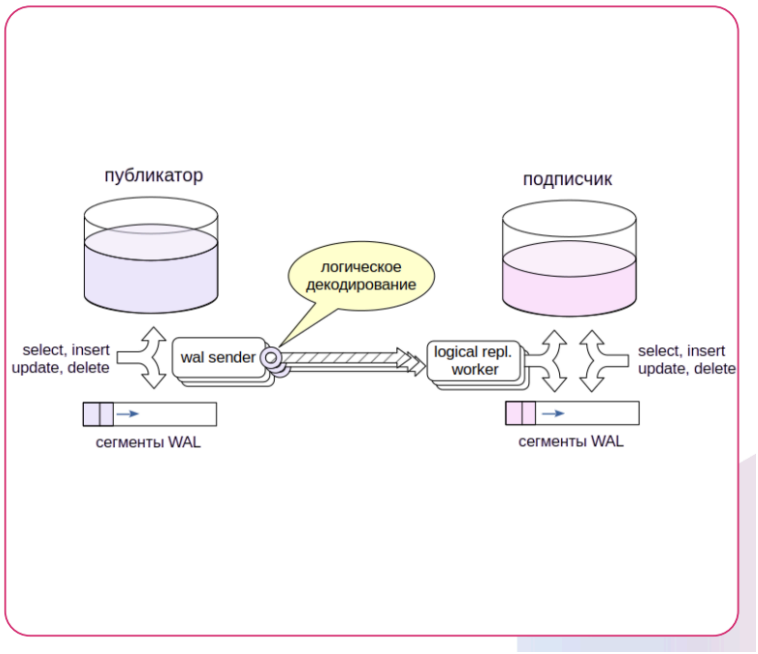
[1] Физическая репликация – слоты репликации для WAL, логическая репликация – логическое декодирование WAL.



На схеме выше изображен кластер БД из двух узлов: с узла «основной сервер (мастер)» на узел «резервный сервер (реплика)» передаётся поток изменения физических страниц БД.

Очевидно, что «Мультимастер» на такой технологии не построить, так как все изменения реплицируются: PostgreSQL на данном уровне репликации не даёт гранулярности в её настройке. Следовательно, запись данных при таком типе репликации возможна только в один узел — ведущий. А хочется использовать все узлы кластера для записи данных, поэтому придумали логическую репликацию.

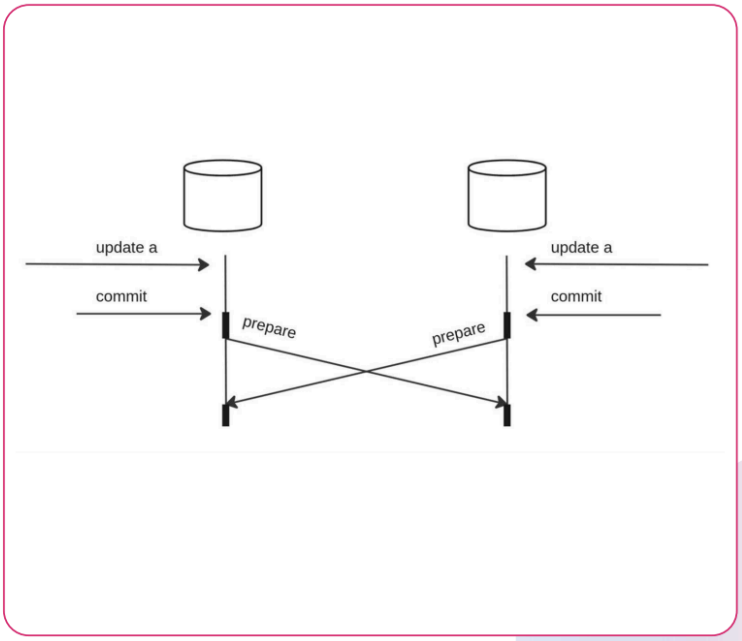
При **логической** или **транзакционной репликации** изменения передаются в виде транзакций, изменяющих данные (INSERT, DELETE, UPDATE). В PostgreSQL логическая репликация появилась в 10-й версии (2017 год). Этот тип репликации более гибкий: в нём реализована модель «публикатор – подписчик», или источник и приёмник репликации, которыми могут быть отдельная база данных, схема или таблица, соответственно можно выбирать, что и куда реплицировать. Но и тут есть особенность: поток изменений такой репликации — **однонаправленный**, на этой технологии полноценный «Мультимастер» тоже не построить.



Но мысль разработчиков сообщества PostgreSQL не стояла на месте! Во времена, когда не было встроенной логической репликации создали расширение pgLogical. А поверх него построили BDR (Bi-Directional Replication) — двунаправленную логическую репликацию с разрешением конфликтов. В более поздних версиях связка pgLogical/BDR превратилась в закрытый коммерческий код (про это будет рассказано ниже).

Следующий виток эволюции передачи изменений с узла на узел заключается в появлении технологии **двунаправленной логической репликации**. Технология появилась в 16-й версии PostgreSQL, то есть совсем недавно, в 2023 году. И в этот момент горячие головы энтузиастов PostgreSQL сказали: «О! Вот он — встроенный Мультимастер!». Но всё оказалось не так просто, давайте разберёмся, почему.

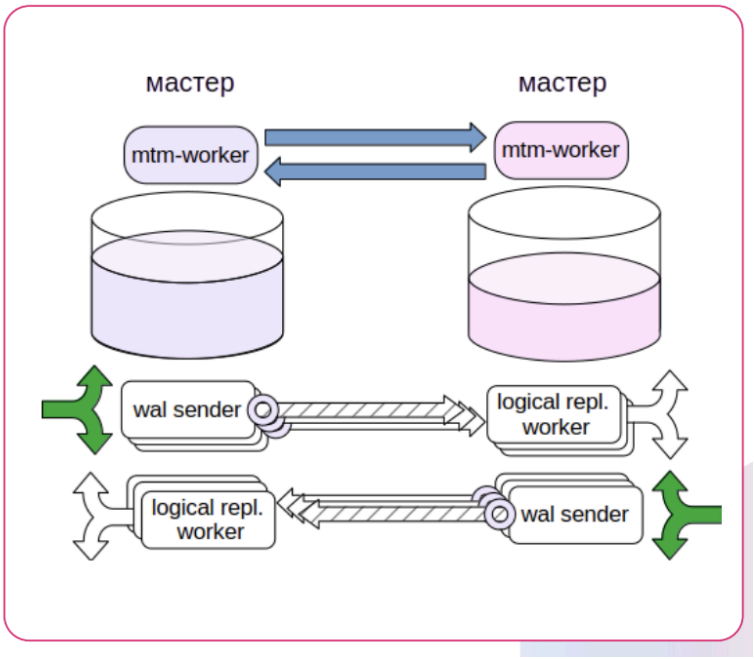
Что должно быть в «Мультимастере»?



Есть такое понятие — «Честный Мультимастер». На наш взгляд он должен обязательно отвечать следующим требованиям:

- сохранять порядок транзакций (strong consistency);
- предотвращать распределённые взаимные блокировки (conflict resolution);
- исключать split-brain (fault tolerance);
- обязательно должен быть доступен (availability) хотя бы один рабочий узел при сбоях.

А ещё хорошо бы иметь автоматическое восстановление узлов, защиту от повторного применения транзакций. Если с узлом что-то случилось, и он вдруг «сломался», то, «починившись», он должен автоматически восстановиться и снова стать рабочим узлом кластера «Мультимастер». Поэтому «Мультимастер», с нашей точки зрения, должен выглядеть следующим образом:



Если разобрать схему подробней, то в ней должны быть показаны дополнительные процессы внутри инстанса PostgreSQL. Эти процессы должны соответствовать предъявленным выше требованиям, как минимум должны быть процесс управления БД и процесс координации транзакций (для упрощения схемы приведен всего лишь один процесс — mtm-worker).

На текущий момент в Open Source можно найти примеры «Мультимастера», построенного на двунаправленной логической репликации 16-й версии PostgreSQL: Traktor, pgEdge/spock. При кратком рассмотрении этих проектов вы не обнаружите ни строгой согласованности данных, ни отказоустойчивости узлов, ни автоматического их восстановления. А главное — не будет строгой транзакционной согласованности данных в масштабах всего кластера.

pgEdge/spock — расширение, которое реализует фоновый процесс внутри PostgreSQL, а также некоторый набор модификаций (патчей) ядра PostgreSQL. Поддерживается логическая репликация из нескольких БД, некоторые виды конфликтов могут быть разрешены автоматически, для этого предусмотрено использование различных стратегий. Изюминка данного решения — реализация работы с бесконфликтными типами данных (патчи в ядро), а также доступна репликация секций партицированных таблиц и несколько других интересных возможностей, про которые можно прочитать на странице проекта на Github. Однако когда у вас возникает конфликт, разрешение которого не предусмотрено выбранной стратегией, репликация останавливается, пока вы вручную этот конфликт не устраните. Проект предлагает отказоустойчивость узлов, каждый узел в таком развёртывании будет кластером на базе Patroni. Для автоматизации развёртывания авторами написана отдельная утилита.

Tractor — другой вариант «Мультимастера» поверх встроенной двунаправленной логической репликации. Решение представляет собой внешний сервис, базирующийся на отслеживании журнала работы PostgreSQL. Если вдруг в журнале обнаруживается конфликт, то сервис пытается перезапустить логическую репликацию и требует разрешение конфликта.

Эволюционировали ли мы с 2017 года, и появился ли у нас наконец встроенный в ванильный PostgreSQL «Честный Мультимастер»? Очевидный ответ — скорее нет, чем да. Но, полагаем, что сообщество постепенно движется к его реализации.

Выше мы описали минимальные требования к «Честному Мультимастеру». Но, честно говоря, хочется большего:

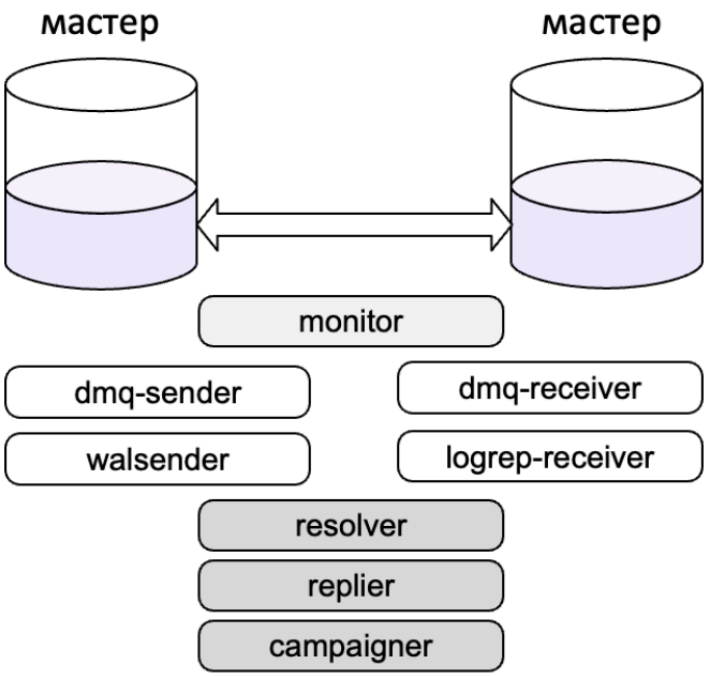
- уметь параллельно применять транзакции для ускорения работы всего кластера ;
- разрешать конфликты изменений одних и тех же данных в один момент времени, но на разных узлах;
- управлять узлом, восстанавливать его в случае сбоя, и много что ещё.

Суммируя эти требования, получаем набор специальных компонент внутри PostgreSQL.

Если смотреть с точки зрения предъявляемых требований на реализации ванильного «Мультимастера», то можно отметить его минусы:

- нет строгой согласованности данных;
- ограниченное определение и разрешение конфликтов;
- нет отказоустойчивости узлов;
- нет автоматического восстановления узлов;
- нет транзакционности в масштабах кластера.

«Честный Мультимастер»



«Честный Мультимастер» появился достаточно давно. Реализовала его компания 2nd Quadrant, которую купила EnterpriseDB. Первые версии «Мультимастера» BDR она опубликовала в 2012 году. Он устроен гораздо сложнее, чем те реализации Open Source, что мы описывали ранее.

На сегодняшний день можно утверждать, что «Честный Мультимастер» PostgreSQL доступен в виде двух коммерческих продуктов. Первый — от компании EnterpriseDB, второй — от Postgres Professional. Обе компании — крупнейшие контрибьюторы в ванильный PostgreSQL, то есть постоянно делятся своими разработками с сообществом.

Что же внедрили в оба продукта, и чем эти две реализации лучше уже доступных в Open Source?

Добавлена DML+DDL логическая репликация. Логическая репликация в ванильном PostgreSQL умеет переносить данные из базы №1 в базу №2. Но она *не умеет* переносить изменения схемы данных. Если нужно добавить атрибут в табличку, то логическая репликация тут же остановится из-за ошибки.

Распределенные алгоритмы фиксации транзакций. В «Мультимастер» добавили возможность организовать согласованный распределённый коммит между узлами кластера, чего нет в ванильном варианте двунаправленной логической репликации.

Вспомогательные фоновые процессы. Задумавшись о повышении производительности репликации, добавили вспомогательные фоновые процессы. Они выполняют работу, связанную с параллельным переносом данных и их применением на узлах кластера.

Алгоритмы разрешения конфликтов. Придумали огромное количество алгоритмов, чтобы решать возникающие конфликты.

Управляющий канал между узлами. Одно из самых важных нововведений — канал управления между узлами и соответствующий фоновый процесс для управления PostgreSQL

на узле. Это дало возможность легко собирать кластер, быстро понимать, когда в кластере произошел сбой, и нужно ли восстанавливать его работу.

Дополнительные фоновые процессы внутри PostgreSQL можно описать так:

- процессы монитора (monitor) отвечают за связь узлов друг с другом, за запуск и остановку других служебных фоновых процессов;
- процессы sender и receiver отвечают за согласованную фиксацию транзакций на узлах кластера, предотвращают несогласованное состояние базы данных между узлами;
- процесс resolver, предназначен для разрешения конфликтов и отслеживания «оборванных» транзакций, когда один из узлов кластера по каким-либо причинам стал недоступен;
- несколько других дополнительных процессов, для обработки потери узла и его восстановления.

Казалось бы, всё предусмотрено для бесперебойной и безошибочной работы кластера, но, увы, мы живём в неидеальном мире: новые возможности имеют и свои ограничения.

Особенности «Мультимастера»

Реплицируется только одна база. «Мультимастер» для одной или нескольких выбранных баз данных пока не реализован ни в Postgres Professional, ни в EDB имплементациях. Признаться, из-за этого довольно неудобно создавать «коммуналку» баз данных. Хочется, чтобы какие-то из этих баз реплицировались, а какие-то — остались локальными.

Большое количество ограничений, связанных с DDL, нет фоновых операций создания индекса. Начиная с 12-й версии PostgreSQL появилась возможность создавать и пересоздавать индексы в фоновом режиме. Если какая-то таблица раздулась (bloat), то можно запустить pg_repack, который использует фоновый процесс переиндексации. В «Мультимастере» это пока недоступно.

Обязательно требуются первичные ключи на таблицах. Для отслеживания конфликтов обязательно требуется создавать первичный ключ на таблице, а это не всегда может быть простой операцией. Порой действительно сложно придумать, какой первичный ключ требуется, настолько сложными бывают данные.

Коммерческие продукты с закрытым кодом. Единственный вариант в котором доступен «Мультимастер», и это замедляет рост популярности этой архитектуры.

Но есть и достоинства, которые дает «Мультимастер»:

- почти нулевое время простоя (режим Always On);
- бесшовное переключение на другой узел при отказе какого-либо из узлов;
- нет ограничений по версиям, можно сочетать разные версии «Мультимастера»;
- возможность масштабировать чтение;
- отсутствие ограничений на запись.

Одно из самых значимых преимуществ — возможность почти бесшовно обновлять узлы кластера. Можно снять нагрузку с одного узла, выключить его, обновить версию, запустить обратно, и узел автоматически вернётся в кластер, никто даже не заметит отсутствие такого узла.

Другое преимущество — возможность масштабирования по чтению, читать данные можно с каждого узла.

Отметим, что при попытке масштабирования по записи мы можем и не получить значимого преимущества. Даже напротив: может быть *снижение производительности*, так как при записи данных в какой-либо из узлов кластера данные должны быть синхронно перенесены в другие узлы кластера. При этом возможны случаи, когда масштабирование будет осуществляться и по записи, и по чтению. Например, приложение спроектировано так, что разные его экземпляры, подключенные к разным узлам кластера, обновляют разные части одной большой таблицы (подробнее см. главу Как выявлять конфликты?). Следовательно, для получения выигрыша следует проектировать приложение так, чтобы учитывалась архитектура СУБД.

Создание кластера

Насколько просто развернуть кластер «Мультимастера»? К примеру, «Мультимастер» по умолчанию входит в продукт Postgres Pro Enterprise, начиная с 12-й версии (актуальная версия – 16). Чтобы его установить, достаточно создать расширение (extension) multimaster на каждом узле кластера, далее вызвать команду init_cluster на одном из узлов, перечислив адреса других.

Эта команда обратится ко всем узлам кластера и соберёт кластер, готовый к эксплуатации. Можно посмотреть, как узлы общаются друг с другом с помощью команд status или nodes.

Пример уже собранного трехузлового «Мультимастера» (2 узла + 1 арбитр (referee) или 3 узла):

1. CREATE EXTENSION multimaster; (на всех узлах)
2. mtm.init_cluster('node-1','{“node-2”,“node-3”}');
3. mtm.status() / mtm.nodes();

Ожидаемый вывод последней команды:

```
 id | connected |                            conninfo
----+-----------+-----------------------------------------------------------
  1 | t         | dbname=postgres user=postgres port=5432 host=10.128.5.124
  2 | t         | dbname=postgres user=postgres port=5432 host=10.128.5.143
  3 | t         | dbname=postgres user=postgres port=5432 host=10.128.4.220
```

В EDB PGD (EnterpriseDB Postgres Distributed — PGD, с версии 3.x, актуальная — 5.3.0) развёртывание кластера выглядит примерно так же, только требуется чуть больше команд:

1. CREATE EXTENSION bdr;
2. SELECT bdr.create_node_group('group-1');
3. SELECT bdr.create_node('node-1');
4. SELECT bdr.join_node_group('group-1');
5. SELECT * FROM bdr.node_summary;

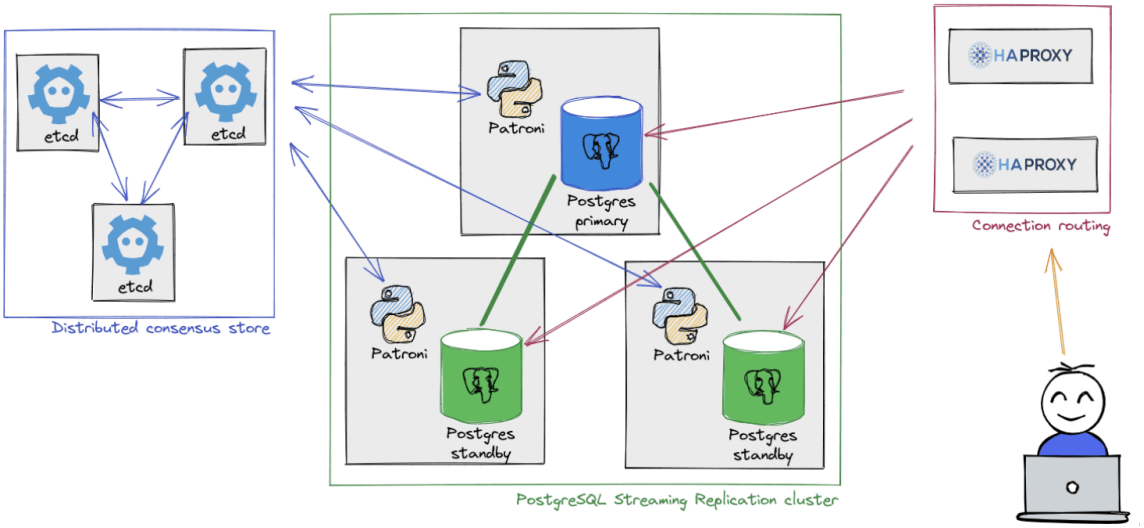
Устанавливаем расширение BDR, создаём группу узлов, в которую добавляем узлы, выводим статус кластер с помощью SQL-запроса:

```
 name   | ord | current_state | target_state |                 dsn
--------+-----+---------------+--------------+-----------------------------------------------
-
node-1 |   1 | ACTIVE        | ACTIVE       | host=node-1 port=5432 dbname=mydb user=postgres
node-2 |   2 | ACTIVE        | ACTIVE       | host=node-2 port=5432 dbname=mydb user=postgres
node-3 |   3 | ACTIVE        | ACTIVE       | host=node-3 port=5432 dbname=mydb user=postgres
```

Коллеги из EnterpriseDB пошли дальше: они создали специальную утилиту Trusted PostgreSQL Architect, отдельный инструмент для развертывания «Мультимастера» в различных окружениях, в зависимости от требований. Это могут быть контейнеры, облачные провайдеры или физические сервера. Развернуть кластер можно всего за несколько простых команд.

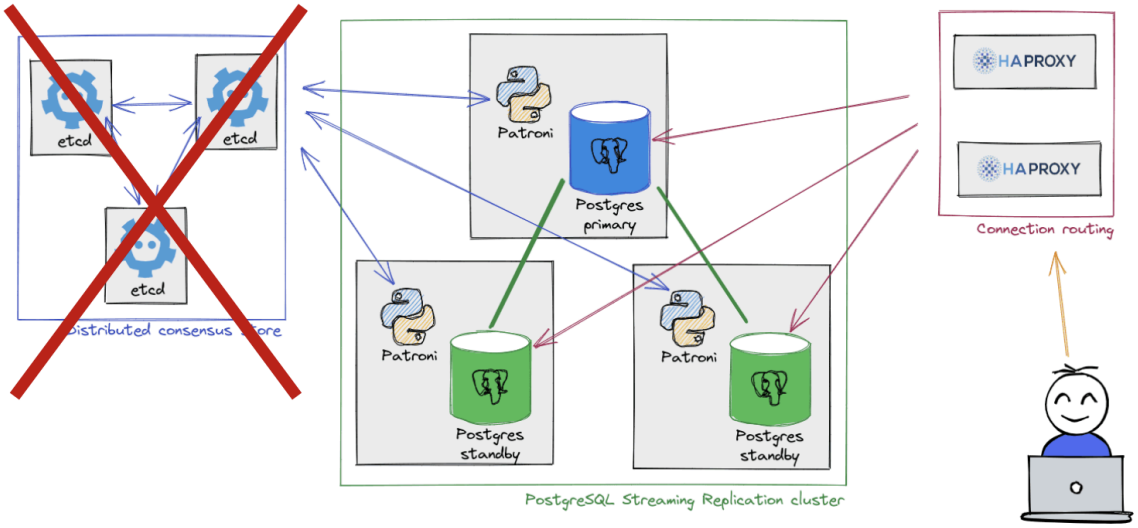
Обеспечение отказоустойчивости кластера СУБД

В мире PostgreSQL есть несколько популярных решений для построения отказоустойчивого кластера: Corosync/Pacemaker, Patroni, Stolon, pg_auto_failover и т. д. Решение на базе Patroni, пожалуй, наиболее популярно, поэтому рассмотрим схему классического кластера из его документации .



Для обеспечения безотказной работы кластера СУБД дополнительно потребуется etcd кластер: кластерное ПО хранит в нём состояния кластера СУБД и его конфигурации.

В случае с «Мультимастером» согласованный консенсус достигается внутри самого PostgreSQL, узлы умеют это делать сами за счет специальных процессов, которые за это отвечают. Тогда потребность в etcd отпадает.



Результат: на три узла (контейнеров или виртуальных машин) в инфраструктуре стало меньше! Освободились ресурсы, которые можно использовать для чего-то другого.

Так же потребуется какой-то TCP-прокси для обращения к текущему лидеру в кластере СУБД, ведь если мы обратимся к реплике с целью записи данных, то получим ошибку. Как правило, в кластерах Patroni используется HAProxy. При изменении роли узла в кластере

Patroni tcp-прокси будет направлять запросы от пользователей или приложений к новому лидеру в кластере.

Нужен ли нам tcp-прокси или балансировщик для «Мультимастера»? Требуется ли нам понимать на уровне приложения или пользователя, какова их роль узлов в кластере? Но ведь все узлы в кластере «Мультимастер» — мастера: можно подключиться к любому из них и тут же начать работать.

Правда, следует учесть особенности возможного состояния узлов при работе с кластером «Мультимастера». Например, в продукте Postgres Professional у каждого узла, кроме состояний «всё хорошо, я работаю» и «я не работаю», есть состояния, сигнализирующие о фазе восстановления, если произошел сбой:

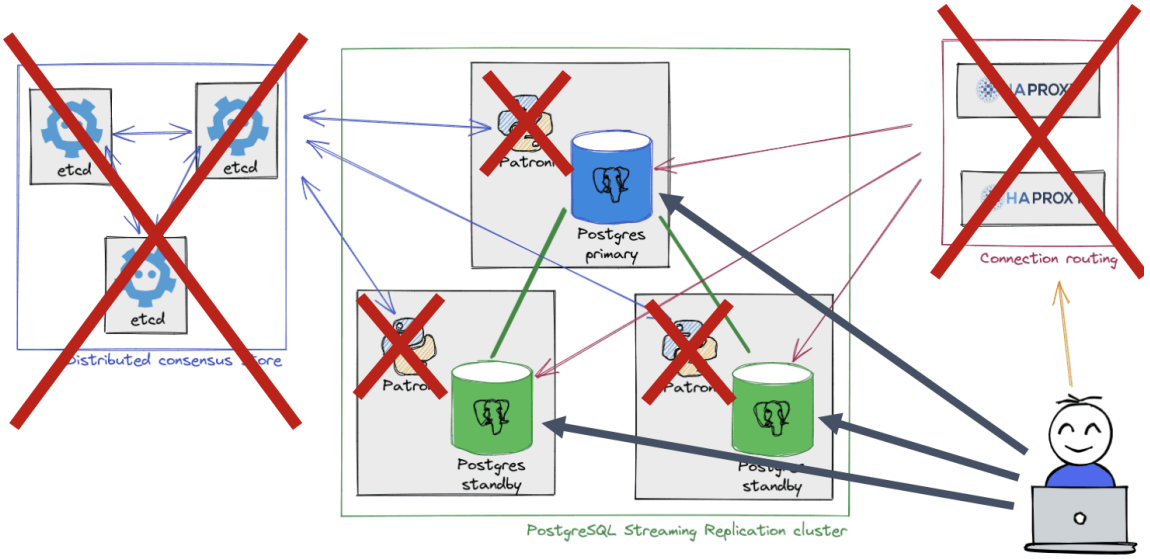
- CATCHUP – догоняем другие узлы кластера;
- RECOVERY – восстанавливаемся из журналов WAL.

В продукте от EDB названия статусов узла могут отличаться, но сути это не меняет. Если попытаться подключить приложение к такому узлу, то приложение получит ошибку: *[MTM] multimaster node is not online: current status catchup*.

Допустим, наше приложение написано на Java и работает через PostgreSQL JDBC драйвер. После создания TCP-соединения драйвер пошлёт startup-пакет, подключится к узлу, авторизуется и попробует выполнить проставление сессионных переменных. Самая популярная такая переменная — extra_float_digits, работа с данными с плавающей точкой, параметр указывает сколько цифр после запятой выдавать на экран: SET extra_float_digits = 3. На этом этапе он получит ошибку, что с узлом нельзя работать: [MTM] multimaster node is not online: current status catchup. Но есть довольно простой выход из этой ситуации: перечислим все узлы кластера в строке соединения с базой данных, например: jdbc:postgresql://host1,host2,host3/database.

Когда драйвер будет подключаться к узлу в состоянии восстановления и получит после подключения к нему ошибку, то просто перейдёт к следующему перечисленному в строке узлу до тех пор, пока не найдёт рабочий. Переподключение к БД может быть быстрым, если у нас высокоскоростная сеть и выставлены маленькие таймауты на подключение, по истечении которых мы переходим к попытке подключения к другому узлу из строки подключения в случае недоступности предыдущего. Так можно научить приложение быстро перебирать узлы кластера, минуя балансировщик.

Ура! Теперь не нужен ни кластер etcd, ни балансировщики. Убираем лишнее, оставляем только PostgreSQL и приложение: меньше компонентов в инфраструктуре — проще её архитектура, проще внедрение, меньше затраты на сопровождение.

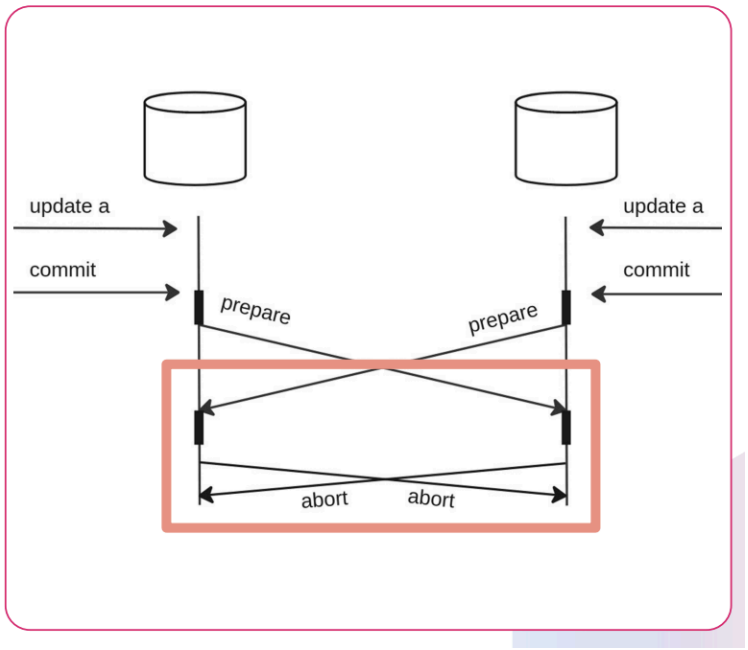


И кажется, вот оно, счастье! …Или нет?

Часть 2. Надёжность: гарантии согласованности данных и разрешение конфликтов

Важнейшая задача любой БД — надёжно хранить данные, если эта задача не решается, то тогда зачем такая база данных нужна? Если с одиночным экземпляром БД всё относительно просто, то как при распределённой модели БД гарантировать строгую консистентность и разрешать конфликты? Эти вопросы составляют предмет академических исследований, которые легли в основу реализаций алгоритмов по разрешению конфликтов.

Конфликты бывают разные. Самое распространенное определение конфликта в БД может выглядеть так: две транзакции пытаются обновить одни и те же данные в один момент времени и могут друг друга взаимно заблокировать. Каждая транзакция будет ожидать, когда другая завершит свою работу. Как в этом случае должна вести себя БД, как ей решить какая транзакция «главнее»? Прервать обе? Действовать по принципам «прав тот, кто первый» или «прав тот, кто последний»?



Увы, не существует универсальной стратегии для разрешения всех возможных вариантов конфликтов. Что же с ними происходит в «Мультимастере»? Как правильно с ним работать?

Первый и очевидный ответ: не нужно создавать ситуации, когда возникают конфликты. Правило «не создавать конфликты» относится прежде всего к проектированию приложения и к организации его работы с распределённым кластером. Общие рекомендации от вендоров (Postgres Professional, EnterpriseDB) звучат так: писать и изменять некоторые данные следует только в один из узлов кластера «Мультимастера», читать данные можно через все узлы.

Еще можно использовать бесконфликтные типы данных (в английской литературе используется аббревиатура CRDT, от Conflict-free Replicated Data Types). Кстати, на эту тему уже были хорошие публикации на Хабре, например, CRDT: Conflict-free Replicated Data Types, неплохая статья, которая может быть отправной точкой в понимании, что это такое.

Бесконфликтный реплицируемый тип данных не так давно «изобрели», если смотреть на время его появления в контексте академических исследований по теме баз данных. Марк Шапиро в 2011 году выпустил своё исследование по данной теме.

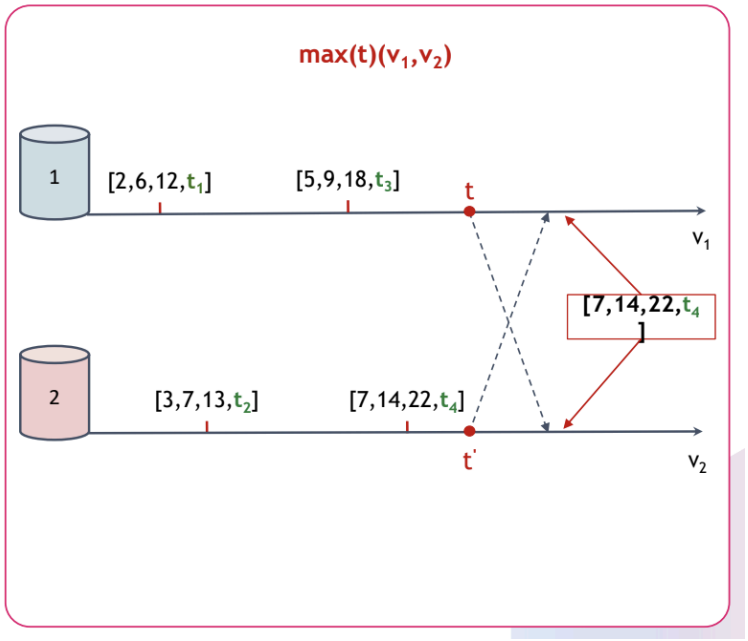
Кратко расскажем про эти типы данных. Бесконфликтные реплицируемые данные можно разделить на две большие группы: *конвергентные* (CvRDT — state-based, синхронизация состояния) и *коммутативные* (CmRDT — operation-based, синхронизация операциями).

Дефиниция бесконфликтного типа выглядит как сочетание математических операций и свойств, которые может удовлетворять избранный тип данных. Например, одновременные обновления данных могут быть коммутативны и удовлетворять равенству $a+b=b+a$.

Если применить вышеописанное к базе данных, то между узлами кластера БД можно передавать не записанные транзакцией значения, а изменения (дельты). При этом не важно, в каком порядке они приходят на каждый узел: финальный результат всегда будет одинаков. Получается, что данные гарантированно сходятся, при том, что обновления данных могут выполняться на каждом из узлов. Заметим, что данный подход к репликации, скорее свойственен БД, в которых реализована Eventual Consistency (согласованность в конечном счете). По итогу, на какой-то момент времени БД будет гарантированно консистентна. Эта концепция не всегда подходит для реляционных БД, но это не значит, что она в них не применяется.

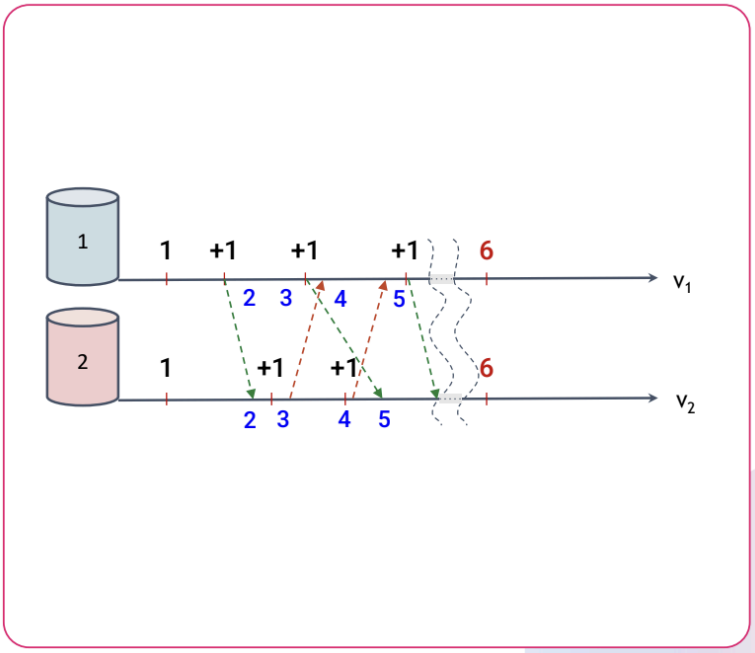
Репликация состояния возможна, когда мы оперируем, например, целыми числами или вектором состояния. Обновление состояния может происходить на любом из узлов кластера. Далее состояние распространяется, выполняется изменение на всех узлах слиянием. Каждый узел в конкретный момент времени при транзакции распространяет состояние на другой узел. Выборка состояния, применение его к другой БД, к другой реплике заключается в выборе наибольшего значения, например, по времени. Пусть у нас вектор состоит из целых чисел: x, y, z и метки времени *timestamp*, $v = [x, y, z, timestamp]$, тогда последнее актуальное значение для сохранения будет последним временем измерения — $max(t)(v1,v2)$.

Типичный пример таких данных из жизни — разного рода IoT-датчики, которые посылают либо одно измерение, либо вектор измерений. Важно, что при обращении к базе данных мы могли всегда из нее извлечь достоверное состояние нашего устройства на нужный момент времени.



Репликация изменений — другой вариант бесконфликтных реплицируемых данных. В этом случае происходит распространение операций *изменения*, которые могут быть воспроизведены на узлах кластера. В результате мы получим одинаковое результирующее состояние данных.

Первый пример, который сразу приходит в голову — монотонно возрастающая последовательность, например, счётчик. Изменение данных можно описать как *a = a + 1*. При этом счётчик может обновляться на нескольких узлах кластера. Если на каком-то узле кластера происходит изменение счётчика, то можно тут же передать это изменение на другой узел и прибавить пришедшее изменение к значению или записать его в специальную колонку. Получится то, что изображено на схеме ниже: для получения значения счётчика на какой-то момент времени каждый раз происходит вычисление значения по имеющимся дельтам. Но так как реализация этого скрыта, то для клиента это выглядит как обычное получение данных с помощью операции SELECT.



На сегодняшний день известно некоторое количество CRDT, для некоторых в PostgreSQL существует реализация, что упрощает и ускоряет репликацию данных для «Мультимастера» (EDB, pgEdge). Как уже было сказано, не для всех типов данных подходит бесконфликтная репликация, следовательно, всё равно требуется каким-то образом выявлять конфликты, которые неизбежно появляются.

Как выявлять конфликты?

Рассмотрим конфликты с точки зрения работы логической репликации. Все изменения переносятся с узла на узел *построчно*. Коротко подход к выявлению конфликтов можно описать так:

1. При изменении данных в какой-либо строке в БД на одном узле (назовём его «узел-1») изменения передаются на другие узлы кластера с инструкцией: «найди эту строку в БД и поменяй в такой-то колонке значение **X** на значение **Y**». Чтобы найти эту строку на стороне приемника («узел-1» — источник, другие узлы кластера — приемники данных), нужен *первичный ключ*.

2. Далее первичный ключ используется при *проверке строк на блокировку*. Если кто-то ещё попытается изменить строку, но на другом узле, то на «узел-1» придёт сообщение, похожее на предыдущее: «найди эту строку в БД и поменяй в такой-то колонке значение *X* на значение *Z*». Так как у нас образуются две транзакции, изменяющие одни и те же данные, то конфликт, который нужно как-то разрешить.

3. «Узел-1» может проигнорировать блокировку, пришедшую из другого узла кластера, а может *установить блокировку* или *ожидать*, пока транзакция, пришедшая с другого узла, не будет зафиксирована, в надежде, что не случится глобальной взаимоблокировки (deadlock). Такое тоже возможно, например, когда какая-то из транзакций «откатилась».

4. В моменты, когда проверяются или берутся блокировки, происходит *определение конфликта* (conflict detection) и его последующее разрешение (conflict resolution). Последнее

может происходить и асинхронно, когда узлы сохранили данные в один момент времени, при этом разрешение конфликта отложили «на потом», сохранив и информацию о конфликте.

Конфликт определён: что же делать дальше?

Как разрешать конфликты?

Важно понять, какая из транзакций в конфликте «главнее»: для этого в «Мультимастере» реализован своеобразный «суд». При вынесении вердикта используется различная информация, например, откуда пришла строка, origin — источник реплицируемых данных, приоритет узла, или версия строки (row version), если версия старше, то отбрасываем младшее значение.

В реализации «Мультимастера» Postgres Professional пользователь принимает решение о способе разрешения самостоятельно (то есть настраивает способы разрешения конфликтов). При конфликте можно:

- вернуть ошибку транзакции в сессию (ERROR);
- пропустить эти изменения (SKIP) — лучше ничего не блокировать;
- обновить эту строку (UPDATE).

Помимо этого, есть специальная настройка, контролирующая процесс проверки и разрешения конфликтов — multimaster.deadlock_prevention. Она необходима, чтобы не случилось взаимоблокировки (deadlock). Если установлено значение «off» — предотвращение взаимоблокировок отключено. Если установлено значение «simple» — конфликтующие транзакции отклоняются. Если установлено значение «smart», то для улучшения доступности ресурсов специальный алгоритм выбирает, какие транзакции фиксировать, а какие отклонить.

Проблема взаимоблокировок сложно разрешается, также это довольно «дорогая» операция, поэтому их следует максимально избегать.

Один из довольно остроумных способов избегания блокировок и борьбы с ними реализован в EnterpriseDB PGD (в Postgres Pro пока, увы, нет) — Column-Level Conflict Resolution (разрешение конфликтов на уровне колонок).

Column-Level Conflict Resolution позволяет двум транзакциям бесконфликтно обновлять одну и ту же строчку, но обновляемые данные должны находиться в разных колонках. Под «капотом» этого метода используется отслеживание времени изменения данных для каждого столбца в таблице.

Для таблицы, где нужно таким образом разрешать конфликты, нужно выполнить ALTER TABLE <table_name> REPLICA IDENTITY FULL. Сама по себе эта функциональность не нова, она реализована в ванильном PostgreSQL для корректной работы логической репликации. Атрибут IDENTITY FULL позволяет изменять информацию, записываемую в журнал предзаписи для идентификации изменяемых или удаляемых строк.

Рассмотрим краткий пример. Создадим таблицу t(id bigint primary key, a int, b int), вставим в нее строку с id=1, попробуем обновить одну и ту же строку, но будем обновлять отдельными запросами данные в колонках *a* и *b*. Посмотрим на разные стратегии разрешения конфликта при обновлении строки.

Итак:

1. create table t(id bigint primary key, a int, b int);
2. insert into t values (1,0,0);
3. ALTER TABLE t REPLICA IDENTITY FULL;
4. Узел 1: update t set a = 5 where id = 1;
5. Узел 2: update t set b = 7 where id = 1;

При стратегии *«ошибка»* (error) у нас откатятся обе транзакции на обоих узлах.

При стратегии *«последний коммит выигрывает»* (last commit wins) получим результат — (1,0,7).

Для многих вариантов поведение, описанное абзацем выше — и желаемое, и ожидаемое. Однако в некоторых случаях это может стать проблемой: например, есть многоузловой кластер СУБД. Каждая часть приложения подключена к отдельному узлу такого кластера и обновляет выделенное подмножество столбцов в общей таблице. В этом случае могут возникнуть конфликты, которых мы хотим избежать.

При выборе стратегии *«последний коммит выигрывает»* изменения данных в разных столбцах могут быть перезаписаны.

При стратегии *«column-level»* у каждой колонки будет временная метка, отражающая время последней ее модификации. Если временные метки не совпадают, то можно разрешить конфликт и получить консистентные данные: две транзакции обновили одну и ту же строку, но разные колонки, результат — (1,5,7).

Непреднамеренное повреждение данных (Silent Data Corruption)

Представим, что у нас есть кластер «Мультимастера» из двух узлов и размер БД в 10 Тб. Так как в жизни бывает всё что угодно, мы не можем быть на 100% уверены, что в БД одного из узлов случайно не прилетит «излучение из космоса» и не испортит данные. Может возникнуть ситуация, когда в одном кластере на разных узлах находятся разные данные, что нарушает принцип консистентности.

Не так давно, в октябре 2023 года, коллеги из Alibaba Cloud опубликовали статью «Understanding Silent Data Corruptions in a Large Production CPU Population». Наиболее близкий по смыслу перевод может выглядеть как «Изучение скрытых повреждений данных на большом числе вычислительных узлов» с результатами проверки своих дата-центров.

Исследователи измеряли, как часто может происходить Silent Data Corruptions (скрытые повреждения данных). Проверка дисков и процессоров показала, что некоторые процессоры из общего их количества неправильно вычисляют контрольные суммы. Отклонение составляет несколько базисных пунктов (для справки: % — процент = 1/100, ‰ — промилле = 1/1000, ‱ — базисный пункт = 1/10000). Как можно доверять процессорам, которые в одном случае из 10000 неправильно считают контрольную сумму? Понятно, что у Alibaba дата-центры занимают огромную площадь, поэтому попадание условного «космического луча» в какую-нибудь ячейку памяти более вероятно, чем при эксплуатации единичного сервера. Всё вышесказанное навело разработчиков на мысль, что в реализации «Мультимастера» должен быть инструментарий для проверки согласованности данных. Эти инструменты помогут убедиться, что данные на всех узлах консистентны.

Проверка согласованности данных на узлах

В EnterpriseDB PGD для этого есть отдельный продукт Live Compare, который может сравнивать несколько баз данных кластера «Мультимастера» целиком на предмет аномалий.

В документации EnterpriseDB сказано, что аномалии расхождения данных самые тяжелые, потому что их сложнее всего устранять. Для решения такой задачи нужны критерии, на основе которых можно достоверно определить, какие данные актуальные при сравнении объектов в базе данных разных узлов, а какие нет. Составление таких критериев — довольно непростой труд.

В Postgres Pro Enterprise для такой проверки есть отдельная функция: mtm.check_query('SELECT * FROM table ORDER BY id'). Она позволяет сделать снимок данных (транзакционный снимок) выбранной для верификации таблицы на всём кластере «Мультимастера», после чего сравнивает данные в пределах одного снимка.

Какие еще есть способы разрешения конфликтов?

Partner или Smart. Для этого варианта разрешения конфликтов кластер «Мультимастера» должен состоять из двух узлов, каждый из которых проверяет список статус транзакций другого. Он имеет представление о том, какие транзакции прошли, и с каким результатом они завершились. Если транзакция откатилась или, наоборот, повторяется, то может возникнуть конфликт. Узел на котором она выполняется может посмотреть в память соседнего узла и принять решение о текущей транзакции на основе этого знания.

CAMO (Commit At Most Once). Это логичное развитие предыдущей идеи, которое решает вопрос производительности: при таком способе фиксация транзакции должна выполняться не более одного раза. Представим, что есть приложение, которое делает COMMIT, но при этом ответа от сервера не дожидается (например, что-то с сетью): что мы должны предпринять в следующий раз? Скорее всего, заново повторим транзакцию. Но вдруг эта транзакция уже зафиксирована? Тогда возникнет конфликт. В данном случае приложение является третьим участником нашего кластера, который может принять участие в консенсусе при коммите транзакции и знает о судьбе предыдущего коммита. От рабочего узла кластера он получает сведения, что коммит предыдущей транзакции прошёл успешно.
Eager Replication — определение потенциально конфликтующих транзакций с помощью консенсуса между узлами кластера. Реализованы распределенные транзакции с использованием протокола распределённого консенсуса RAFT.
Conflict triggers — создание специальных триггеров для разрешения конфликтов в особенных ситуациях.

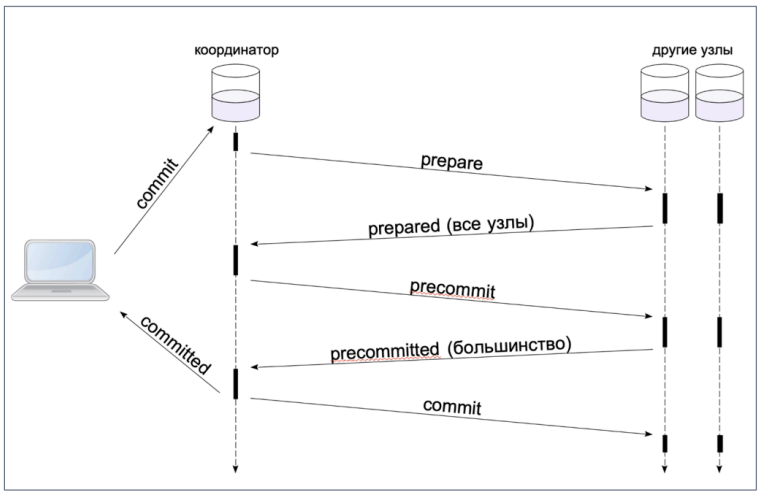
Устойчивость к сбоям

В «Мультимастере» при фиксации транзакций (глобальный коммит) для решения проблемы устойчивости к сбоям используют протоколы распределенного консенсуса:

- Протоколы консенсуса RAFT (EDB) и PAXOS (Postgres Pro);

- 2PC (двухфазный коммит);
- Различные их комбинации с описанными выше способами разрешения конфликтов.

При использовании распределенных протоколов консенсуса классическая схема взаимодействия между узлами кластера при глобальном коммите может выглядеть так, как изображено на схеме ниже. Координатор на данной схеме – узел который инициировал транзакцию. В кластере «Мультимастера» координатором может быть любой узел.



Транзакции должны быть зафиксированы на всех узлах кластера, использование для этого протоколов консенсуса может привести к двукратному или трехкратному сетевому обмену между узлами кластера. Схемы глобального коммита постоянно оптимизируются, для снижения количества шагов при глобальной фиксации транзакций в «Мультимастере» Postgres Professional на сегодняшний день это всего два шага.

Резервное копирование и восстановление

Допустим, в кластере «Мультимастера» есть 3 узла (как мы надеемся, с одними и теми же данными). С какого узла нужно сделать резервную копию? Как следует восстанавливать весь кластер ?

В кластере «Мультимастера» добавление нового узла происходит так:

- снимается физическая резервная копия с любого рабочего узла;
- резервная копия разворачивается на новом узле и запускается сервер СУБД;
- новый узел добавляется в кластер, при этом новый узел «догоняет» другие узлы.

Очевидно, что идентификатор БД всех узлов кластера — один и тот же. Поэтому резервную копию можно сделать с любого узла. Восстановление узла из резервной копии (если только мы напрямую не клонируем каталог БД с других узлов кластера) происходит для «Мультимастера» Postgres Professional следующим образом:

- восстанавливаем каталог БД из резервной копии на нужном узле;
- запускаем экземпляр сервера СУБД;
- удаляем служебные метаданные «Мультимастера» из восстановленного узла;
- выполняем добавление узла в кластер — SELECT mtm.join_node(node-id,'LSN');

Теперь необходимо дождаться, когда узел получит накопившиеся изменения за время, пока он отсутствовал в кластере, данный режим работы узла называется CATCHUP.

Часть 3. Производительность: почему Мультимастер замедляет транзакции?

И вот мы подошли к самому вкусному — к производительности. Всё, что было сказано выше — хорошие и надёжные механизмы по обеспечению консистентности данных и разрешению конфликтов. Однако любые механизмы по обеспечению надёжности, как известно, отражаются на производительности.

Описав механизм глобального коммита, разрешения конфликтов, возникают вопросы: «Насколько всё это быстро происходит?», «Где стоит применить «Мультимастер?», потому что он точно не про «скорость без границ».

Представим, что к нам пришел заказчик с требованиями:

- необходим кластер «Мультимастера» из двух или трёх узлов;
- кластер предназначен OLTP-процессингу;
- закрытая модель нагрузки в десятки параллельных потоков;
- требования к средней задержке фиксации транзакции (commit average latency) — 300 мс;
- транзакционная нагрузка — процессинг JSON-файлов, размер каждого около 20 Мб.

В первых тестах с «Мультимастера» для этой задачи результат был катастрофически медленным, всё работало в 3-4 раза медленнее по сравнению с одиночным экземпляром СУБД:

- «Мультимастер», 3 узла — 800 мс;
- Postgres Pro Enterprise, 1 узел — 250 мс;

Заказчик выразил мнение, что такая низкая скорость — из-за надёжности «Мультимастера» и его супер возможностей. «Да ваш «Мультимастер» — просто черепаха!» — сказал нам заказчик. И мы пошли разбираться, где же он притормаживает.

Почему может тормозить «Мультимастер»?

Настройка объёма информации, записываемой в WAL-журналы. При включении дополнительного уровня логирования данных в WAL-файлы с информацией для логического декодирования, получили *замедление в три раза*. Хотя при таком уровне логирования в журналы добавляется всего пара записей, и WAL-файлы не раздуваются в размерах.

Проигрывание изменений на всех узлах. Изменения данных с первого узла применяются на втором узле, а следом применяются и на третьем. Но на третьем узле мы применим изменения за то же время, да и изменения на обоих узлах применяются параллельно. Откуда возникло замедление в 3 раза?

Глобальный коммит. Все узлы при применении изменений должны проголосовать по протоколу PAXOS, договориться о согласованном применении изменений: это требует дополнительные временные затраты, но они невелики.

В ходе разбирательства нашлись проблемы как в настройках «Мультимастера», так и в способах работы с ним со стороны приложения.

Проблема №1. Таблицы без Primary Key

Если на узел кластера приходит строка изменения без первичного ключа, то приходится делать полное сканирование таблицы и искать, в какой строке появились изменения. В «Мультимастере» Postgres Pro есть специальная настройка: если есть таблицы без первичного ключа, то их можно игнорировать. Тогда они будут реплицироваться: *multimaster.ignore_tables_without_pk*.

Конечно, эта опция выглядит, как опция для параноиков. Главное, чтобы не было сильной просадки производительности. Поэтому, если мы хотим реплицировать все таблицы, нужно обязательно создавать для всех таблиц первичные ключи.

Проблема №2. Время фиксации при глобальном коммите

Далее стали разбираться, что происходит с фиксацией транзакций в кластере: извлекли информацию из WAL-файлов, когда происходили изменения, когда начинался и заканчивался распределенный коммит. Выяснилось, что множество транзакций изменяли всего лишь одну строку, но при этом время глобального коммита было в 10 раз больше, чем время самой транзакции.

```
XID: 168838657
Start time (estimated): 2022-11-09 13:18:43.302
Prepare: 2022-11-09 13:18:43.302
Commit: 2022-11-09 13:18:43.311
Duration: 00.010
Waiting global commit: 00.009

1 table insert
1 index insert
90-100%
```

Просмотрели информацию о нескольких транзакциях: везде одна и та же картина. Поинтересовались у разработчиков, что именно фиксируется в транзакциях. Логично было собирать небольшие изменения JSON (пусть и размером в 20 Мб) в «пачку» и фиксировать их одним коммитом. Оказалось, что на стороне Java-приложения был включен автокоммит, то есть *каждая* DML-операция создавала распределённый глобальный коммит. После рекомендации выключить автокоммит ситуация нормализовалась:

```
XID: 168839109
Start time (estimated): 2022-11-09 13:18:43.667
Prepare: 2022-11-09 13:18:44.038
Commit: 2022-11-09 13:18:44.183
Duration: 00.516
Waiting global commit: 00.145

128 table inserts
64 table updates + 64 table locks
384 indexes insert
28%
```

«Мультимастер» стал работать очень быстро, время глобального коммита резко сократилось и стало занимать не 99% времени от всей транзакции, а всего лишь 28%, а это уже сравнимо с работой обычной физической репликации.

Проблема №3. Производительность сети виртуальной машины[2]

Кластер «Мультимастера» был развёрнут в среде виртуализации. У виртуальной машины QEMU был виртуальный сетевой интерфейс virtio-net-pci, драйвер которого по умолчанию работает в однопоточном (или одноочередном) режиме.

Чтобы глобальный коммит происходил быстро, требуется, чтобы сеть отвечала практически мгновенно. Скорость глобального коммита при решении Проблемы №2 хоть и уменьшилась, но так и не достигла требуемых значений. При дальнейших исследованиях причин мы заметили, что скорость коммита зависит от производительности сетевого устройства. Мы переключили драйвер virtio-net-pci[3] в многопоточный (многоочередный) режим, включили multiqueue на всех узлах кластера и выставили количество очередей в значение 4. После этих манипуляций время выполнения коммита уменьшилось и достигло требуемых заказчиком значений.

Аварийное нарушение работы логической репликации

В кластере «Мультимастера» произошла аварийная ситуация. Общую картину может дать фрагмент журнала работы сервера (отредактирован и сокращен для лучшего понимания):

14:52:07 – нарушение работы логической репликации, «узел-2» не забирает изменения от «узла-1»

15:48:52 – диск pg_wal на «узле-1» полностью заполняется WAL-файлами и уходит в перезагрузку

15:49:28 – «узел-1» запускается и переходит в catchup. Рабочая нагрузка переходит на «узел-2»

16:29:50 – диск pg_wal на «узле-2» полностью заполняется WAL-файлами

В один прекрасный момент логическая репликация, которая выбирала данные для узла-2, перестала работать. Причина: на стороне приёмника «узла-2» что-то пошло не так. На «узле-1» начали копиться WAL-файлы. Они копились до тех пор, пока дисковое пространство выделенное под хранение WAL-файлов, не закончилось. Как только оно закончилось, «узел-1» ушёл в перезагрузку, а затем перешёл в режим восстановления. «Узел-2» ждёт, когда «узел-1» вернётся в нормальное состояние и тоже накапливает WAL-файлы для этого узла. Накапливает он их до тех пор, пока и у него не заканчивается дисковое пространство — так весь кластер «Мультимастера» перешёл в неработоспособное состояние.

---

[2] Для физических серверов, разумеется, будут другие рекомендации, они выходят за рамки данной статьи.

[3] Тут стоит отметить, что заставить virtio-net-pci работать в режиме multiqueue далеко не всегда просто: как минимум сетевой адаптер хоста виртуализации должен поддерживать данный режим.

У этой проблемы есть довольно простое решение: в PostgreSQL есть настройка, с помощью которой можно указать, какое количество WAL-файлов необходимо хранить до того момента, пока слот репликации не остановится: *max_slot_wal_keep_size* — ограничение размера слотов репликации. Если вдруг WAL-файлов накопилось больше указанного размера, то следует найти слот, который заставляет держать WAL-файлы, и удалить его. Пусть лучше остановится работа логической репликации, но «Мультимастер» продолжит работать. Поэтому ограничиваем размер хранимых WAL-файлов, предотвращая возможное возникновение аварийной ситуации. Теперь обратимся к вопросу производительности.

Производительность: рекомендации

Чтобы добиться производительности «Мультимастера» можно применить следующие рекомендации:

- обязательно наличие первичного ключа (primary key) на реплицируемых таблицах;
- делать более объемные транзакции, чтобы глобальный коммит стал «дешевле». Чем объёмнее транзакция — тем меньше время её фиксации (commit);
- настроить сеть виртуальных машин под требуемую сетевую нагрузку;
- выставить max_slot_wal_keep_size, чтобы кластер «Мультимастера» не уходил в отказ из-за переполнения диска.

Дополнительно можно принять во внимание такие замечания:

Стремитесь избегать конфликтов на уровне кода на разных узлах: разрешение конфликтов — «дорогая» операция. Какой способ разрешения будет выбран, настолько «дорогим» и долгим будет время этой операции.

Синхронизируйте время на узлах кластера. «Мультимастер» использует глобальные распределенные коммиты, построенные на протоколе распределённого консенсуса PAXOS. Этот консенсус использует временные метки. Время должно быть синхронизировано с одним источником времени на всех узлах кластера. Если будет расхождение времени на узлах, то возникнут дополнительные задержки при фиксации транзакций, что сильно снижает производительность.

Внимательно отнеситесь к требованиям со стороны службы безопасности. Антивирус отслеживает сетевую активность между узлами кластера и изменения в каталоге БД, auditd-правила следят за бинарными файлами экземпляра СУБД. Эта активность со стороны ПО, обеспечивающего безопасность, всегда отражается на производительности СУБД, и не в лучшую сторону. Одним из требований со стороны ДБА к службе ИБ может быть исключение сетевой активности внутри кластерного трафика и каталога БД из области наблюдения средств безопасности.

Крупные миграции данных следует делить на части. Если у вас есть огромная миграция (сотни, а то и тысячи миллиардов строчек), то постарайтесь разделить её на части. Когда вы сделаете одну большую транзакцию, её нужно потом логически декодировать в памяти: это зачастую приводит к срабатыванию OOM-киллера или повышенной нагрузке на все узлы кластера.

Итоги

Сначала о том, чем закончился реальный кейс с заказчиком: время транзакции упало с 800 мс до 300 мс, система успешно прошла все тесты и перешла в промышленную эксплуатацию. Мы получили ценный опыт и знаем теперь, какие настройки надо обязательно сделать сразу, чтобы кластер «Мультимастера» работал стабильно.

Исходные показатели производительности:

- Фиксация транзакций в кластере «Мультимастер» — 800 мс;
- Фиксация транзакций в одиночном экземпляре Postgres Pro Enterprise —250 мс.

Достигнутые результаты:

- Фиксация транзакций в кластере «Мультимастер» — 300–350 мс;
- Система в промышленной эксплуатации на Postgres Pro Enterprise Multimaster;
- Все приобрели ценный опыт.

Подведем итоги нашей статьи. В начале нее мы говорили о мифах «Мультимастера». Напомним, что «Мультимастер»:

- не про масштабирование записи, но масштабирование чтения;
- не про пиковую производительность, но для нагруженных систем;
- не про надёжность;
- не про доступность;
- не для промышленного применения.

В итоге можно уверенно утверждать, что реальность «Мультимастера» (от Postgres Pro):

- не про масштабирование записи, а про масштабирование чтения;
- не про пиковую производительность, а для нагруженных систем;
- ~~не про надёжность~~ надежен;
- ~~не про доступность~~ уровень доступности может достигать 99,99*%;
- ~~не для промышленного применения~~ проверенное промышленное решение.

Реализация технологии «Мультимастер» в Postgres Pro Enterprise позволяет добиться высокой доступности («AlwaysOn» архитектура, база данных доступна всегда). Период недоступности классического кластерного ПО («Active-Passive» архитектура) можно рассчитать, как сумму времён, затраченных на определение кластерным ПО сбоя мастер-узла, выборы нового мастера, переключение роли выбранного узла, перенастройку (пусть и автоматическую), TCP-прокси, переподключение приложения к новому «мастеру».

В кластере «Мультимастера» приложение просто подключится к другому узлу кластера. Время на такое переподключение может отличаться на один-два порядка в меньшую сторону от времени переподключения при использовании классического кластерного решения.

Надёжность, по сравнению с любым кластерным решением, основанным на физической репликации, обеспечивается встроенными протоколами распределенного консенсуса при глобальной фиксации транзакций.

Решения на базе «Мультимастера» уже внедрены в промышленную эксплуатацию весьма серьезными заказчиками (рассказали бы, но, увы, вынуждены соблюдать NDA). Кластер «Мультимастера» уверенно можно использовать в промышленных решениях!

Материалы для изучения

«Мультимастер» последней версии — проверенное решение, которое можно использовать в продуктиве. Раньше мы упоминали, что Postgres Pro Multimaster — продукт с закрытым кодом, но это не значит, что прежде чем решить, нужен он вам или нет, вы не можете о нем ничего узнать.

Для изучения технологии и ее реализации есть:

- Бесплатные учебные курсы по PostgreSQL
- Бесплатный курс по Postgres Pro Enterprise
- В курсе по Postgres Pro Enterprise раздел – «Синхронный кластер Мультимастер»
- Учебная виртуальная машина с Postgres Pro Enterprise
- Документация по Postgres Pro Enterprise

Вы можете загрузить себе учебную виртуальную машину, запустить «Мультимастер» и попробовать всё, о чём мы рассказали в данной статье. Кроме того, Postgres Pro Enterprise может быть выдан и бесплатно для тестирования (необходимо связаться с отделом продаж компании).

Материалы для более вдумчивого изучения технологии «Мультимастер» и связанными с ней темами:

- Conflict-free Replicated Data Types. Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski
- https://crdt.tech
- PostgreSQL и отказоустойчивость. Михаил Кулагин, PgConf, Москва, 2017
- Привет, встроенный Мультимастер? Сравнение двунаправленной репликации в ваниле и Postgres Pro Multimaster. Андрей Рудометов, PgConf, Москва, 2023
- Workshop Logical Replication Deep Dive. Peter Eisentraut
- Supporting multi master replication with Geosharding in PostgreSQL. Hari Kiran
- Supporting multi-active replication with Geosharding in PostgreSQL. Denis Lussier
- Pglogical and Postgres BDR update. Simon Riggs
- Bi-directional replication using origin filtering in PostgreSQL 16. Vigneshwaran C
- EnterpriseDB Postgres Distributed

Карточки

Карточка 1

«Мультимастер»: великий и ужасный! Об этой технологии часто говорят, но многие до сих пор уверены, что она сырая и не подходит для продакшена. Так ли это? Давайте посмотрим как развивалась технология «Мультимастер» в экосистеме PostgreSQL и остановимся на том, что она из себя представляет на сегодняшний день, на каких внутренних механизмах PostgreSQL она основана и для чего может быть востребована. А ещё поговорим о «Честном Мультимастере». О его заслугах и проблемах, и о том как сделать «Мультимастер» производительным и правильно с ним обращаться.